PHP & MYSQL

NOVICE TO NINJA

SEVENTH EDITION

BY TOM BUTLER

GET UP TO SPEED WITH PHP THE EASY WAY

PHP & MySQL: Novice to Ninja, 7th Edition

Copyright © 2022 SitePoint Pty. Ltd.

- **Product Manager:** Simon Mackie
- **Technical Editor:** Tim Boronczyk
- **English Editor:** Ralph Mason
- **Cover Designer:** Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.
10-12 Gwynne St,
Richmond, VIC, 3121
Australia
Web: www.sitepoint.com
Email: books@sitepoint.com
ISBN 978-1-925836-46-2 (print)
ISBN 978-1-925836-47-9 (ebook)

Printed and bound in the United States of America

About Tom Butler

Tom Butler is a web developer and university lecturer. He has a PhD in the area of software engineering best practices and enjoys evaluating different approaches to programming problems.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit https://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Preface xv

Who Should Read This Book xvi

Programming Has Changed xvii

It Takes 10,000 Hours to Become an Expert xviii

Resist the Temptation to Skip Ahead xviii

The Concorde Fallacy xviii

You're Not Learning PHP xix

Getting Braces and Semicolons in the Right Place Is the Easy Part xx

You Won't Get Anything Done by Planning xxi

Where to Find Help xxv

Your Feedback xxvi

Let's Go xxvi

Chapter 1: **Installation** 1

Your Own Web Server 3

Server Setup 1: Manually Installing All the Software Components 4

Server Setup 2: Pre-packaged Installations 5

Server Setup 3: Virtual Servers 6

Server Setup 4: Docker 8

Getting Started 9

Installation on Windows 9

Installation on macOS 11

Installation on Linux 14

Getting Started with Docker 15

Connecting to the Server and Creating Your First File 20

We're All Set Up 23

Chapter 2: Introducing PHP **24**

Basic Syntax and Statements 28

Variables, Operators, and Comments 34

Variables 34

Operators 35

Control Structures 36

If Statements 37

Loops 43

Arrays......54

User Interaction and Forms......61

Passing Variables in Links......62

Passing Variables in Forms......71

Hiding the Seams......75

PHP Templates......76

Security Concerns......79

Many Templates, One Controller......83

Bring On the Database......88

Chapter 3: **Introducing MySQL......89**

An Introduction to Databases......90

MySQL......92

Using MySQL Workbench to Run SQL Queries......93

Creating a Database......100

Structured Query Language......103

Creating a Table......106

Adding Data......112

A Word of Warning......118

Viewing Stored Data......119

Modifying Stored Data 124

Chapter 4: **Publishing MySQL Data on the Web** 128

The Big Picture 129

Creating a MySQL User Account 131

Connecting to MySQL with PHP 132

A Crash Course in Object-oriented Programming 140

Configuring the Connection 143

Sending SQL Queries with PHP 147

Handling SELECT Result Sets 151

Thinking Ahead 160

Inserting Data into the Database 172

Magic Quotes 177

Prepared Statements 178

Deleting Data from the Database 183

Mission Accomplished 193

Chapter 5: **Relational Database Design** 194

Giving Credit Where Credit Is Due....195
Rule of Thumb: Keep Entities Separate 198
SELECT with Multiple Tables....205
Simple Relationships....210
Many-to-many Relationships....213
One for Many, and Many for One....218

Chapter 6: Structured PHP Programming 219

Include Files 221
Including HTML Content....221
Including PHP Code 224
Types of Includes 227
Custom Functions and Function Libraries 229
Variable Scope.... 232
Breaking Up Your Code into Reusable Functions....241
Using Functions to Replace Queries.... 242
Updating Jokes.... 245
Editing Jokes on the Website 246
Delete Function....250

Select Function ... 251

The Best Way ... 252

Chapter 7: **Improving the Insert and Update Functions** ... 254

Improving the Update Function ... 255

Improving the Insert Function ... 261

Handling Dates ... 264

Displaying the Joke Date ... 272

Making Your Own Tools ... 274

Generic Functions ... 276

Using These Functions ... 285

Repeated Code Is the Enemy ... 290

Creating a Page for Adding and Editing ... 292

Further Polishing ... 298

Moving Forward ... 303

Chapter 8: **Objects and Classes** ... 304

Time for Class ... 306

Public vs Private ... 310

Protected 311

Objects 312

Class Variables 313

Constructors 321

Type Hinting 325

Private Variables 328

Constructor Property Promotion 330

Using the `DatabaseTable` Class 331

Updating the Controller to Use the Class 336

DRY 339

Creating a Controller Class 341

Single Entry Point 347

Keeping it DRY 354

Template Variables 359

Be Careful with `extract` 363

Summary 366

Chapter 9: Creating an Extensible Framework 367

Search Engine Optimization 369

Thinking Ahead: User Registration ... 375
1. Include the Relevant Controller ... 377
2. Create an Instance of the Controller ... 377
Dependencies ... 378
3. Call the Action on the Correct Controller ... 382
Done ... 383
URL Rewriting ... 385
Tidying Up ... 398
Make it OOP ... 398
Reusing Code on Different Websites ... 401
Generic or Project-specific? ... 402
Making `EntryPoint` Generic ... 405
Autoloading ... 414
Case Sensitivity ... 416
Implement an Autoloader ... 417
Redecorating ... 418
Namespaces ... 420
Autoloading with PSR-4 ... 424
And the REST ... 428
Non-web Applications ... 431

Enforcing Dependency Structure with Interfaces 433
Error Handling 439
Your Own Framework 443

Chapter 10: Allowing Users to Register Accounts 445

Validating Email Addresses 454
Preventing the Same Person from Registering Twice 455
Securely Storing Passwords 456
Registration Complete 460
Chapter Summary 462

Chapter 11: Cookies, Sessions, and Access Control 463

Cookies 465
PHP Sessions 474
Counting Visits with Sessions 478
Access Control 479
Logging In 479

Protected Pages ... 488
Creating a Login Form ... 496
Logging Out ... 501
Assigning Added Jokes to the Logged-in User ... 504
User Permissions ... 507
Mission Accomplished? ... 510
The Sky's the Limit ... 513

Chapter 12: **Relationships** ... 515
Object Relational Mappers ... 522
Public Properties ... 525
Methods in Entity Classes ... 526
Using Entity Classes from the DatabaseTable Class ... 531
Joke Objects ... 539
Using the Joke Class ... 543
References ... 544
Simplifying the List Controller Action ... 546
Tidying Up ... 398
Caching ... 551

Joke Categories ... 553

List Page ... 559

Assigning Jokes to Categories ... 560

Assigning Categories to Jokes ... 574

Displaying Jokes by Category ... 576

Editing Jokes ... 582

User Roles ... 586

Creating a Form to Assign Permissions ... 592

Author List ... 592

Edit Author Permissions ... 593

Setting Permissions ... 597

Storing Permissions in the Database ... 598

A Crash Course in Binary ... 599

Be Bit-wise ... 601

Back to PHP ... 602

Storing Bitwise Permissions in the Database ... 606

Join Table and Bitwise Approaches: Pros and Cons ... 607

Cleaning Up ... 608

Editing Other People's Jokes ... 609

Phew! 612

Chapter 13: Content Formatting and Pagination....614

Regular Expressions....616

String Replacement with Regular Expressions....620

Emphasized Text....621

Paragraphs....628

Hyperlinks....632

Putting It All Together....634

Sorting, Limiting and Offsets....638

Sorting....639

Pagination with `LIMIT` and `OFFSET`....645

Pagination in Categories....652

Achievement Unlocked: Ninja....655

What Next?....655

Preface

It was 1998, I was twelve, and my parents had just bought the family our first modern PC. It wasn't long before I had figured out how to change the code for one of my favorite first-person shooter games—little things like making the rocket launcher fire a hundred rockets a second instead of one, then having it fire a hundred rockets in every direction … and promptly crashing the game. I was hooked, and I've been programming ever since.

The game was multiplayer. Other people had also discovered how to change the code, and the arms race quickly escalated. Someone would fire a hundred rockets at me. I'd have a script ready that would instantly build a wall right in front of me to block them all.

My opponent would spawn a dozen land mines underneath me. I'd turn off the gravity, then jump, soaring away from the impending explosion. Everyone could fly. It got to the point where it was no longer fun. You'd enter a game and someone had written a script to teleport you to the other side of the map, kill you instantly and force you to respawn, repeating the process a dozen times a second. They'd freeze your controls too, of course.

We discovered ways to block all this, but by the end it was a stalemate. Whoever managed to enter the game first could take complete control of it, and no matter how good your scripts were, there was nothing you could do. It was fun while it lasted.

That's how I learned the basics of coding, and that the only limit is your own imagination and creativity. During that time, I'd also taught myself HTML, and had my own website where I shared some of my game hacking techniques and scripts. No, the website isn't still up. Yes, it was terrible, full of bad grammar and cheesy animations (which was the style at the time, I promise!).

By 2000, I had taught myself the basics of PHP and MySQL and was running a website for a group of fellow gamers. I wrote some crude PHP scripts for

posting news on the website, as well as polls, and even a script for handling our mini-tournament rankings and fixtures.

After that, I moved on to writing desktop applications in a horrible language called Delphi, writing tools that aided people in modding various games. I graduated from University in 2007 with a degree in Software Engineering, worked for various companies as a PHP developer, then went back to academia to study for a PhD in Software Engineering. I currently teach at the University of Northampton in the UK.

I'm 34 now, and I've been programming for more of my life than not. It's fun, and it's something I thoroughly enjoy doing. I'm writing this book to share my knowledge with you and help you steer clear of some traps that are easy to fall into.

Learning to code is very enjoyable and rewarding. You can watch your program come alive as you build it. However, it can also be an incredibly frustrating experience. In this book, I'm going to try to use my own experience to give you a smoother ride than I and a lot of developers have had. I can steer you in the right direction from the start.

Before I introduce you to any code, I'm going to give you some general advice about programming and learning to code—advice which I give to all my students.

Who Should Read This Book

This book is aimed at intermediate and advanced web designers looking to make the leap into server-side programming. You'll be expected to be comfortable with simple HTML, as I'll make use of it without much in the way of explanation. No knowledge of Cascading Style Sheets (CSS) or JavaScript is assumed or required, but if you *do* know JavaScript, you'll find it will make learning PHP a breeze, since the basics of both languages are quite similar.

By the end of this book, you can expect to have a grasp of what's involved in building a modern PHP website, the basics of PHP, and tried and tested

techniques that are used by developers today. Most importantly, you'll come away with everything you need to build your own website!

Programming Has Changed

As a novice developer starting now, there's a lot more you need to know before you can publish a website than someone who was building a website in 2001.

When I started, it was a much simpler time. For example, website security wasn't much of a consideration. Unless you were a bank or a company taking credit card payments, there was very little chance anyone would target your site.

These days, however, every single website is constantly bombarded by bots and scripts specifically looking to exploit even the smallest doors you may have left open.

The way PHP scripts are written has changed dramatically as well—certainly for the better. It's now much, much easier to download and use someone else's code in your own project. The downside to this is that you need a much broader understanding of programming concepts before you can do anything useful.

To keep up with the competition, and with the needs of more demanding projects, PHP and MySQL have also had to evolve. PHP is now a far more intricate and powerful language than it was back in 2001, and MySQL is a vastly more complex and capable database. Learning PHP and MySQL today opens up a lot of doors that would have remained closed to the PHP and MySQL experts of 2001.

That's the good news. The bad news is that, in the same way that a butter knife is easier to figure out than a Swiss army knife (and less likely to cause self-injury!), all these dazzling new features and improvements have indisputably made PHP and MySQL more difficult for beginners to learn.

It Takes 10,000 Hours to Become an Expert

The science behind this statement is questionable, but the sentiment is correct. Programming is a skill, and it's incredibly difficult to master. Don't expect to become proficient overnight. By the end of this book, you'll have a good understanding of PHP, but there's always more to learn, regardless of the level you're at.

Having said that, in programming a little knowledge can go a long way. You'll be surprised how much you can do with just a few tools at your disposal!

You'll find that, after you've learned the very basics, you can achieve almost anything you want. There'll be very little you can't do, even though you only know a fraction of the programming concepts that are out there. The more advanced concepts are about making your code more efficient, quicker and easier to write, and much simpler to build on top of.

Resist the Temptation to Skip Ahead

This is one I reiterate time and time again for my students who miss lectures. Programming concepts build on top of each other. For the most part, you need to learn the earlier concepts before you can move on to the next one. If you try to move too fast, you'll get needlessly confused and make it more difficult for yourself.

There aren't many programming concepts that exist in isolation, so if you get stuck, it's often a result of not fully understanding an earlier concept. Don't be afraid to go back and give yourself a refresher on what you think you already know from before. It's usually quicker overall than struggling and trying to press forward when you get stuck!

The Concorde Fallacy

In the late 1970s, the British and French governments continued to fund the Concorde aircraft even though it was losing vast amounts of money. Their reasoning was that they had already spent so much on the project that, if they

scrapped it, they'd lose everything they had invested so far. Of course, they eventually lost considerably more because they kept throwing money at it. If they had stopped sooner, they would have saved a lot of money in the long run. This is often referred to as the "Concorde fallacy". There are times when it's better to cut your losses than keep working on a failed project!

There will come a time when you've spent hours on something and it's just not working. When this happens, take a step back and try to solve the problem in a different way. Use some of the alternative tools you have at your disposal. The solution might not be as elegant, but once you have it working you can tweak it.

Never be afraid to scrap everything and start again. When you're starting out, you'll end up writing a lot of code, trying to fit it into what you've done before, and gradually build a monster. And you won't *really* understand what the code is doing. It will become unworkable, and you'll get frustrated. Even making the slightest change will be hard work, as it will most likely break something else.

When this happens, don't be afraid to start again from scratch. I've lost count of the number of times I've started a project from scratch after getting it partially completed. You can usually get to the same point you got stuck at within a couple of hours, and you have far neater code and a better understanding of it as a result!

However, I strongly recommend keeping that code as a point of reference, rather than deleting it.

Everyone starts off writing terrible code. Ask any programmer to look at some code they wrote when they first started and they'll cringe, even if they only started a few months ago.

You're Not Learning PHP

Yes, you read that right. This book is focused entirely on PHP and MySQL, but don't fall into the trap of thinking you're *learning PHP*. Well, you are learning PHP, but I'm using PHP to teach you how *to code*.

When you learn to drive, you don't learn to drive a Ford. You learn the concepts of driving, and you can apply them to any car you get in, even if a few of the controls are in a different place.

Concepts you'll learn here will apply to almost any other language you wish to learn in the future. Sure, there are some differences, but the underlying concepts are the same.

Once you can program proficiently in one language, you can get to a reasonable standard in another within a few days! So don't read this book thinking "I'm learning PHP", but instead think "I'm learning to code".

It's more important to remember the *concepts* than the *syntax*. You can always look up the correct syntax, but understanding the underlying concepts is more difficult. Which brings me to my next point ...

Getting Braces and Semicolons in the Right Place Is the Easy Part

When you start out, you'll constantly put brackets, braces, semicolons, dots and pretty much everything else in the wrong place. You'll forget to put in a single character and your whole program won't work.

This can be incredibly frustrating at first! But once you get the hang of it, you soon realize that getting the syntax right is the easy part. It's easy because it's strict. It's either right or it's wrong. It works or it doesn't.

The hard part is actually writing the *logic*—breaking a problem down to its smallest parts so you can explain it to the computer. The computer will quickly tell you if the syntax is wrong, but there's no way for it to tell you whether you've given it the right instructions to solve the problem at hand.

You Won't Get Anything Done by Planning

> *You won't get anything done by planning. — Karl Pilkington*

If you've done any reading about programming, you've probably heard that you need to spend lots of time designing your code—that you should carefully plan the logic of your program and how it will work before writing a single line of code. You'll come across books and articles that teach development methodologies, something called "requirements engineering", diagrams for visually representing code, and all sorts of tips on how to plan your code out before you write it. For example, there's this advice in Sam's *Teach Yourself Beginning Programming in 24 Hours*:

> *There are three fundamental steps you should perform when you have a program to write:*
>
> 1. Define the output and data flows.
> 2. Develop the logic to get to that output.
> 3. Write the program.
>
> *Notice that writing the program is the last step in writing the program. This is not as silly as it sounds. Remember that physically building the house is the last stage of building the house; proper planning is critical before any actual building can start. You will find that actually writing and typing in the lines of the program is one of the easiest parts of the programming process. If your design is well thought out, the program practically writes itself; typing it in becomes almost an afterthought to the whole process.*

I'm now going to say something that will make most programmers wince:

ignore that advice entirely and get stuck into writing code.

When I say this in lectures, my students breathe a sigh of relief. They're there to learn to code, and the best way to learn to code is to start writing.

The fundamental problem with this advice is that it forgets a somewhat obvious fact: to design software, you need to know what tools are available and the problems they solve. Otherwise, any design you come up with will be meaningless if you don't know what tools are available.

Using Sam's analogy, let's assume you know nothing about building a house. You don't know how to use a hammer, a saw, how strong a beam needs to be to support your roof, how deep your foundations need to be, how to plumb in the bathroom, what materials are suitable for which part of the house, and so on.

You can spend as long on the design as you like and plan things as carefully as possible, but unless you know what your tools are capable of and their limitations, you'll end up with a design that doesn't fully utilize the tools, or a design that just isn't possible with the tools/materials available to you. Without knowing that you need a six-meter foundation for a three-story house, you can't design a three-story house.

Equally, you can't design a computer program if you don't know how to program!

To demonstrate my point, here's a story from a TED talk called "Want to help someone? Shut up and listen", by Ernesto Strolli[1]:

> *It was a project where we Italians decided to teach Zambian people how to grow food. So we arrived there with Italian*

---

[1] https://www.ted.com/talks/ernesto_sirolli_want_to_help_someone_shut_up_and_listen

> *seeds in southern Zambia in this absolutely magnificent valley going down to the Zambezi River. And we were amazed that the local people in such a fertile valley would not have any agriculture. But instead of asking them how come they were not growing anything, we simply said, "Thank God we're here. Just in the nick of time to save the Zambian people from starvation."*
> *And of course, everything in Africa grew beautifully and we had these magnificent tomatoes. In Zambia, the tomatoes grew even larger than they did in Italy. And we were telling the Zambians, look how easy agriculture is. When the tomatoes were nice and ripe and red, overnight, some 200 hippos came up from the river and they ate everything. And we said to the Zambians, "My God, the hippos." And the Zambians said, "Yes, that's why we have no agriculture here."*

Ernesto's team knew exactly what they were doing. They carefully planned everything out and managed to get the result they wanted. However, all that planning and designing was wasted because of something they didn't see coming.

Programmers don't encounter hippos, but there are lots of obstacles you won't be able to anticipate, and you'll inevitably run into them. Any time you spend designing is wasted when the equivalent of 200 hippos come and eat your code. You have to scrap the design and start again.

During this book, I'll warn you about the various hippos you might encounter, but it's a good idea to test it for yourself. Learn by doing. Rush in. Write some code. It almost certainly won't work the first time, but you'll have learned something in the process. Try again with a different approach and you'll come up with something that does work.

There's no way to design a program until you're aware of the problems you're

likely to encounter and the limitations of the tools available to you.

Okay, Design Isn't All Bad

To prevent a wave of hate mail from other programmers, I'm going to conclude this section by saying that, for *professional programmers*, spending time up front designing the code before building it is *vital*. However, professionals are writing code they may need to work with for years or decades to come. The code they write needs to be written in such a way that it's extensible and easy for others to follow.

During this book, I'll get you to think about the structure of your code and how to write code that's reusable and extensible. But you're not here to write code that will be used in real projects and will need to be maintained for years to come. You're here to learn. Go and find all those hippos. You'll learn more from making mistakes than you will from code that works right away.

The time you spend planning your code should be proportional to your programming ability. If you're just starting out, as long as you have a broad understanding of what you want the program to do, jump in and start writing code until it does what you want. You can get stuck and try a different approach without feeling like you're *doing it wrong* because it's going against that design you spent hours working on. What I said above about the Concorde fallacy applies here as well.

For the first few chapters, at least, just dive in. Run your code, see if it works. Try solving some of the problems I set before I give you the solutions. You'll learn more by discovering the solutions yourself than blindly typing in the code I give you.

As your knowledge grows, you'll have a firmer understanding of what tools are available and the way problems need to be broken up. Once you reach that level, you can start planning things out in more detail before writing your code.

Where to Find Help

PHP and MySQL are moving targets, so chances are good that, by the time you read this, some minor detail or other of these technologies has changed from what's described in this book. Thankfully, SitePoint has a thriving community of PHP developers ready and waiting to help you out if you run into trouble, and we also maintain a list of known errata for this book you can consult for the latest updates.

The SitePoint Forums

The SitePoint Forums are discussion forums where you can ask questions about anything related to web development. You may, of course, answer questions, too. That's how a discussion forum site works—some people ask, some people answer, and most people do a bit of both. Sharing your knowledge benefits others and strengthens the community. A lot of fun and experienced web designers and developers hang out there. It's a good way to learn new stuff, have questions answered in a hurry, and just have fun.

The SitePoint Forums include separate forums for PHP and MySQL:

- https://www.sitepoint.com/community/c/php/31
- https://www.sitepoint.com/community/c/databases/38

The Code Archive

As you progress through this book, you'll note a number of references to the **code archive**. This is a GitHub repository that contains each and every line of example source code that's printed in this book. If you want to cheat (or save yourself from carpal tunnel syndrome), go ahead and download the archive[2]. Select the example from the dropdown that says **Branch**, then choose **Clone or Download**, and you can download a `.zip` file for that example.

Alternatively, if you're familiar with Git, you can clone the repository.

---

2. https://github.com/spbooks/phpmysql7

Your Feedback

If you're unable to find an answer through the forums, you can also report and discuss issues in the book's GitHub repository. To report problems issues with teh book, you can write to us at `books@sitepoint.com`. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

Let's Go

Now that I've introduced myself, given you some broad tips, and shown you where to find help, it's time to get started! You'll begin by setting up a development environment, and you'll be writing your first lines of code soon.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
   ⋮
new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("https://www.sitepoint.com/responsive-web-
➥design-real-user-testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1: Installation

In this book, I'll guide you as you take your first steps beyond the static world of building web pages with the purely client-side technologies of HTML, CSS, and JavaScript. Together, we'll explore the world of building websites, and discover the dizzying array of dynamic tools, concepts, and possibilities they open up. Whatever you do, don't look down!

Okay, maybe you *should* look down. After all, that's where the rest of this book is. But remember, you were warned!

Before you build your first dynamic website, you must gather together the tools you'll need for the job. Like baking a cake, you'll need the ingredients before you can start following the recipe. In this chapter, I'll show you how to download and set up the software packages required.

If you're used to building websites with HTML, CSS, and perhaps even a smattering of JavaScript, you're probably familiar with uploading the files that make up your site to a certain location. It might be a web hosting service you've paid for, a web space provided by your internet service provider, or maybe a web server set up by the IT department of the company you work for. In any case, once you copy your files to any of these destinations, a software program called a web server is able to find and serve up copies of those files whenever they're requested by a web browser like Microsoft Edge, Internet Explorer, Google Chrome, Safari, or Firefox. Common web server software programs you may have heard of include Apache HTTP Server (Apache), NGINX, and Internet Information Services.

PHP is a server-side scripting language. It's completely free to download and use. You can think of it as a plugin for your web server that enables it to do more than just send exact copies of the files requested by web browsers. With PHP installed, your web server will be able to run little programs—called PHP scripts—that can do tasks like retrieve up-to-the-minute information from a database and use it to generate a web page on the fly, before sending it to the browser that requested it. Much of this book will focus on writing PHP scripts to do exactly that.

For your PHP scripts to retrieve information from a database, you must first

have a database. That's where MySQL comes in. MySQL is a **relational database management system**, or RDBMS. We'll discuss the exact role it plays and how it works later, but briefly, it's a software program that's able to organize and manage many pieces of information efficiently while keeping track of how all those pieces of information are related to each other. MySQL also makes that information really easy to access with server-side scripting languages such as PHP. And, like PHP, it's completely free for most uses.

The goal of this first chapter is to set you up with a web server equipped with PHP and MySQL. I'll provide step-by-step instructions that work on recent versions of Windows, macOS, and Linux, so no matter what flavor of computer you're using, the instructions you need should be right here.

Your Own Web Server

Chances are, your current web host's web server already has PHP and MySQL installed—which is one of the reasons PHP and MySQL are so popular. If your web host is so equipped, the good news is that you'll be able to publish your first website without having to shop for a web host that supports the right technologies.

When developing static websites, you can simply load your HTML files directly from your hard disk into your browser to see how they look. There's no web server software involved when you do this, which is fine, because web browsers can read and understand HTML code all by themselves.

However, when it comes to dynamic websites built using PHP and MySQL, your web browser needs some help. Web browsers are unable to understand PHP scripts. Instead, PHP scripts contain instructions for a PHP-savvy web server to execute in order to *generate* the HTML code that browsers can understand.

Even if you have an existing web host that supports PHP, you're still going to want to be able to run PHP scripts yourself without needing to use someone else's server. For this, you'll need to set up your own web server. The word "server" might make you think of a large, air-conditioned room filled with big

computers in racks. But don't worry, you don't need any fancy new hardware. Your laptop or desktop computer will work just fine.

To run PHP scripts on your web host, you need to write them in your editor, open your FTP or SSH client and upload them to the server. Only then can you see the result in your browser by navigating to the URI of the file you created. If you made a mistake and there's an error, you'll need to change the code, go back into your FTP program, upload the file again and then reload the page. This is tedious, and uses up precious time that you could be using to write code. By running a server on your own PC, you'll be able to save a file in your editor and view the changes in your browser by simply refreshing the page—no file uploading required. This is a real time saver, and one of the biggest (although not only!) advantages of running a server on your PC—even if you have a perfectly good web host already.

So how do you get a web server running on your PC? There are four ways to achieve this, each with its own advantages and disadvantages.

Server Setup 1: Manually Installing All the Software Components

Apache is a web server, and like most software it comes with an installer that lets you easily set it up on your PC. Without much effort, you can have it *serve* web pages. However, there are hundreds of configuration options, and unless you know what you're doing, it can be time consuming and confusing to get it working for developing PHP websites.

For our purposes of running PHP scripts, a web server alone is not enough. For manual installation, you'll also need to install PHP—which doesn't have an installer—and configure it. As with Apache, there are lots of options, and the defaults are set up as if you're running a live website. For developing code, this is bad, as there are no errors shown. If you make a mistake, you'll get a blank page with no indication of what went wrong. Even a single character out of place—such as a missing brace or semicolon—will give you a blank page, with no indication of what caused the problem. To solve this, you'll need to manually configure the PHP installation and tweak the settings to show error

messages and enable other tools that make development a more pleasant task.

You'll also need to configure Apache to talk with PHP, so that when someone connects to the server and requests a file with a `.php` extension, the file is first sent to PHP for processing.

To follow along with this book, you'll also want MySQL, which means manually installing and configuring that as well.

Apache, MySQL and PHP each have dozens of configuration options, and unless you know exactly what you're doing, they can be difficult to set up. Even if you're an expert, it will take at least an hour to get everything working!

Manual installation requires a significant amount of knowledge or research and is beyond the scope of this book. Being able to configure a server is a useful skill, to be sure, but it doesn't help you learn how to program using PHP—which is what you're really interested in if you're reading this book.

This option is not for the faint hearted, and even for seasoned professionals it's very easy to miss some important settings. Luckily for us, we don't need to worry about setting up and configuring all the software individually.

Server Setup 2: Pre-packaged Installations

The problems with manual installations have been recognized by groups of developers over the years, and to overcome them they've built pre-packaged installations—a single installer that installs PHP, Apache, MySQL and other relevant software, all pre-configured with appropriate settings for developers like you. The most popular example of this kind of package is XAMPP: X (any operating system), Apache, MySQL (or more specifically, MariaDB, a "fork" of MySQL with a better license), PHP, and Perl. Alternatives include WAMP (Windows, Apache, MySQL, and PHP), LAMP (Linux, Apache, MySQL, and PHP), and MAMP (macOS, Apache, MySQL, and PHP).

This is obviously a lot simpler than manually installing each piece of software,

and doesn't require you to learn how to configure your server. It's quick and easy and a lot better than a manual installation, though there are still a couple of problems you may encounter with this method, listed below.

- Your web host is probably running Linux, but your PC probably isn't. Although Apache, MySQL and PHP work in Windows, Linux or macOS, there are some big differences between the way the operating systems work. On Windows, file names are not *case-sensitive*, meaning that `FILE.PHP` is the same as `file.php` and `fILE.pHp`. On your web host, this will almost certainly not be the case! This causes frequent problems when a script working perfectly on your Windows development server doesn't work once it's uploaded, because files are being referenced in the code with the wrong case.

- Apache and MySQL are *servers*, and they run in the background. Even when you're not developing software, they'll be running, using up your computer's RAM and processing power.

- Pre-packaged software is always slightly out of date. Although security fixes aren't a priority for a development computer (you shouldn't be allowing people to access it across the Web!), it's always useful for developers to stay on the most recent versions of software to check for problems that might be encountered when the software on your web host is updated. If your web host is using a newer version of PHP than your development server, this can cause problems with features that have been changed or removed. Finally, developers like to play with new features as they're released. You won't be able to do this if you're not using the latest versions!

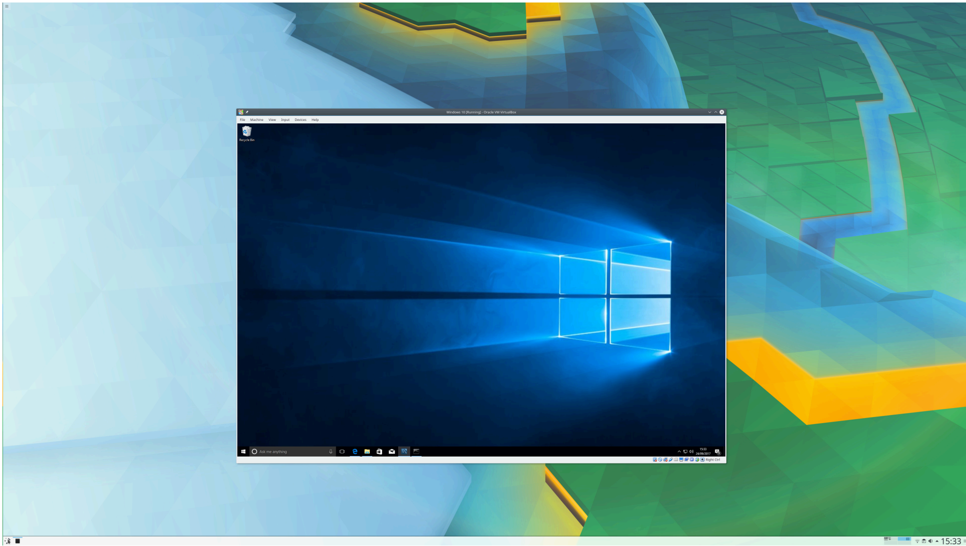
Although pre-packaged installations are much better than installations, these problems don't make them ideal. Luckily, there's an even better approach!

Server Setup 3: Virtual Servers

The third method of getting a server up and running is a "virtual server". A **virtual server** acts like a web server on a different computer. This computer

can be running any operating system, and you can connect to it from your PC as if it were somewhere else in the world.

Virtualization software such as VirtualBox and the tools offered by VMware is common. As a web developer, you may be familiar with tools such as modern.ie[3], a helpful service provided by Microsoft that lets you download **virtual machines** running various versions of Windows, Microsoft Edge and Internet Explorer. If you want to see what your website looks like in Internet Explorer 8 on Windows XP, you can download the relevant virtual machine and run it in a Window on your Windows 10/macOS/Linux desktop without having to actually install and run Windows 7 with Internet Explorer 8 inside your existing Windows 10, Linux or macOS installation.



1-1. Windows 10 running inside Arch Linux

Software like VirtualBox allows you to run an operating system inside another operating system. For testing Internet Explorer 8, you can run Windows 7 in a virtual machine. However, for our purposes of running PHP scripts, this allows us to do something a lot cooler: we can run a Linux web server with PHP, Apache and MySQL installed on our Windows or macOS PC.

---

3. http://modern.ie/

This can be used to allow you to run the exact same versions of PHP, MySQL and Apache that are being used on your web host, on the exact same operating system, which prevents any issues that may exist due to version differences or differences in the operating systems being used.

One of the biggest advantages is that you can download pre-configured virtual machines, like the Windows XP and Internet Explorer 8 virtual machine provided by Microsoft, or a virtual machine that has PHP, Apache and MySQL installed and configured to work together. This is like the pre-configured package but runs on Linux as if it's a real web server on your network.

The downside to all this is that you have to download an entire operating system in order to run your code. That means more downloading. It also means that packages are locked to whatever is provided by the virtual machine you download. Swapping out PHP 7 for PHP 8 requires downloading a whole new copy of the operating system.

Server Setup 4: Docker

Docker takes the idea of virtualization and flips it on its head. Each program (or "service", in Docker's terminology) runs in its isolated environment called a **container**.

Docker allows a software developer to provide a configuration file that describes all the programs needed to run an application—for example, PHP, MySQL, Apache, and so on.

This configuration file is then treated as its own application. When you run the application, Docker downloads and sets up all the software listed in the configuration file automatically.

The overhead of this is much lower than you might think, and there are numerous benefits:

- The server configuration is provided as part of the application. When you make your website live, you can upload all of the configuration files along

with the website. In the traditional model, you'd have to manually set up the web server and configure it.

- Multiple websites can run at the same time on your development machine, with different configurations and even different server software (for example, one website using Apache, another using NGINX). Without Docker, there's generally one PHP version installed with a single configuration applied to every website running on the machine.

- You can easily swap out a piece of software. If you want to change the website from PHP 7 to PHP 8, it's one change in one file, and you can do it on a per-website basis rather than forcing an upgrade to all your websites at once.

Docker is currently the best option for setting up a PHP development environment. If you want to know more about setting up a development environment for yourself using Docker, check out my article "Setting Up a Modern PHP Development Environment with Docker"[4].

Getting Started

Before writing any PHP code and developing your website, we'll set up the development environment using Docker. I've provided all the configuration for you, but before you start, you'll need to install Docker.

Installation on Windows

Firstly, download and install the latest version of Docker for Windows from the Docker site[5].

Once you've installed Docker (and rebooted if asked), create a folder on your computer where you want to store your website. This can be anywhere: your `Documents` folder, the desktop, an external hard drive, and so on, but make

---

4. https://www.sitepoint.com/docker-php-development-environment/
5. https://hub.docker.com/editions/community/docker-ce-desktop-windows/

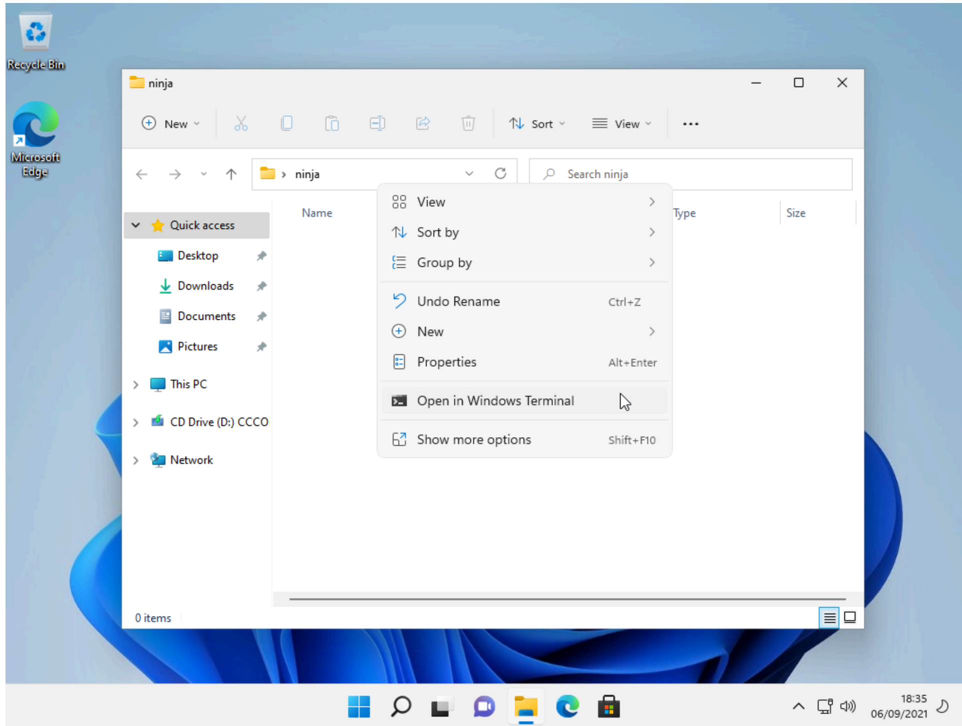
sure you know where this is as you'll need to go back to this folder frequently.

You'll also need to know how to open a terminal. Windows 10 makes this easy. With the folder open in the file explorer, choose the **File** menu at the top left of the window, then click **Open Windows PowerShell**. Make sure you do this from the **File** menu and not the start menu, as clicking it from **File** will open the command prompt in your chosen folder.

With PowerShell open, verify that the path it's showing is your chosen folder—for example, `C:\Users\Tom\Documents\My Website`. Then go to the "Getting Started with Docker" section below.

At the time of writing, in the current preview build of Windows 11, PowerShell has been renamed to *Windows Terminal* and is accessed in a different way:

1. Enable the option by manually opening the *Windows Terminal* application from the start menu once. You can just open and close the program. Inexplicably, Microsoft has designed Windows 11 in such a way that opening one program turns on a hidden setting in another, and if you skip this step the menu item from step three is missing.
2. Use File Explorer to navigate to your folder.
3. Right click in the main panel (it should say "This Folder is Empty") and select **Open in Windows Terminal**.



1-2. The Open Terminal option in Windows 11

Windows 11 Caveat

These instructions may have changed by the time Windows 11 is released and after this book has been published.

Installation on macOS

Firstly, download and install Docker for macOS from the Docker site[6].

Once you've installed Docker (and rebooted if asked), create a directory on your computer where you want to store your website. This can be anywhere: your `Documents` directory, the desktop, an external hard drive, and so on, but make sure you know where this is, as you'll need to go back to this folder

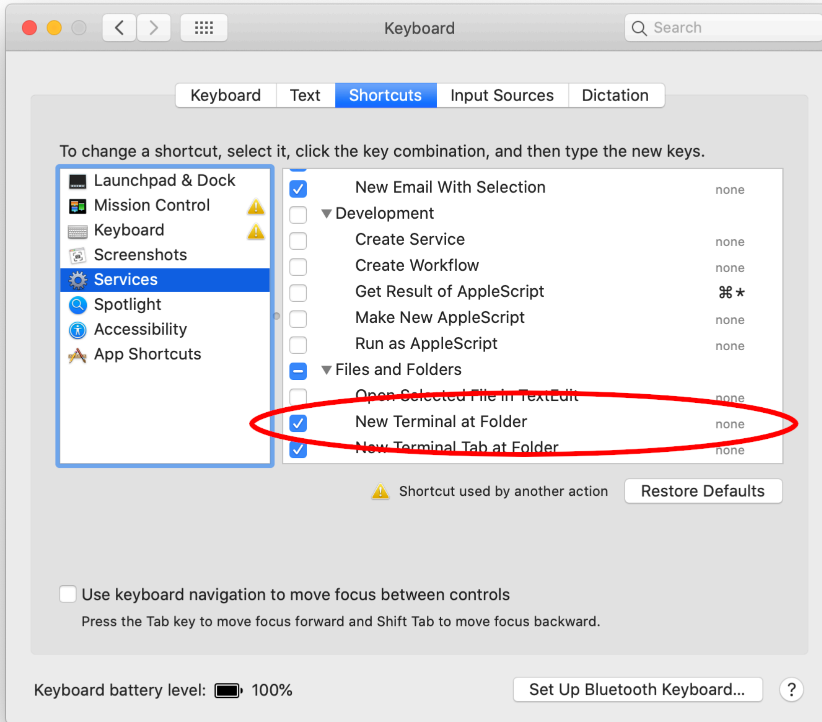
---

6. https://hub.docker.com/editions/community/docker-ce-desktop-mac/

frequently.

If you’re familiar with navigating around your computer using the terminal you can skip this step, but for convenience I recommend enabling a feature that lets you open a terminal in the current directory. This can be turned on using the following steps:

1. open System Preferences
2. go to **Keyboard**
3. select the **Shortcuts** tab
4. click the **Services** button
5. check the box **New Terminal at Folder**



1-3. macOS: Enable New Terminal at Folder

With this enabled:

1. open the Docker app from the `Applications` folder and accept the terms
2. go back into Finder
3. navigate to the directory you want to store your website in
4. right click (or `control` click) on the directory name, choose the **Services** option from the content menu, and select **New Terminal At Folder**

With this terminal open, you're ready to proceed to the "Getting Started with Docker" section below.

Going forward, you'll only need to repeat the last set of steps each time you wish to start or stop your server.

Installation on Linux

Linux generally makes installing software very simple, and Docker is actually a Linux program by design. On most distributions, it can be installed via your package manager, but you may need to set up a custom package repository.

For Debian-based distributions (Debian, Ubuntu, Mint and KDE Neon) and RedHat-based distributions (RedHat, CentOS, Fedora) there's an install script which sets up a custom repository and installs Docker:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

For specific instructions for your distribution, take a look at the Docker manual[7].

Once Docker is installed, you have to enable and start the service using these commands:

```
sudo systemctl enable docker
sudo systemctl start docker
```

Finally, you can add yourself to the `docker` group so that you won't need to run Docker applications through `sudo`:

```
sudo usermod -aG docker ${USER}
```

Changing user groups requires logging out and back in, so log out and back in

7. https://docs.docker.com/engine/install/

(remember to save anything you have open!), and once you're back in you're ready to open a terminal.

After you've logged back in, create a directory on your computer where you want to store your website. This can be anywhere: your `Home` directory, the desktop, an external hard drive, and so on, but make sure you know where this is as you'll need to go back to this folder frequently.

Open the folder in your file manager and choose the **Open Terminal** option. In the Dolphin file manager, you can right click and choose **Open Terminal**, but this option may be in a different location and possibly named differently depending on which file manager you're using.

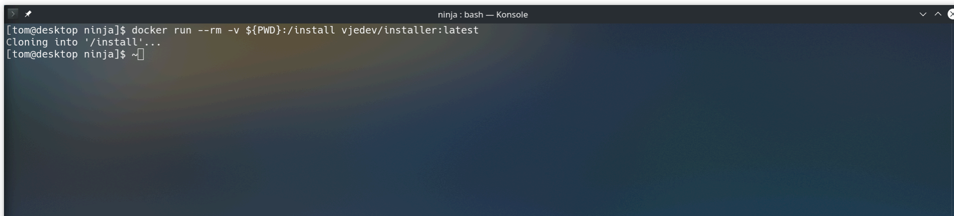
Getting Started with Docker

Now that you have all the software installed and your terminal is open at the correct location, it's time to run some Docker commands.

While you could manually set up the environment and configure everything yourself, this requires a different skill set from programming and is beyond the scope of this book. I've provided some configuration based on the article I linked to earlier[8].

Firstly, copy the configuration using this command:

```
docker run --rm -v ${PWD}:/install vjedev/installer:latest
```



1-4. Running the installer

8. https://www.sitepoint.com/docker-php-development-environment/

This command downloads some configuration files from a GitHub repository and copies them to the current folder. The directory has to be empty for this to work, so make sure there are no files or subdirectories in the directory you're trying to run this command from.

GitHub and Repositories

GitHub is a source code hosting platform used by a lot of open-source projects. A **repository** is the name given to an individual project. In this case, the code is being downloaded from https://github.com/v-je/docker.

Using Git

If you have Git installed, you could just run the following command at this point:

```
git clone https://github.com/v-je/docker .
```

However, rather than asking every reader of this book to install Git for a single command, I created an installer script which does this for you.

You only need to do this the first time, and once the command completes you'll see some configuration files appear in your directory. Feel free to look though the files created. There's `nginx.conf`, which configures the web server; `PHP.Dockerfile`, which configures PHP extensions; and `docker-compose.yml`, which lists all the programs that will be installed and run when you start the server (NGINX, MySQL, PHP, and so on).

Don't worry about trying to understand these files. You don't need to do anything with them, as they're just there to configure the server when you run the next command.

Apache and NGINX

Before we continue, it's worth mentioning that the environment we've just set up uses NGINX and not Apache. If you've been using XAMPP or a similar package, the web server you're using is Apache. The web server is the part of the server that listens to requests from a web browser and sends it files.

Apache is fine, and it works, but it's been around forever. When Apache was created, the Web was a very different place. Apache is big, and there are lots of different features that have come and gone—but which Apache still supports. The Web has changed a lot since Apache was created, and although Apache is a capable server and will work fine, most websites these days tend to use NGINX. It's easier to configure, more lightweight, and better tuned for the kinds of tasks a lot of of modern websites make use of (such as video streaming); and as such, its market share is growing rapidly at the expense of Apache's[9].

My general advice is: if you already have a website running Apache, there's no reason to go through the hassle of changing it, but if you're starting a new project from scratch, use NGINX.

Now that you have the server configuration files downloaded, it's time to get Docker to download and run all the programs required for the server. Run the following command:

```
docker compose up
```

9. https://news.netcraft.com/archives/category/web-server-survey/

```
ninja : docker-compose — Konsole
mysql_1              |
mysql_1              | The latest information about MariaDB is available at http://mariadb.org/.
mysql_1              | You can find additional information about the MySQL part at:
mysql_1              | http://dev.mysql.com
mysql_1              | Consider joining MariaDB's strong and vibrant community:
mysql_1              | https://mariadb.org/get-involved/
mysql_1              |
mysql_1              | 2021-09-06 15:21:31+00:00 [Note] [Entrypoint]: Database files initialized
mysql_1              | 2021-09-06 15:21:31+00:00 [Note] [Entrypoint]: Starting temporary server
mysql_1              | 2021-09-06 15:21:31+00:00 [Note] [Entrypoint]: Waiting for server startup
mysql_1              | 2021-09-06 15:21:31 0 [Note] mysqld (mysqld 10.4.12-MariaDB-1:10.4.12+maria~bionic) starting as process 123 ...
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Using Linux native AIO
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Uses event mutexes
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Compressed tables use zlib 1.2.11
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Number of pools: 1
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Using SSE2 crc32 instructions
mysql_1              | 2021-09-06 15:21:31 0 [Note] mysqld: O_TMPFILE is not supported on /tmp (disabling future attempts)
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Initializing buffer pool, total size = 256M, instances = 1, chunk size = 128M
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Completed initialization of buffer pool
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: If the mysqld execution user is authorized, page cleaner thread priority can be changed. See t
he man page of setpriority().
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: 128 out of 128 rollback segments are active.
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Creating shared tablespace for temporary tables
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Setting file './ibtmp1' size to 12 MB. Physically writing the file full; Please wait ...
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: File './ibtmp1' size is now 12 MB.
mysql_1              | 2021-09-06 15:21:31 0 [Note] InnoDB: Waiting for purge to start
mysql_1              | 2021-09-06 15:21:32 0 [Note] InnoDB: 10.4.12 started; log sequence number 60972; transaction id 21
mysql_1              | 2021-09-06 15:21:32 0 [Note] InnoDB: Loading buffer pool(s) from /var/lib/mysql/ib_buffer_pool
mysql_1              | 2021-09-06 15:21:32 0 [Note] Plugin 'FEEDBACK' is disabled.
mysql_1              | 2021-09-06 15:21:32 0 [Note] InnoDB: Buffer pool(s) load completed at 210906 15:21:32
mysql_1              | 2021-09-06 15:21:32 0 [Warning] 'user' entry 'root@6b414938368c' ignored in --skip-name-resolve mode.
mysql_1              | 2021-09-06 15:21:32 0 [Warning] 'user' entry '@6b414938368c' ignored in --skip-name-resolve mode.
mysql_1              | 2021-09-06 15:21:32 0 [Warning] 'proxies_priv' entry '@% root@6b414938368c' ignored in --skip-name-resolve mode.
mysql_1              | 2021-09-06 15:21:32 0 [Note] Reading of all Master_info entries succeeded
mysql_1              | 2021-09-06 15:21:32 0 [Note] Added new Master_info '' to hash table
mysql_1              | 2021-09-06 15:21:32 0 [Note] mysqld: ready for connections.
mysql_1              | Version: '10.4.12-MariaDB-1:10.4.12+maria~bionic'  socket: '/var/run/mysqld/mysqld.sock'  port: 0  mariadb.org binary distribution
mysql_1              | 2021-09-06 15:21:32+00:00 [Note] [Entrypoint]: Temporary server started.
mysql_1              | Warning: Unable to load '/usr/share/zoneinfo/leap-seconds.list' as time zone. Skipping it.
```

1-5. The output of docker compose up

Compose Yourself

For boring technical reasons that aren't worth getting into here, the `docker compose` command used to be a separate program called `docker-compose`. This functionality has now been moved into Docker itself. However, if you're on an older Linux distribution, or running an older version of Docker Desktop, you may need to replace `docker compose` with `docker-compose` in all the `docker compose` commands. Only do this replacement for `compose`. Other commands like `docker run` should retain the space.

You may get the following error when running `docker compose up`:

```
docker: 'compose' is not a docker command.
```

If so, you'll need to replace the command with `docker-compose up`, and if you're on Linux, you'll probably also need to install the separate `docker-compose` package using your package manager.

The `docker compose` command loads `docker-compose.yml`, then downloads

and installs all the software listed as *services*. The main things we're interested in for now are PHP, NGINX, and MariaDB.

The first time you run this command it will take a couple of minutes, as it has to download all the required software. Don't worry, as this only happens the first time. Once the software is downloaded and installed, starting the server in future will be significantly easier.

Unlike using a manual NGINX/PHP/MySQL installation directly on your PC, the server is only started when you want it to be, by running `docker compose up`. You can stop the server at any time by going back into the terminal and pressing `Ctrl` + `C`.

Alternatively, you can start the server in the background using this command:

```
docker compose up -d
```

You can stop the server by running `docker compose down` from your terminal. (You must open the terminal in the correct directory!)

Each time you want to work on the website, you can start and stop the server using the same process:

1. navigate to the folder you created
2. open your terminal/Windows PowerShell as you did earlier
3. run `docker compose up -d`

When you're done, you can stop the server using the same process:

1. navigate to the folder you created
2. open your terminal/Windows PowerShell as you did earlier
3. run `docker compose down`

If you've left the terminal open in the background, you can just open the window and run step 3.

Ups and Downs

The command `docker compose down` stops the running services and removes them, freeing up disk space. It also forces them to be recreated each time.

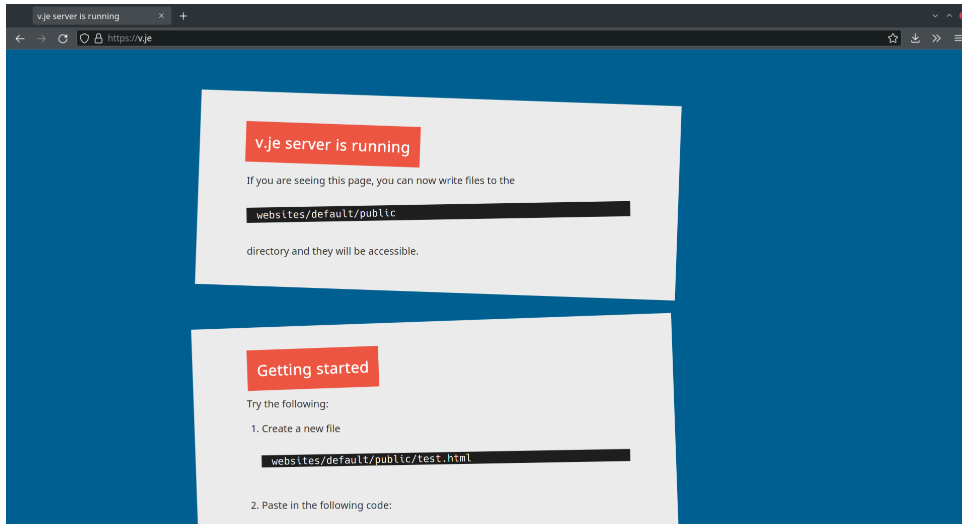
You can also use `docker compose stop` and `docker compose start` to stop and start the server. But these rely on the container, network and other things Docker creates behind the scenes being kept between executions. System restore, moving files between computers, or even running other Docker commands you're using for other tasks (be careful with `docker prune`!) can remove these.

Using `up` and `down` will use a blank slate approach and be consistently reliable. However, it does require removing when it's not in use and re-creating it when you use it. Don't worry, though: all your PHP, JavaScript and CSS files in the directory you created won't be removed!

Connecting to the Server and Creating Your First File

Now that you have the server running, you can connect to it using your favorite web browser. The server acts like a real web server running somewhere on the Internet. You can connect to it by opening your browser, typing `v.je` into the address bar and pressing `enter`.

You should see a test page like the one pictured below.



1-6. A screenshot of the server test page

With the server running, it's time to create your first file. When you ran the `docker-compose up` command, a folder called `websites` was created automatically in the directory from which the command was run. The files in the `websites/default/public` folder are the files used for the test page in the screenshot above.

Using your favorite text editor, create a file called `test.html` in the `websites/default/public` directory that contains the following code:

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

You can now view your web page on the server. By default (we'll see later on that this can be overridden) a URL directly maps to a file. A file called `test.html` placed in the `public` directory can be accessed on `https://v.je/test.html`, `public/products.html` can be viewed on `https://v.je/products.html`, and files can even be put in subdirectories. For example,

`public/images/logo.png` can be viewed at the URL `https://v.je/images/logo.png`.

Folder Levels

You might be wondering why there are three levels of folder—`websites`, `default` and `public`—instead of just having the `public` directory with the files in it.

The environment is configured to host multiple websites from different folders within the created `websites` directory. To create a new website that's available at `https://mysite.v.je/`, take the following steps:

1. Create the directory `mysite` inside the `websites` directory.
2. Create the directory `public` inside the `mysite` directory.
3. Place your web-accessible files inside the `websites/mysite/public` directory. For example, the file in `websites/mysite/public/phpinfo.php` will be accessible at the URL `https://mysite.v.je/phpinfo.php`.

Any directory you create inside the `websites` directory is treated as a subdomain of `v.je`. If no subdomain is specified and you just visit `v.je`, it will load the website from the `default` directory.

As we'll see later on in the book, it's good practice to keep certain files outside the `public` directory for security, which is why each website has a `public` directory.

Text Editors

Text editors provided by your operating system—such as Notepad or TextEdit—aren't really suitable for editing HTML and PHP scripts. However, there are several solid text editors with rich support for editing PHP scripts that you can download for free. Here are a few that work on Windows, macOS, and Linux:

- Visual Studio Code[10] (Getting Started guide[11])
- Atom[12] (Getting Started guide[13])
- Sublime Text[14] (Documentation[15])

These are all very similar, and for the purposes of this book, any one of them is a good choice and will make your life as a developer a lot simpler than Notepad or TextEdit.

We're All Set Up

In this chapter, you've learned how to set up a web server with Docker, and how to host an HTML file on the server. I've only covered the basics in order to quickly get to the meat and bones of this book: actually programming in PHP. However, having a good development workflow as a PHP developer is a skill in its own right. For our purposes, however, the server is up and running, and you're ready to write your first PHP script.

---

10. https://code.visualstudio.com/
11. https://visualstudio.microsoft.com/vs/getting-started/
12. http://atom.io/
13. https://flight-manual.atom.io/getting-started/
14. http://www.sublimetext.com/
15. http://www.sublimetext.com/docs/

Chapter

Introducing PHP

2

Now that you have your server up and running, it's time to write your first PHP script. PHP is a server-side language. This concept may be a little difficult to grasp, especially if you've only ever designed websites using client-side languages like HTML, CSS, and JavaScript.

A server-side language is similar to JavaScript in that it allows you to embed dynamically generated content into the HTML code of a web page, giving you greater control over what appears in the browser window than HTML alone can provide. The key difference between JavaScript and PHP is the stage at which the code is run.

Client-side languages like JavaScript are read and executed by the web browser after downloading the web page (embedded programs and all) from the web server. In contrast, server-side languages like PHP are run by the web *server*, before the web page is sent to the browser. Whereas client-side languages give you control over how a page behaves once it's displayed by the browser, server-side languages let you generate customized pages on the fly before they're even sent to the browser.

Once the web server has executed the PHP code embedded in a web page, the result takes the place of the PHP code in the page. All the browser sees is standard HTML code when it receives the page. That's why PHP is called a "server-side language": its work is done on the server.

JavaScript on the Server

In the early days of the Web—and when the first edition of this book was published!—JavaScript was a client-side scripting language used mainly in the browser. Then along came technologies like Ajax, which allowed JavaScript to communicate with the server. And more recently still, JavaScript has been used both in the browser and on the server to create database-driven apps. While these new uses for JavaScript offer exciting possibilities, there's still very much a place for PHP—as this book sets out to demonstrate!

PHP code is written in PHP tags. Like most HTML tags, PHP has a start tag and and end tag: `<?php` and `?>` respectively. Anything inside these PHP tags is treated as PHP code and run on the *server*.

PHP code must be placed in a file with a `.php` extension. When someone connects to the server and asks it to load a file with a `.php` extension, the server then runs it as a PHP script. If you put PHP tags in a file with a `.html` or any extension other than `.php`, the web server won't run any PHP code, and the PHP code will be sent directly to the browser—which doesn't understand PHP code!

Let's take a look at a PHP script.

Example: PHP-RandomNumber[1]

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Random Number</title>
  </head>
  <body>
    <p>Generating a random number between 1 and 10:
      <?php
      echo rand(1, 10);
      ?>
    </p>
  </body>
</html>
```

To run this code, save it as `random.php` in the `public` directory and navigate to `https://v.je/random.php`.

Most of this is plain HTML. Only the line between `<?php` and `?>` is PHP code. `<?php` marks the start of an embedded PHP script and `?>` marks its end. The web server is asked to interpret everything between these two delimiters and

1. "https://github.com/spbooks/phpmysql7/tree/PHP-RandomNumber

convert it to regular HTML code before it sends the web page to the requesting browser. If you right-click inside your browser and choose **View Source** (the label may be different depending on the browser you're using) you can see that the browser is presented with something like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Random Number</title>
  </head>
  <body>
    <p>Generating a random number between 1 and 10:
      5
    </p>
  </body>
</html>
```

There's no HTML file on the server that contains this exact code. This HTML is generated dynamically on the server before being sent to the browser.

Try running the script a couple of times and notice how the number changes. PHP code has been used to generate a random number, but all signs of the PHP code have disappeared when viewing the source in the browser. In its place, the output of the script has appeared, and it looks just like standard HTML. This example demonstrates several advantages of server-side scripting:

- **Security**. In the example above, we placed a random number generated by the web server into the web page. If we had inserted the number using JavaScript, the number would be generated in the browser and someone could potentially amend the code to insert a specific number.

- **No browser compatibility issues**. PHP scripts are interpreted by the web server alone, so there's no need to worry about whether the language features you're using are supported by the visitor's browser.

- **Access to server-side resources**. The code on the server can do anything;

it's not limited to the capabilities of the browser. The resulting HTML could be generated from a database or an Excel file, or it could be the result of a calculation, such as the total of the user's shopping cart.

- **Reduced load on the client**. JavaScript can delay the display of a web page significantly (especially on mobile devices!), since the browser must download and then run the script before it can display the web page. With server-side code, this burden is passed to the web server, which you can make as beefy as your application requires (and your wallet can afford).

- **Choice**. When writing code that's run in the browser, the browser has to understand how to run the code given to it. All modern browsers understand HTML, CSS and JavaScript. To write some code that's run in the browser, you must use one of these languages. By running code on the server that generates HTML, you have a choice of many languages—one of which is PHP.

Basic Syntax and Statements

PHP syntax will be very familiar to anyone with an understanding of JavaScript, C, C++, C#, Objective-C, Java, Perl, or any other C-derived language. But if these languages are unfamiliar to you, or if you're new to programming in general, there's no need to worry.

A PHP script consists of a series of commands, or **statements**. Each statement is an instruction that must be followed by the web server before it can proceed to the next instruction. PHP statements, like those in the aforementioned languages, are always terminated by a semicolon (`;`).

This is a typical PHP statement:

```
echo 'This is a <strong>test</strong>!';
```

This is an `echo` statement, which is used to generate content (usually HTML code) to send to the browser. An `echo` statement simply takes the text it's

given and inserts it into the page's HTML code at the position where the PHP script was located.

In this case, we've supplied a string of text to be output: `This is a <strong>test</strong>!`. Notice that the string of text contains HTML tags (`<strong>` and `</strong>`), which is perfectly acceptable.

So, if we take this statement and put it into a complete web page, here's the resulting code.

Example: PHP-Echo[2]

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Test page</title>
  </head>
  <body>
    <p><?php echo 'This is a <strong>test</strong>!'; ?></p>
  </body>
</html>
```

If you place this file on your web server and then request it using a web browser, your browser will receive this HTML code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Test page</title>
  </head>
  <body>
    <p>This is a <strong>test</strong>!</p>
  </body>
</html>
```

2. https://github.com/spbooks/phpmysql7/tree/PHP-Echo

The *random.php* example we looked at earlier contained a slightly more complex *echo* statement:

```
echo rand(1, 10);
```

You'll notice that, in the first example, PHP is given some text to print directly, and in the second, PHP is given an instruction to follow. PHP tries to read anything that exists outside quotes as an instruction it must follow. Anything inside quotes is treated as text—or, to use the technical term, as a **string**. PHP doesn't process strings as commands to follow. They're treated as *data* in the application. You can do things with them, such as sending them to the browser, but PHP doesn't treat one string differently from another.

So the following code will pass the string *This is a test!* directly to the *echo* command:

```
echo 'This is a <strong>test</strong>!';
```

A string is signified using a start quote and an end quote. PHP will see the first *'* as the start of the string and find the next *'* and use that as the end of the string.

Anything outside of quotes is treated as a series of commands to run. The following is not valid code:

```
echo This is a <strong>test</strong>!;
```

Because the quotes have been removed, PHP will first try to run the command *This*, then the command *is*, followed by the command *a*, and so on. As none of these are valid commands in PHP, the above code would produce an error message. If you want to treat something as text, remember to put quotes around it!

In contrast, the following code will run a valid command—the built-in “function” *rand*—to generate a random number and then pass the result to

the echo command:

```
echo rand(1, 10);
```

A **function** is a special type of command that performs a specific task. Behind the scenes, PHP will do some processing and then generate a result. In this case, the `rand` function will produce a random number, but different functions perform different tasks.

You can quickly identify if something is a *function* because it's followed by parentheses. PHP has many built-in functions that let you do all sorts of things, such as sending email and working with information stored in various types of databases.

PHP won't try to run anything that's inside a string. The following code won't process the `rand` function:

```
echo 'rand(1, 10)';
```

PHP will see `'rand(1, 10)'` as a string, and will send the text `rand(1, 10)` to the browser, which probably isn't what you'd want to do. It's important to understand the difference between a string and *code*. PHP will see any text outside quotes as a series of commands it should follow. Anything inside quotes is a string and is *data* that PHP will work with.

PHP doesn't try to understand strings. They can contain any characters in any order. But code (anything not inside quotes)—which is essentially a series of *instructions*—must follow a rigid structure for a computer to understand it.

Code Editors and Syntax Highlighting

Using a code editor with syntax highlighting makes it easy to quickly see if something is a string or code. Strings will be a different color from code that needs to be processed.

The image below shows simple code highlighting in the Visual Studio Code editor.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Test page</title>
  </head>
  <body>
    <p><?php echo rand(1, 10); ?></p>

    <p><?php echo 'rand(1, 10)'; ?></p>
  </body>
</html>
```

2-1. An example of code highlighting in VS Code

PHP supports both single quotes (`'`) and double quotes (`"`) to encase strings. For most purposes, they're interchangeable. PHP developers tend to favor single quotes, because we deal with HTML code a lot, which tends to contain a lot of double quotes. For example:

```
echo '<a href="http://www.sitepoint.com">Click here</a>';
```

If double quotes were used at each end here, we'd need to tell PHP that the quote after `href=` is not the end of the string by placing a `\` before it (known as an **escape character**) and do the same with any double quotes we actually wanted to send to the browser as part of the HTML:

```
echo "<a href=\"http://www.sitepoint.com\">Click here</a>";
```

For this reason, PHP developers use single quotes, although there are some differences between single and double quotes. For our purposes here, though,

they're effectively interchangeable.

A function can be thought of as a miniature program within your program. You can instruct PHP to run the function by using its name (such as `rand` in our example earlier) followed by parentheses. The instruction `rand(1, 10)` tells PHP to run the function with the name `rand`. Running a function is also more commonly referred to as **calling** a function.

Most functions in PHP return a value when they're called. Once the value is returned, PHP behaves as if you'd actually just typed that returned value in your code instead. In the `echo rand(1, 10);` example, our `echo` statement contains a call to the `rand` function, which returns a random number as a string of text. The `echo` statement then outputs the value that was returned by the function call.

Every function in PHP can have one or more **arguments** that allow you to make the function behave in a slightly different way. The `rand` function takes two arguments: a minimum and maximum random number. By changing the values that are passed to the function, you're able to change the way it works. For example, if you wanted a random number between 1 and 50, you could use this code:

```
echo rand(1, 50);
```

We surround the arguments with parentheses (`(1, 50)`) for two reasons. First, they indicate that `rand` is a function that you want to call. Second, they mark the beginning and end of a list of arguments—PHP statements that you wish to provide—in order to tell the function what you want it to do. In the case of the `rand` function, you need to provide a minimum and a maximum value. Those values are separated by a comma.

Later on, we'll look at functions that take different kinds of arguments. We'll also consider functions that take no arguments at all. These functions will still need the parentheses, even though nothing will be typed between them.

Variables, Operators, and Comments

Variables

Variables in PHP are identical to variables in most other programming languages. For the uninitiated, a **variable** can be thought of as a name given to an imaginary box into which any value may be placed. The following statement creates a variable called `$testVariable` (all variable names in PHP begin with a dollar sign) and assigns it a value of `3`:

```
$testVariable = 3;
```

PHP is a **loosely typed** language. This means that a single variable may contain any type of data—be it a number, a string of text, or some other kind of value—and may store different types of values over its lifetime. If you were to type the following statement after the statement shown above, it would assign a new value to the existing `$testVariable`. While the `$testVariable` variable used to contain a number, it would now contain a string of text:

```
$testVariable = 'Three';
```

Once you have data stored in a variable, you can send it to the browser using the `echo` command from earlier:

```
$testVariable = 3;
echo $testVariable;
```

When this code runs, the digit “3” will be sent to the browser to be displayed on the page.

In addition to specific strings or numbers, it’s possible to store the result of a function and then use it later on in the script:

```
$randomNumber = rand(1, 10);
echo $randomNumber;
```

Operators

The equals sign we used in the last two statements is called the **assignment operator**, as it's used to assign values to variables. (There's a different way to actually indicate "equals" in PHP, as we'll see below.) Other operators may be used to perform various mathematical operations on values:

```
$testVariable = 1 + 1;  // assigns a value of 2
$testVariable = 1 - 1;  // assigns a value of 0
$testVariable = 2 * 2;  // assigns a value of 4
$testVariable = 2 / 2;  // assigns a value of 1
```

From these examples, you can probably tell that `+` is the **addition operator**, `-` is the **subtraction operator**, `*` is the **multiplication operator** and `/` is the **division operator**. These are all called **arithmetic operators**, because they perform arithmetic on numbers.

Comments

Each of the arithmetic lines above ends with a "comment" after the semicolon. **Comments** enable you to describe what your code is doing. They insert explanatory text into your code—text that the PHP interpreter will ignore.

Single-line comments begin with `//`. A single-line comment can be on its own line or, as in the example above, it can be placed at the end of a line of code.

If you want a comment to span several lines, start it with `/*`, and end it with `*/`. The PHP interpreter will ignore everything between these two delimiters. I'll be using comments throughout the rest of this book to help explain some of the code I present.

One operator that sticks strings of text together is called the **string concatenation operator**:

```
$testVariable = 'Hi ' . 'there!';  // Assigns a value of 'Hi there!'
```

Variables may be used almost anywhere you use a value. Consider this series of statements:

```
$var1 = 'PHP';          // assigns a value of 'PHP' to $var1
$var2 = 5;              // assigns a value of 5 to $var2
$var3 = $var2 + 1;      // assigns a value of 6 to $var3
$var2 = $var1;          // assigns a value of 'PHP' to $var2
$var4 = rand(1, 12);    // assigns a value to $var4 using the rand() function
echo $var1;             // outputs 'PHP'
echo $var2;             // outputs 'PHP'
echo $var3;             // outputs '6'
echo $var4;             // outputs the random number generated above
echo $var1 . ' rules!'; // outputs 'PHP rules!'
echo '$var1 rules!';    // outputs '$var1 rules!'
echo "$var1 rules!"     // outputs 'PHP rules!'
```

Note the last two lines in particular. If you place a variable inside *single* quotes, it will print the name rather than the contents of the variable. In contrast, when using double quotes, the variable in the string is replaced with the variable's contents.

Placing variables inside double quotes works in simple situations, but for most of this book it won't be usable, as we won't be using such simple code. So it's a good idea to get used to the practice of concatenation (shown in the third-from-last line: `echo $var1 . ' rules!';`).

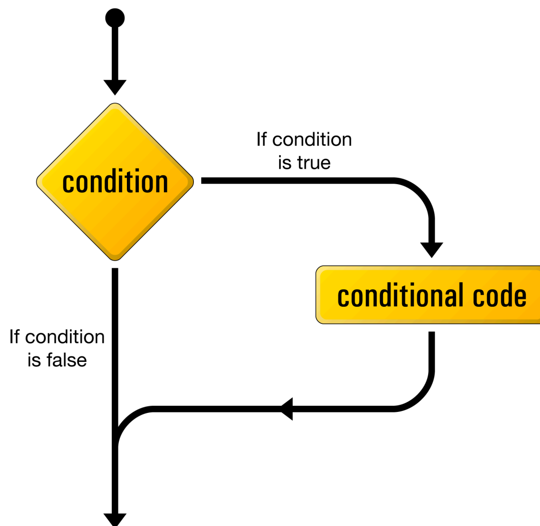
Control Structures

The examples of PHP code we've seen so far have been either one-statement scripts that output a string of text to the web page, or a series of statements that were to be executed one after the other in order. If you've ever written programs in other languages (such as JavaScript, Objective-C, Ruby, or Python), you already know that practical programs are rarely so simple.

PHP, just like any other programming language, provides facilities that enable you to affect the flow of control. That is, the language contains special statements that you can use to deviate from the one-after-another execution order that has dominated our examples so far. Such statements are called **control structures**. Don't understand? Don't worry! A few examples will illustrate it perfectly.

If Statements

The most basic, and most often used, control structure is the `if` statement. The flow of a program through an `if` statement can be visualized as shown below.



2-2. The logical flow of an if statement

Here's what an `if` statement looks like in PHP code:

```
if (condition) {
  ⋮ conditional code to be executed if condition is true
}
```

This control structure lets us tell PHP to execute a set of statements only if some condition is met.

For example, we might want to create a game that mimics a dice roll and in which you have to roll a six to win. The dice roll can be modeled using the `rand()` function we used earlier, setting the minimum and maximum from 1 to 6:

```
$roll = rand(1, 6);

echo 'You rolled a ' . $roll;
```

To print out a message if the player rolls a six and wins, you can use an `if` statement.

Example: PHP-DiceRoll[3]

```
$roll = rand(1, 6);

echo 'You rolled a ' . $roll;

if ($roll == 6) {
  echo 'You win!';
}
```

The `==` used in the condition above is the **equals operator**, which is used to compare two values to see whether they're equal. This is quite different from using a single `=`, which is used for *assignment*, and can't be used for comparison.

The `if` statement uses braces (`{` and `}`) to surround the code you want to run only when the condition is met. You can place as many lines of code as you like between the braces; the code will only be run when the condition is met. Any code placed after the closing brace (`}`) will be run all the time:

```
$roll = rand(1, 6);

echo 'You rolled a ' . $roll;
```

---

3. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll

```
if ($roll == 6) {
  echo 'You win!'; // This line will only be printed if you roll a 6
}

echo 'Thanks for playing'; // This line will always be printed
```

Assignment and Equality

Remember to type the double-equals (`==`). A common mistake among beginner PHP programmers is to type a condition like this with a single equals sign:

```
if ($roll = 6)     // Missing equals sign!
```

This condition is using the assignment operator (`=`) instead of the equals operator (`==`). Consequently, instead of comparing the value of `$roll` to the number `6` , it will actually *set* the value of `$roll` to `6` . Oops!

To make matters worse, the `if` statement will use this assignment operation as a condition, which it will consider to be true, so the conditional code within the `if` statement will always be executed, regardless of what the original value of `$roll` happened to be.

If you save this as `diceroll.php` and run the code, you'll see the random number being generated. If you run it until you win, you'll see this in the browser:

```
You rolled a 6You win!Thanks for playing
```

This isn't very pretty. But because PHP outputs HTML, you can add some paragraph tags in order to format the output.

Example: PHP-DiceRoll-Formatted[4]

```
$roll = rand(1, 6);

echo '<p>You rolled a ' . $roll . '</p>';

if ($roll == 6) {
  echo '<p>You win!</p>';
}

echo '<p>Thanks for playing</p>';
```

If you run the updated code, you'll see that it now prints this in the browser:

```
You rolled a 6

You win!

Thanks for playing
```

This is much more user friendly. To make the game itself more user friendly, you might want to display a different message to people who didn't roll a 6 and didn't win. This can be done with an `else` statement. The `else` statement must follow an `if`, and will be run if the condition isn't met.

Example: PHP-DiceRoll-Else[5]

```
$roll = rand(1, 6);

echo '<p>You rolled a ' . $roll . '</p>';

if ($roll == 6) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
}

echo '<p>Thanks for playing</p>';
```

4. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-Formatted
5. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-Else"

A Lucky Escape

Because the word `didn't` contains a single quote, it needs to be *escaped*. By preceding the single quote with a backslash (`\`), it tells PHP not to treat the `'` in `didn't` as the end of the string.

With an else statement, one (and only one!) of the two blocks of code is guaranteed to run. Either the code in the `if` block will run if the condition is met, or the code in the `else` block will run if it isn't.

Conditions can be more complex than a single check for equality. An `if` statement can contain more than one condition. For example, imagine if the game were adjusted so that both `5` and `6` were winning numbers. The `if` statement could be changed to the following.

Example: PHP-DiceRoll-Or[6]

```
if ($roll == 6 || $roll == 5) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
}
```

The double pipe operator (`||`) means "or". The condition above is now met if either condition is met. This can be read as "If you roll a 6 or you roll a 5".

However, this can be expressed in an even better way. `if` statements aren't limited to using the equals (`==`) operator. They can also utilize the mathematical greater than (`>`) and less than (`<`) operators. The `if` statement above could also be done with a single expression.

Example: PHP-DiceRoll-Greater[7]

---

6. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-Or
7. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-Greater

```
if ($roll > 4) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
}
```

The condition `$roll > 4` will be met if the value stored in the `$roll` variable is greater than `4`, allowing us to have 5 and 6 as winning numbers with a single condition. If we wanted 4, 5 and 6 as winning numbers, the condition could be changed to `$roll > 3`.

Like the "or" expression (`||`), there's an "and" expression that's only met when both conditions are met. We could expand the game to include two dice and require players to roll two sixes to win.

Example: PHP-DiceRoll-TwoDice[8]

```
$roll1 = rand(1, 6);
$roll2 = rand(1, 6);

echo '<p>You rolled a ' . $roll1 . ' and a ' . $roll2 . '</p>';

if ($roll1 == 6 && $roll2 == 6) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
}

echo '<p>Thanks for playing</p>';
```

The condition `$roll1 == 6 && $roll2 == 6` will only be met if `$roll1 == 6` is true *and* `$roll2 == 6`. This means that the player has to roll a 6 on both dice to win the game. If we changed the and (`&&`) to an or (`||`)—that is, `if ($roll1 == 6 || $roll2 == 6)`—the player would win if they rolled a 6 on either dice.

---

8. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-TwoDice

We'll look at more complicated conditions as the need arises. For the time being, a general familiarity with `if … else` statements is sufficient.

and **and** or

PHP also allows the use of `or` in place of `||` and `and` in place of `&&`. For example:

```
if ($roll == 6 or $roll == 5) { … }
```

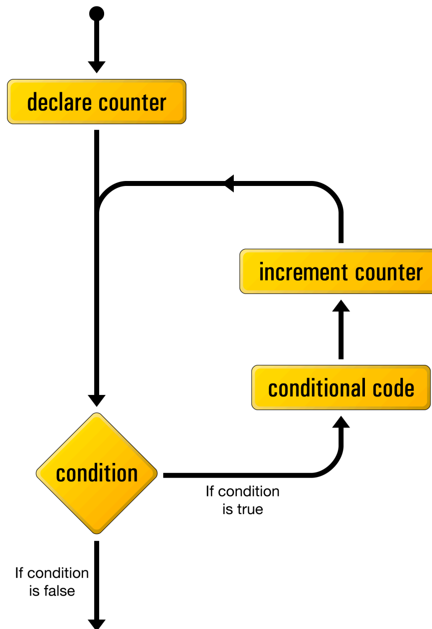
There are some minor differences between the way `or` and `||` work that can lead to unexpected behavior. Generally speaking, avoid the "spelled out" operators. Sticking to the double pipe (`||`) and double ampersand (`&&`) will help prevent a number of confusing bugs.

Loops

Another type of control structure that's very useful is a "loop". **Loops** allow you to repeat the same lines of code over and over. Two important kind of loops are `for` loops and `while` loops. Let's look at how they work.

For Loops

The `for` loop is used when you know up front how many times you need to run the same code. The image below shows the flow of a `for` loop.



2-3. The logical flow of a for loop

Here's what it looks like in code:

```
for (declare counter; condition; increment counter) {
  ⋮ statement(s) to execute repeatedly as long as condition is true
}
```

The `declare counter` statement is executed once at the start of the loop. The `condition` statement is checked each time through the loop before the statements in the body are executed. The `increment counter` statement is executed each time through the loop after the statements in the body.

To count to ten using a `for` loop, you can use the following code:

```
for ($count = 1; $count <= 10; $count++) {
  echo $count . ' ';
}
```

This looks quite scary, as there's a lot going on, but let me break it down for

you:

- `$count = 1;` . You're already familiar with this code. It's creating a variable called `$count` and setting the value to `1` .
- `$count <= 10;` . This is the condition. It can be read as "keep looping while `$count` is less than or equal to 10".
- `$count++` . This says "add 1 to the counter each time". It's identical to `$count = $count + 1` .
- `echo $count . ' ';` . This prints the value of the counter followed by a space.

The condition in this example uses the `<=` operator. This acts similarly to the less than operator (`<`), but evaluates to true if the number being compared is *less than or equal to* the second. Other available operators include `>=` (greater than or equal) and `!=` (not equal).

All three parts of the `for` loop are combined to produce the complete loop. Although, at first glance, the code seems a little more difficult to read, putting all the code that deals with controlling the loop in the same place actually makes it easier to understand once you're used to the syntax. Many of the examples in this book will use `for` loops, so you'll have plenty of opportunities to practice reading them.

You can change each part of the `for` loop to have different results. For example, let's look at how to add `3` each time you can change the `for` loop by amending the last part.

The final part of the loop, `$count++` , says *add 1 to the value of* `$count` , and it's a short way of writing `$count = $count + 1` .

By changing `$count++` to `$count = $count + 3` , the for loop can be used to count in threes.

Example: PHP-For[9]

```
for ($count = 1; $count <= 10; $count = $count + 3) {
  echo $count . ' ';
}
```

This will result in the following:

```
1 4 7 10
```

`for` loops can be combined with other statements—such as `if` statements—to perform specific tasks on each iteration. For example, rather than refreshing the page on our dice game each time, we might want to roll the dice ten times and print the results.

Example: PHP-DiceRoll-ManyDice[10]

```
for ($count = 1; $count <= 10; $count++) {
  $roll = rand(1, 6);
  echo '<p>You rolled a ' . $roll . '</p>';

  if ($roll == 6) {
    echo '<p>You win!</p>';
  }
  else {
    echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
  }
}

echo '<p>Thanks for playing</p>';
```

This lets us roll the dice ten times without needing to refresh the page each time. Using a loop is functionally identical to copying and pasting the code ten times, and will produce the exact same outcome as the following:

```
$roll = rand(1, 6);
echo '<p>You rolled a ' . $roll . '</p>';
```

---

9. https://github.com/spbooks/phpmysql7/tree/PHP-For
10. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-ManyDice"

```
if ($roll == 6) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
}

$roll = rand(1, 6);
echo '<p>You rolled a ' . $roll . '</p>';

if ($roll == 6) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
}

$roll = rand(1, 6);
echo '<p>You rolled a ' . $roll . '</p>';

if ($roll == 6) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
}

// and so on …
```

The computer doesn't care which method you use: you can copy/paste or use a loop. It will just run the code. However, as a developer, you'll quickly realize that a loop is the better choice. If you wanted to update the code to also allow `5` as a winning number, you'd need to update the condition in ten different places. Using a loop, you can change the code in one place and it will affect each iteration of the loop. If you ever find yourself copy/pasting code, there's always a better way of achieving what you're trying to do.

For Loop Challenges

Given your new knowledge of loops, you should be able to start coding for yourself. Can you complete these challenges?

- **Challenge 1**: print all the odd numbers from 21 to 99.
- **Challenge 2**: print the nine times table up to 12 x 9 (`9 18 27` … etc.) without using the multiplication operator (`*`) or an `if` statement.
- **Challenge 3**: print the nine times table in this format:

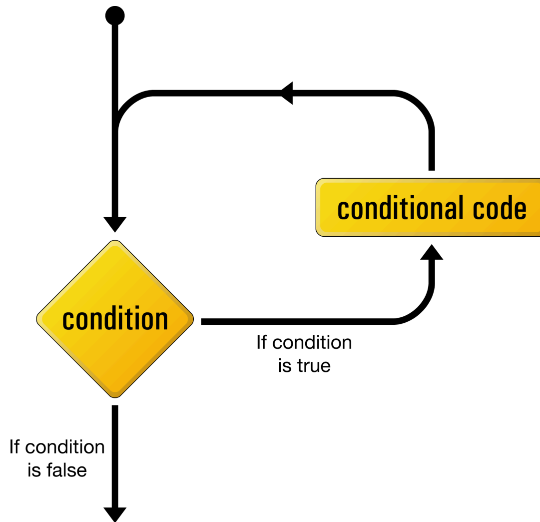
```
 1x9 = 9
2x9 = 18
… etc.
```

This time, you'll need to use the multiplication operator!

While Loops

Another often-used PHP control structure is the `while` loop. Where the `if … else` statement allows us to choose whether or not to execute a set of statements depending on some condition, the `while` loop allows us to use a condition to determine how many *times* we'll execute a set of statements repeatedly.

The following image shows how a `while` loop operates.



2-4. The logical flow of a while loop

Here's what a `while` loop looks like in code:

```
while (condition) {
  ⋮ statement(s) to execute repeatedly as long as condition is true
}
```

The `while` loop works very similarly to an `if` statement. The difference arises when the condition is true and the statement(s) are executed. Instead of continuing the execution with the statement that follows the closing brace (`}`), the condition is checked again. If the condition is still true, the statement(s) are executed a second time, and a third, and will continue to be executed as long as the condition remains true. The first time the condition evaluates false (whether it's the first time it's checked, or the hundredth), the execution jumps immediately to the statement that follows the `while` loop, after the closing brace.

You can use a `while` loop to create a counter that has a similar result to a `for` loop.

Example: PHP-WhileCount[11]

```
$count = 1;
while ($count <= 10) {
  echo $count . ' ';
  ++$count;
}
```

This works in exactly the same way as a `for` loop, and you'll notice a lot of the same statements in different places. This code may look a bit frightening, I know, but let me talk you through it line by line:

- `$count = 1;` . The first line creates a variable called `$count` and assigns it a value of `1` .
- `while ($count <= 10)` . The second line is the start of a `while` loop, the condition being that the value of `$count` is less than or equal (`<=`) to 10.
- `{` . The opening brace marks the beginning of the block of conditional code for the `while` loop. This conditional code is often called the **body** of the loop, and is executed over and over again, as long as the condition holds true.
- `echo $count . ' ';` . This line simply outputs the value of `$count` , followed by a space.
- `++$count;` . The fourth line adds `1` to the value of `$count` . `++$count` is a shortcut for `$count = $count + 1` . `$count++` will also work here! The position of the `++` can be important, but in this case it doesn't matter. If the `++` is before the variable name, the counter is incremented before the value is read. When `$count` is zero, the code `echo ++$count;` will print `1` , whereas `echo $count++;` will print `0` . Be careful when using `++` , as putting it in the wrong place can cause bugs.
- `}` . The closing brace marks the end of the `while` loop's body.

So here's what happens when this code is executed. The first time the condition is checked, the value of `$count` is `1` , so the condition is definitely true. The value of `$count` (`1`) is output, and `$count` is given a new value of `2` . The condition is still true the second time it's checked, so the value (`2`) is output and a new value (`3`) is assigned. This process continues, outputting

---

11. https://github.com/spbooks/phpmysql7/tree/PHP-WhileCount

the values `3`, `4`, `5`, `6`, `7`, `8`, `9`, and `10`. Finally, `$count` is given a value of `11`, and the condition is found to be false, which ends the loop.

The net result of the code is shown in the following image.

1 2 3 4 5 6 7 8 9 10

2-5. The net result of the while loop code

While loops aren't generally used for simple counters like this; it's normally the job of the `for` loop. Although you can create a counter with a `while` loop, usually they're used to keep running code until something happens. For example, we might want to keep rolling the dice until we get a 6. There's no way to know how many dice rolls will be needed when we write the code: it could take one roll or hundreds to get a six. So you can place the dice roll in a `while` loop.

Example: PHP-DiceRoll-While[12]

```
$roll = 0;
while ($roll != 6) {
  $roll = rand(1, 6);
  echo '<p>You rolled a ' . $roll . '</p>';

  if ($roll == 6) {
    echo '<p>You win!</p>';
  }
  else {
    echo '<p>Sorry, you didn\'t win, better luck next time!</p>';
  }
}
```

This will keep rolling the dice until a 6 is rolled. Each time you run the code, it will take a different number of rolls before you win.

---

12. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-While

The while statement uses the condition `$roll != 6`. In order for the `while` loop to be run the first time, the `$roll` variable must be set to a value for the initial comparison. That's the purpose of the `$roll = 0;` line above the `while` loop. By setting the value to zero initially, when the `while` loop runs the first time, the condition `$roll != 6` is met, because `$roll` is equal to zero, not 6, and the loop will start. If the `$roll` variable isn't created prior to starting the loop, you'll get an error, because the `$roll` variable hasn't been set to anything before it's used.

There's a variant of the `while` loop called `do … while`, which is useful in these kinds of cases. It allows you to run some code without a condition and then run it again if the code isn't set. This takes the following structure:

```
do {
 statement(s) to execute and then repeat if the condition is true
}
while (condition);
```

For the dice-roll example above, this allows you to ignore the first line.

Example: PHP-DiceRoll-DoWhile[13]

```
do {
  $roll = rand(1, 6);
  echo '<p>You rolled a ' . $roll . '</p>';

  if ($roll == 6) {
    echo '<p>You win!</p>';
  }
  else {
    echo '<p>Sorry, you didn\'t win, better luck next time!</p>';
  }
}
while ($roll != 6);
```

This time, because the condition is at the bottom, by the time the `while`

13. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-DoWhile

statement is run, the `$roll` variable has been given a value, so you don't need to give it an initial value of zero to force the loop to run the first time.

Coding Style

PHP doesn't mind how you format your code, and whitespace is ignored. You could write the previous example like this:

```
do {
  $roll = rand(
  1,
  6);

  echo '<p>You rolled a ' . $roll . '</p>';

  if (
      $roll == 6
  )
  {
    echo '<p>You win!</p>';
  }
  else
  {
    echo '<p>Sorry, you didn\'t win, better luck next time!</p>';
  }
}
while ($roll != 6);
```

The script will execute in the exact same way. Different programmers have different preferred styles, such as using tabs or spaces for indentation, or placing the opening brace on the same line as the statement or after it. There are even coding style guides[14] that dictate where braces and brackets should be placed and how code should be formatted. But the computer doesn't care whether a brace is on the same line or the next line, so use whatever style you feel most comfortable with.

---

14. https://www.php-fig.org/psr/psr-12/

Arrays

An **array** is a special kind of variable that contains multiple values. If you think of a variable as a box that contains a value, an array can be thought of as a box with compartments where each compartment is able to store an individual value.

To create an array in PHP, use square brackets (`[` and `]`) containing the values you want to store, separated by commas:

```
$myArray = ['one', 2, '3'];
```

The `array` Keyword versus Square Brackets

Arrays in PHP can also be defined using the `array` keyword. The following code is equivalent to the square bracket notation above:

```
$myArray = array('one', 2, 3);
```

The square bracket notation was introduced in PHP 5.4 and is preferred by PHP developers, as it's less to type, and square brackets are more easily visible among round brackets in control structures like `if` statements and `while` loops.

This code creates an array called `$myArray` that contains three values: `'one'` , `2` , and `'3'` . Just like an ordinary variable, each space in an array can contain any type of value. In this case, the first and third spaces contain strings, while the second contains a number.

To access a value stored in an array, you need to know its **index**. Typically, arrays use numbers as indices to point to the values they contain, starting with zero. That is, the first value (or element) of an array has index 0, the second has index 1, the third has index 2, and so on. Therefore, the index of the nth element of an array is n–1. Once you know the index of the value you're

interested in, you can retrieve that value by placing that index in square brackets after the array variable name:

```
echo $myArray[0];    // outputs 'one'
echo $myArray[1];    // outputs '2'
echo $myArray[2];    // outputs '3'
```

Each value stored in an array is called an **element**. You can use a key in square brackets to add new elements, or assign new values to existing array elements:

```
$myArray[1] = 'two';     // assign a new value
$myArray[3] = 'four';    // create a new element
```

You can also add elements to the end of an array using the assignment operator as usual, but leaving empty the square brackets that follow the variable name:

```
$myArray[] = 'five';
echo $myArray[4];    // outputs 'five'
```

Array elements can be used like any other variable, and in a lot of cases choosing to use an array or multiple variables will depend on the programmer's preference. However, arrays can be used to solve problems that normal variables can't!

Remember the dice game from the last section? It would be more user-friendly if it showed the English word rather than the numeral for the result. For example, instead of "You rolled a 3" or "You rolled a 6", it might be nicer to read "You rolled a three" or "You rolled a six".

To do this, we need some way of converting from a numeral to the English word for that number. This is possible with a series of `if` statements.

Example: PHP-DiceRoll-English-If[15]

15. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-English-If

```
$roll = rand(1, 6);

if ($roll == 1) {
  $english = 'one';
}
else if ($roll == 2) {
  $english = 'two';
}
else if ($roll == 3) {
  $english = 'three';
}
else if ($roll == 4) {
  $english = 'four';
}
else if ($roll == 5) {
  $english = 'five';
}
else if ($roll == 6) {
  $english = 'six';
}

echo '<p>You rolled a ' . $english . '</p>';

if ($roll == 6) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win. Better luck next time!</p>';
}*
```

This solution works, but it's very inefficient, as you need to write an `if` statement for each possible dice roll. Instead, you can use an array to store each roll value:

```
$english = [
  1 => 'one',
  2 => 'two',
  3 => 'three',
  4 => 'four',
  5 => 'five',
  6 => 'six'
];
```

The `=>` notation (known as a **double arrow operator**) allows you to define both the keys and the values when creating the array. This is equivalent to the following:

```
$english = [];
$english[1] = 'one';
$english[2] = 'two';
$english[3] = 'three';
$english[4] = 'four';
$english[5] = 'five';
$english[6] = 'six';
```

Although these are equivalent, the code required to use the shorthand notation is a lot quicker to type, and is arguably easier to read and easier to understand.

Now that the array is created, it's possible to read each English word from it:

```
echo $english[3];    // Prints "three"
echo $english[5];    // Prints "five"
```

In PHP, a number like `3` can be replaced with a variable that contains that value. This is also possible with array keys. For example:

```
$var1 = 3;
$var2 = 5;

echo $english[$var1];    // Prints "three"
echo $english[$var2];    // Prints "five"
```

Knowing this, we can piece it all together and adjust the dice game to display the English word of the dice roll by reading the relevant value from the array using the `$roll` variable.

Example: PHP-DiceRoll-English-Array[16]

---

16. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-English-Array"

```
$english = [
  1 => 'one',
  2 => 'two',
  3 => 'three',
  4 => 'four',
  5 => 'five',
  6 => 'six'
];

$roll = rand(1, 6);

echo '<p>You rolled a ' . $english[$roll] . '</p>';

if ($roll == 6) {
  echo '<p>You win!</p>';
}
else {
  echo '<p>Sorry, you didn\'t win, better luck next time!</p>';
}
```

As you can see, this is a lot cleaner and tidier than a long list of `if` statements. There are two big advantages here.

Firstly, if you wanted to represent a ten-sided dice, it's a lot easier to add to the array than add an extra `if` statement for each number.

Secondly, the array is reusable. For the version with two dice, you can just reuse the `$english` array rather than repeating all the `if` statements for each dice roll.

Example: PHP-DiceRoll-English-If-TwoDice[17]

```
$roll1 = rand(1, 6);
$roll2 = rand(1, 6);

if ($roll1 == 1) {
  $english = 'one';
}
```

17. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-English-If-TwoDice"

```
else if ($roll1 == 2) {
  $english = 'two';
}
else if ($roll1 == 3) {
  $english = 'three';
}
else if ($roll1 == 4) {
  $english = 'four';
}
else if ($roll1 == 5) {
  $english = 'five';
}
else if ($roll1 == 6) {
  $english = 'six';
}

if ($roll2 == 1) {
  $englishRoll2 = 'one';
}
else if ($roll2 == 2) {
  $englishRoll2 = 'two';
}
else if ($roll2 == 3) {
  $englishRoll2 = 'three';
}
else if ($roll2 == 4) {
  $englishRoll2 = 'four';
}
else if ($roll2 == 5) {
  $englishRoll2 = 'five';
}
else if ($roll2 == 6) {
  $englishRoll2 = 'six';
}

echo '<p>You rolled a ' . $english . ' and a ' . $englishRoll2 . '</p>';
```

Instead, the array can be used for both rolls.

Example: PHP-DiceRoll-English-Array-TwoDice[18]

---

18. https://github.com/spbooks/phpmysql7/tree/PHP-DiceRoll-English-Array-TwoDice

```
$english = [
  1 => 'one',
  2 => 'two',
  3 => 'three',
  4 => 'four',
  5 => 'five',
  6 => 'six'
];

$roll1 = rand(1, 6);
$roll2 = rand(1, 6);

echo '<p>You rolled a ' . $english[$roll1] . ' and a ' . $english[$roll2]
↪ . '</p>';
```

While numbers are the most common choice for array indices, there’s another possibility. You can also use strings as indices to create what’s called an **associative array**. It’s called this because it *associates* values with meaningful indices. In this example, we associate a date (in the form of a string) with each of three names:

```
$birthdays['Kevin'] = '1978-04-12';
$birthdays['Stephanie'] = '1980-05-16';
$birthdays['David'] = '1983-09-09';
```

Like the numerical indexes, you can use the shorthand notation for associative arrays as well:

```
$birthdays = [
  'Kevin' => '1978-04-12',
  'Stephanie' => '1980-05-16',
  'David' => '1983-09-09'
];
```

Now, if we want to know Kevin’s birthday, we look it up using the name as the index:

```
echo 'Kevin\'s birthday is: ' . $birthdays['Kevin'];
```

This type of array is especially important when it comes to user interaction in PHP, as we'll see shortly. I'll also demonstrate other uses of arrays throughout this book.

Escaping the Apostrophe

Because `Kevin's` contains an apostrophe (single quote) and PHP would see this as the end of the string, it must be escaped with a `\` so that PHP treats it as part of the string, rather than marking the end.

User Interaction and Forms

Database-driven websites often need to do more than dynamically generate pages based on database data. You also need to provide some degree of interactivity, even if it's just a search box.

Because JavaScript code can live in the browser, it allows users to have all sorts of immediate interactions with a web page. Server-side scripting languages such as PHP have a more limited scope when it comes to support for user interaction. As PHP code is only activated when a request is made to the server, user interaction occurs solely in a back-and-forth fashion: the user sends requests to the server, and the server replies with dynamically generated pages.

The key to creating interactivity with PHP is to understand the techniques we can employ to send information about a user's interaction, along with a request for a new web page. As it turns out, PHP makes this quite easy.

JavaScript and Interactivity

As the Web has evolved, so too have the options for communication between the front and back ends of a web app. JavaScript allows for communication with a back end without page reloading. A JavaScript framework such as Vue.js, for example, can communicate with PHP frameworks like Laravel. For the purposes of this book, however, we'll focus on what can be done with PHP alone.

Passing Variables in Links

The simplest way to send information along with a page request is to use the URL query string. If you've ever noticed a URL containing a question mark that follows the filename, you've seen this technique in use. For example, if you search for "SitePoint" on Google, it will take you to a search result page with a URL like this:

```
http://www.google.com/search?hl=en&q=SitePoint
```

See the question mark in the URL? The text that follows the question mark contains your search query (`SitePoint`). That information is being sent along with the request for `http://www.google.com/search`.

Let's code up an easy example of our own. Create a regular HTML file called `name.html` (no `.php` filename extension is required, since there won't be any PHP code in this file) and insert this link:

```
<a href="name.php?name=Tom">Hi, I'm Tom!</a>
```

This is a link to a file called `name.php`, but as well as linking to the file, you're also passing a variable along with the page request. The variable is passed as part of the query string, which is the portion of the URL that follows the question mark. The variable is called `name`, and its value is `Tom`. So, you've created a link that loads `name.php`, and which informs the PHP code

contained in that file that `name` equals `Tom`.

To really understand the effect of this link, we need to look at `name.php`. Create it as a new file, but this time, note the `.php` filename extension. This tells the web server that it can expect to interpret some PHP code in the file. In the body of this new web page, type the following.

Example: PHP-GET[19]

```
<?php
$name = $_GET['name'];
echo 'Welcome to our website, ' . $name . '!';
?>
```

Now, put these two files (`name.html` and `name.php`) in the `public` folder, and load the first file in your browser. (The URL should be `https://v.je/name.html`.) Click the link in that first page to request the PHP script. The resulting page should say “Welcome to our website, Tom!”, as shown in the image below.

2-6. The welcome message, seen in the browser

19. https://github.com/spbooks/phpmysql7/tree/PHP-GET

Let’s take a closer look at the code that made this possible. This is the most important line:

```
$name = $_GET['name'];
```

Using what you learned from the “Arrays” section above, you may be able to figure out what this line does. It assigns the value stored in the `'name'` element of the array called `$_GET` to a new variable called `$name`. But where does the `$_GET` array come from?

It turns out that `$_GET` is one of a number of variables that PHP automatically creates when it receives a request from a browser. PHP creates `$_GET` as an array variable that contains any values passed in the URL query string. `$_GET` is an associative array, so the value of the `name` variable passed in the query string can be accessed as `$_GET['name']`. Your `name.php` script assigns this value to an ordinary PHP variable (`$name`), then displays it as part of a text string using an `echo` statement:

```
echo 'Welcome to our website, ' . $name . '!';
```

The value of the `$name` variable is inserted into the output string using the string concatenation operator (`.`) that we looked at in the “Variables, Operators, and Comments” section.

But watch out! There’s a security hole lurking in this code! Although PHP is an easy programming language to learn, it turns out it’s also especially easy to introduce security issues into websites using PHP if you’re unaware of what precautions to take. Before we go any further with the language, I want to make sure you’re able to spot and fix this particular security issue, since it’s probably the most common one on the Web today.

The security issue here stems from the fact that the `name.php` script is generating a page containing content that’s under the control of the user—in this case, the `$name` variable. Although the `$name` variable will normally receive its value from the URL query string in the link on the `name.html` page,

a malicious user could edit the URL to send a different value for the `name` variable.

To see how this would work, click the link in `name.html` again. When you see the resulting page (with the welcome message containing the name "Tom"), take a look at the URL in the address bar of your browser. It should look similar to this:

```
https://v.je/name.php?name=Tom
```

Edit the URL to insert a `<b>` tag before the name and a `</b>` tag following the name:

```
https://v.je/name.php?name=<b>Tom</b>
```

Hit `enter` to load this new URL, and note that the name in the page is now bold, as shown in the following image.

2-7. The name shown in bold

Converting Code to Escape Sequences

You might notice that some browsers will automatically convert the `<` and `>` characters into URL escape sequences (`%3C` and `%3E` respectively), but either way, PHP will receive the same value.

See what's happening here? The user can type any HTML code into the URL, and your PHP script includes it in the code of the generated page without question. If the code is as innocuous as a `<b>` tag, there's no problem, but a malicious user could include sophisticated JavaScript code that performs some low action like stealing the user's password. All the attacker would have to do is publish the modified link on some other site under the attacker's control, and then entice one of your users to click it. The attacker could even embed the link in an email and send it to your users. If one of your users clicked the link, the attacker's code would be included in your page and the trap would be sprung!

I hate to scare you with this talk of malicious hackers attacking your users by turning your own PHP code against you, particularly when you're only just learning the language. The fact is that PHP's biggest weakness as a language is how easy it is to introduce security issues like this. Some might say that much of the energy you spend learning to write PHP to a professional standard is spent on avoiding security issues. The sooner you're exposed to these issues, however, the sooner you become accustomed to avoiding them, and the less of a stumbling block they'll be for you in future.

So, how can we generate a page containing the user's name without opening it up to abuse by attackers? The solution is to treat the value supplied for the `$name` variable as plain text to be displayed on your page, rather than as HTML to be included in the page's code. This is a subtle distinction, so let me show you what I mean.

Open up your `name.php` file again and edit the PHP code it contains so that it looks like the code below.

Example: PHP-GET-Sanitized[20]

```
<?php
$name = $_GET['name'];
echo 'Welcome to our website, ' .
  htmlspecialchars($name, ENT_QUOTES, 'UTF-8') . '!';
?>
```

There's a lot going on in this code, so let me break it down for you. The first line is the same as it was previously, assigning to `$name` the value of the `'name'` element from the `$_GET` array. But the `echo` statement that follows it is drastically different. Whereas previously we simply dumped the `$name` variable—naked—into the `echo` statement, this version of the code uses the built-in PHP function `htmlspecialchars` to perform a critical conversion.

Remember, the security hole occurs because, in `name.php`, HTML code in the `$name` variable is dumped directly into the code of the generated page, and can therefore do anything that HTML code can do. What `htmlspecialchars` does is convert "special HTML characters" like `<` and `>` into HTML character entities like `<` and `>`, which prevents them from being interpreted as HTML code by the browser. I'll demonstrate this for you in a moment.

First, let's take a closer look at this new code. The call to the `htmlspecialchars` function is the first example in this book of a PHP function that takes more than one argument. Here's the function call all by itself:

```
htmlspecialchars($name, ENT_QUOTES, 'UTF-8')
```

The first argument is the `$name` variable (the text to be converted). The second argument is the PHP "constant" `ENT_QUOTES`, which tells `htmlspecialchars` to convert single and double quotes in addition to other special characters.

The third parameter is the string `'UTF-8'`, which tells PHP what character

20. https://github.com/spbooks/phpmysql7/tree/PHP-GET-Sanitized

encoding to use to interpret the text you give it.

Character Encoding

You may have discerned that all the example HTML pages in this book contain the following meta element near the top:

```
<meta charset="utf-8">
```

This element tells the browser receiving this page that the HTML code of the page is encoded as UTF-8 text.

UTF-8 is one of many standards for representing text as a series of ones and zeros in computer memory, called character encodings. If you're curious to learn all about character encodings, check out "The Definitive Guide to Web Character Encoding"[21].

In a few pages, we'll reach the section on "Passing Variables in Forms". By encoding your pages as UTF-8, your users can submit text containing thousands of foreign characters that your site would otherwise be unable to handle.

Unfortunately, many of PHP's built-in functions, such as `htmlspecialchars`, assume you're using the much simpler ISO-8859-1 (or Latin-1) character encoding by default. Therefore, you need to let them know you're using UTF-8 when utilizing these functions.

If you can, you should also tell your text editor to save your HTML and PHP files as UTF-8 encoded text. This is only required if you want to type advanced characters (such as curly quotes or dashes) or foreign characters (like "é") into your HTML or PHP code. The code in this book plays it safe and uses HTML entity references (for example, `’` for a curly right quote), which will work regardless.

---

21. http://www.sitepoint.com/article/guide-web-character-encoding/

PHP Constants

A PHP **constant** is like a variable whose value you're unable to change. Unlike variables, constants don't start with a dollar sign. PHP comes with a number of built-in constants like `ENT_QUOTES` that are used to control built-in functions like `htmlspecialchars`.

Open up `name.html` in your browser and click the link that now points to your updated `name.php`. Once again, you'll see the message "Welcome to our website, Tom!" As you did before, modify the URL to include `<b>` and `</b>` tags surrounding the name:

```
https://v.je/name.php?name=<b>Tom</b>
```

When you hit `enter` this time, instead of the name turning bold in the page, you should see the actual text you typed, as shown in the following image.

2-8. It sure is ugly, but it's secure!

If you view the source code of the page, you can confirm that the `htmlspecialchars` function did its job and converted the `<` and `>` characters into the `<` and `>` entity references respectively. This prevents malicious users from injecting unwanted code into your site. If they try anything like that, the code is harmlessly displayed as plain text on the page.

We'll make extensive use of the *htmlspecialchars* function throughout this book to guard against this sort of security hole. No need to worry too much if you're having trouble grasping the details of how to use it just at the moment. Before long, you'll find its use becomes second nature. For now, let's look at some more advanced ways of passing values to PHP scripts when we request them.

Passing a single variable in the query string was nice, but it turns out you can pass *more* than one value if you want to! Let's look at a slightly more complex version of the previous example. Open up your *name.html* file again, and change the link to point to *name.php* with this more complicated query string:

```
<a href="name.php?firstname=Tom&amp;lastname=Butler">Hi,
  I'm Tom Butler!</a>
```

This time, our link passes two variables: *firstname* and *lastname*. The variables are separated in the query string by an ampersand (*&*, which should be written as *&* in HTML—yes, even in a link URL! … although, if you do wrongly use *&*, browsers will *mostly* fix it for you). You can pass even more variables by separating each *name=value* pair from the next with an ampersand.

As before, we can use the two variable values in our *name.php* file.

Example: PHP-GET-TwoVars[22]

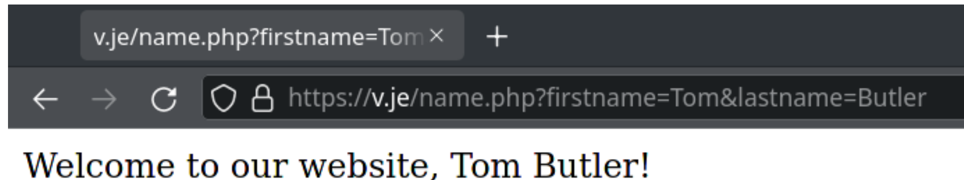
```
<?php
$firstName = $_GET['firstname'];
$lastName = $_GET['lastname'];
echo 'Welcome to our website, ' .
  htmlspecialchars($firstName, ENT_QUOTES, 'UTF-8') . ' ' .
  htmlspecialchars($lastName, ENT_QUOTES, 'UTF-8') . '!';
?>
```

The *echo* statement is becoming quite sizable now, but it should still make

22. https://github.com/spbooks/phpmysql7/tree/PHP-GET-TwoVars

sense to you. Using a series of string concatenations (`.`), it outputs "Welcome to our website", followed by the value of `$firstName` (made safe for display using `htmlspecialchars`), a space, the value of `$lastName` (again, treated with `htmlspecialchars`), and finally an exclamation mark.

The result is pictured below.



2-9. The echoed welcome

This is all well and good, but we're still yet to achieve our goal of true user interaction, where the user can enter arbitrary information and have it processed by PHP. To continue with our example of a personalized welcome message, we'd like to invite the user to type their name and have it appear in the resulting page. To enable the user to type in a value, we'll need to use an HTML form.

Passing Variables in Forms

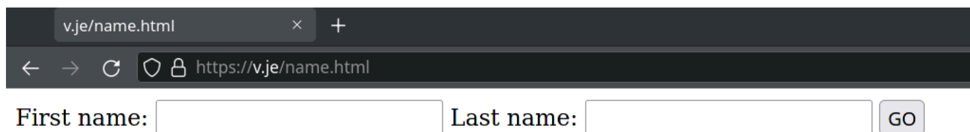
Remove the link code from `name.html` and replace it with this HTML code to create the form.

Example: PHP-GET-Form[23]

23. https://github.com/spbooks/phpmysql7/tree/PHP-GET-Form

```
<form action="name.php" method="get">
  <label for="firstname">First name:</label>
  <input type="text" name="firstname" id="firstname">

  <label for="lastname">Last name:</label>
  <input type="text" name="lastname" id="lastname">

  <input type="submit" value="GO">
</form>
```

The image below shows how the browser displays the form produced from this code.



2-10. The form as it appears in a browser

Styling with CSS

I've added some CSS to the form (available in `form.css` in the sample code) to make it look a little prettier. The CSS I've used is very generic, and can be used to display any form in the format label-input-line break. I'll be including this CSS file on any page that contains a form.

Since this is a book about PHP and MySQL, however, I won't go into detail about how the CSS works. Check out SitePoint's *HTML5 & CSS3 for the Real World*[24] for advice on styling your forms with CSS.

---

24. https://www.sitepoint.com/premium/books/html5-css3-for-the-real-world-2nd-edition

This form has the exact same effect as the second link we looked at in the "Passing Variables in Links" section above (with `firstname=Tom&lastname=Butler` in the query string), except that you can now enter whichever names you like. When you click the submit button (labeled **GO**), the browser will load `name.php` and add the variables and their values to the query string for you automatically. It retrieves the names of the variables from the name attributes of the `type="text"` inputs, and obtains the values from the text typed into the text fields by the user.

The `method` attribute of the form tag is used to tell the browser how to send the variables and their values along with the request. A value of `get` (as used in `name.html` above) causes them to be passed via the query string (and appear in PHP's `$_GET` array), but there's an alternative. It can be undesirable—or even technically unfeasible—to have the values appear in the query string. What if we included a `<textarea>` element in the form, to let the user enter a large amount of text? A URL whose query string contained several paragraphs of text would be ridiculously long, and would possibly exceed the maximum length for a URL in today's browsers. The alternative is for the browser to pass the information invisibly, behind the scenes.

Edit your `name.html` file once more. Modify the form method by setting it to `post`:

```
<form action="name.php" method="post">
  <label for="firstname">First name:</label>
  <input type="text" name="firstname" id="firstname">

  <label for="lastname">Last name:</label>
  <input type="text" name="lastname" id="lastname">

  <input type="submit" value="GO">
</form>
```

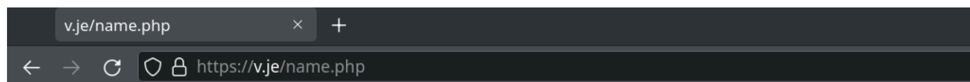
This new value for the method attribute instructs the browser to send the form variables invisibly as part of the page request, rather than embedding them in the query string of the URL.

As we're no longer sending the variables as part of the query string, they stop appearing in PHP's `$_GET` array. Instead, they're placed in another array reserved especially for "posted" form variables: `$_POST`. We must therefore modify `name.php` to retrieve the values from this new array.

Example: PHP-POST-Form[25]

```
<?php
$firstname = $_POST['firstname'];
$lastname = $_POST['lastname'];
echo 'Welcome to our website, ' .
  htmlspecialchars($firstname, ENT_QUOTES, 'UTF-8') . ' ' .
  htmlspecialchars($lastname, ENT_QUOTES, 'UTF-8') . '!';
?>
```

The image below shows what the resulting page looks like once this new form is submitted.



Welcome to our website, Tom Butler!

2-11. The resulting page once the form is submitted

The form is functionally identical to the previous one. The only difference is that the URL of the page that's loaded when the user clicks the **GO** button will be without a query string. On the one hand, this lets you include large values (or sensitive values such as passwords and credit card numbers) in the data that's submitted by the form without them appearing in the query string. On the other hand, if the user bookmarks the page that results from the form's submission, that bookmark will be useless, as it lacks the submitted values. This, incidentally, is the main reason why search engines use the query string

25. https://github.com/spbooks/phpmysql7/tree/PHP-POST-Form

to submit search terms. If you bookmark a search results page on Google, you can use that bookmark to perform the same search again later, because the search terms are contained in the URL.

GET or POST?

As a rule of thumb, you should only use `GET` forms if, when the form is submitted, nothing on the server *changes*—such as when you're requesting a list of search results. Because the search terms are in the URL, the user can bookmark the search results page and get back to it without having to type in the search term again. But if, after submitting the form, a file is deleted, a database is updated, or a record is inserted, you should use `POST`. The primary reason for this is that if a user bookmarks the page (or presses the back button in their browser) it won't trigger the form submission again and potentially create a duplicate record.

`POST` is also more secure. Anything sent via `GET` appears in the URL and is stored in the user's history and bookmarks—which are very insecure locations for sensitive data such as passwords and credit card details.

That covers the basics of using forms to produce rudimentary user interaction with PHP. We'll look at more advanced issues and techniques in later examples.

Hiding the Seams

You're now armed with a working knowledge of the basic syntax of the PHP programming language. You understand that you can take any HTML web page, rename it with a `.php` file name extension, and inject PHP code into it to generate page content on the fly. Not bad for a day's work!

Before we go any further, however, I want to stop and cast a critical eye over the examples we've discussed so far. Assuming your objective is to create

database-driven websites that hold up to professional standards, there are a few unsightly blemishes we need to clean up.

The techniques in the rest of this chapter will help advance your programming skills beyond the beginner level, giving them a certain professional polish. I'll rely on these techniques throughout the rest of this book to ensure that, no matter how simple the example, you can feel confident in the quality of the product you're delivering.

PHP Templates

In the simple examples we've seen so far, inserting PHP code directly into your HTML pages has been a reasonable approach. As the amount of PHP code that goes into generating your average page grows, however, maintaining this mixture of HTML and PHP code can become unmanageable.

Particularly if you work in a team of web designers who aren't especially PHP savvy, having large blocks of cryptic PHP code intermingled with the HTML is a recipe for disaster. It's far too easy for designers to accidentally modify the PHP code, causing errors they'll be unable to fix.

A much more robust approach is to separate out the bulk of your PHP code so that it resides in its own file, leaving the HTML largely unpolluted by PHP code.

The key to doing this is the PHP `include` statement. With an `include` statement, you can insert the contents of another file into your PHP code at the point of the statement. To show you how this works, let's rebuild the "count to ten" `for` loop example we looked at earlier.

Start by creating a new file, `count.php`, in your `public` directory. Open the file for editing and type in this code:

```
<?php
$output = '';
for ($count = 1; $count <= 10; $count++) {
```

```
    $output .= $count . ' ';
}

include 'count.html.php';
```

Yes, that's the *complete* code for this file. It contains no HTML code whatsoever. The `for` loop should be familiar to you by now, but let me point out the interesting parts of this code:

- Instead of `echo`ing out the numbers 1 to 10, this script will add these numbers to a variable named `$output`. At the start of this script, therefore, we set this variable to contain an empty string.
- The line `$output .= $count . ' ';` adds each number (followed by a space) to the end of the `$output` variable. The `.=` operator you see here is a shorthand way of adding a value to the end of an existing string variable, by combining the assignment and string concatenation operators into one. The longhand version of this line is `$output = $output . $count . ' ';`, but the `.=` operator saves you some typing.
- The `include` statement instructs PHP to execute the contents of the `count.html.php` file at this location. You can think of the `include` statement as a kind of *copy and paste*. You would get the same result by opening up `count.html.php`, copying the contents and pasting them into `count.php`, overwriting the `include` line.
- Finally, you might have noticed that the file doesn't end with a `?>` to match the opening `<?php`. You can put it in if you really want to, but it's not necessary. If a PHP file ends with PHP code, there's no need to indicate where that code ends; the end of the file does it for you. The big brains of the PHP world generally prefer to leave it off the end of files like this one that contain only PHP code.

Parentheses with Includes

Outside of this book, you'll sometimes see parentheses surrounding the filename in `include` code, as if `include` were a function like `date` or `htmlspecialchars`, which is far from the case. For example, instead of `include 'count.html.php';`, you might see `include('count.html.php');`. These parentheses, when used, only serve to complicate the filename expression, and are therefore avoided in this book. The same goes for `echo`, another popular one-liner.

Since the final line of our `count.php` file includes the `count.html.php` file, you should create that file next.

Example: PHP-Count-Template[26]

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Counting to Ten</title>
  </head>
  <body>
    <p>
      <?php echo $output; ?>
    </p>
  </body>
</html>
```

This file is almost entirely plain HTML, except for the one line that outputs the value of the `$output` variable. This is the same `$output` variable that was created by the `count.php` file.

What we've created here is a PHP template: an HTML page with only very small snippets of PHP code that insert dynamically generated values into an

26. https://github.com/spbooks/phpmysql7/tree/PHP-Count-Template

otherwise static HTML page. Rather than embedding the complex PHP code that generates those values in the page, we put the code to generate the values in a separate PHP script—`count.php` in this case.

Using PHP templates like this enables you to hand over your templates to front-end designers without worrying about what they might do to your PHP code. It also lets you focus on your PHP code without being distracted by the surrounding HTML code.

I like to name my PHP template files so that they end with `.html.php`. As far as your web server is concerned, though, these are still `.php` files; the `.html.php` suffix serves as a useful reminder that these files contain both HTML and PHP code.

To see how our pages work now, type `https://v.je/count.php` into your browser. You'll see the results of the `count.php` script displayed in our `count.html.php` template.

Security Concerns

One problem with separating out the HTML and PHP code into different files is that someone could potentially run the `.html.php` code without having had it included from a corresponding PHP file. This isn't a big problem, but anyone could visit `count.html.php` directly. If you type `https://v.je/count.html.php` into your web browser, instead of seeing the count from one to ten, you'll see an error message: `Notice: Undefined variable: output in /websites/default/public/count.html.php on line 9`.

It's better not to let people run code in a manner you're not expecting. Depending on what the page is doing, this might let them bypass security checks you have in place and view content they shouldn't have access to. For example, consider the following code:

```
if ($_POST['password'] == 'secret') {
  include 'protected.html.php';
```

```
}
```

Looking at this code, it appears that you need to submit a form and type `secret` in the password box to see the protected content in `protected.html.php`. However, if someone can navigate directly to `protected.html.php` and see the contents of the page, it makes the security check redundant. There are other potential security issues introduced by making all your files accessible via a URL. Avoiding security problems like these is easy. You can actually include files from a directory other than the `public` directory.

You may have wondered earlier why the development environment created a `public` directory inside the `default` directory and we then wrote all our files to the `public` directory. Well, this issue of security is the reason why. None of the files outside the `public` directory are accessible via a URL (by someone typing the file name into their web browser).

The `include` command can be tweaked to include files from *another directory*. In our case, that directory is going to be the `default` directory, which contains the `public` directory we've been writing our files to.

So the question is, when the include file is in a *different* directory, how does a PHP script find it? The most obvious method is to specify the location of the include file as an absolute path. Here's how this would look on a Windows server:

```
<?php include 'C:/Program Files/Apache Software Foundation/Apache2.2/
↪protected.html.php'; ?>
```

And here's how it would look using the Docker environment we're using:

```
<?php include '/websites/default/protected.html.php'; ?>
```

While this method will work, it's undesirable because it ties your site's code to your web server configuration. Because we're using Docker, deploying the

website would also use the same Docker environment, so this is not really an issue these days. However, you should ideally be able to drop your PHP-based website onto any PHP-enabled web server and just watch it run. That's impractical if your code refers to drives and directories that are specific to one particular server. Even if you *do* have the luxury of working on a single server, you'll be kicking yourself if you ever need to move your website to another drive/directory on that server.

A better method is to use a *relative* path—that is, the location of a file relative to the current file. When you use `include 'count.html.php'`, this is actually a *relative path*: `count.html.php` is being included from the same directory as the script that was executed.

To include a file from the *directory above*, you can use the following code:

```
include '../count.html.php';
```

The `../` bit tells PHP to look for the file in the directory *above* the directory of the current script. It will look for `count.html.php` in the `default` directory instead of the `public` directory.

Go ahead and move `count.html.php` up a level into the `default` directory and amend `count.php` to reference the new location.

Example: PHP-Count-Template-Secured[27]

```
<?php
$output = '';
for ($count = 1; $count <= 10; $count++) {
    $output .= $count . ' ';
}

include '../count.html.php';
```

If you run the code above, it will work. But there's a potential problem when

27. https://github.com/spbooks/phpmysql7/tree/PHP-Count-Template-Secured

you include files in this way. Relative paths are relative to the script that was run, not to each file.

That is, if you open up `default/count.html.php` and add the line `include 'count2.html.php';` you would expect `count2.html.php` to be included from the `default` directory. However, the path is relative to something called the *current working directory*, which, when you run a PHP script, is initially set to the directory that script is stored in. So running `include 'count2.html.php';` from `count.html.php` will actually try to load `count2.html.php` from the `public` directory!

The current working directory is set at the start of the script and applies to all the `include` statements, regardless of what file they are in. To make things even more confusing, it's possible to change the current working directory using the `chdir()` function.

Because of this, we can't rely on the following:

```
include '../count.html.php';
```

It will work, but if the directory is changed, or `count.php` itself is an include file, it may not have the result we're expecting.

To overcome this, we do actually need to use absolute paths. Luckily, PHP provides a constant called `__DIR__` (that's *two* underscores, before and after the word `DIR`) which will always contain the path that contains the *current file*.

For example, you could create a file called `dir.php` inside the `public` directory with the following code:

```
echo __DIR__;
```

This will display `/websites/default/public`, which is the full path to the directory containing `dir.php`. To read `count.html.php` from the directory

above `public`, it's possible to combine the `/../` operator and the `__DIR__` constant:

```
include __DIR__ . '/../count.html.php';
```

This will now include the file `/websites/default/public/../count.html`. That is, PHP will look in the `public` directory, then go up one level into `default` and include `count.html.php`.

This approach will work on any server, because `__DIR__` will differ depending on where the file is stored, and it doesn't depend on the changing *current working directory*. I'll be using this approach for including files throughout this book.

From now on, we'll only write files to the `public` directory that we actually want users to be able to access directly from their web browser. The `public` directory will contain any PHP scripts the user needs to access directly along with any images, JavaScript and CSS files required by the browser. Any files only referenced by an `include` statement will be placed outside the `public` directory so users can't access them directly.

As the book goes on, I'm going to introduce you to several different types of *include files*. To keep things organized, it's sensible to store different types of include files in different directories. We'll store template files (with a `.html.php` extension) inside a directory called `templates` inside the `default` folder. We can then reference them in an `include` statement using `include __DIR__ . '../templates/file.html.php';`.

Many Templates, One Controller

What's nice about using `include` statements to load your PHP template files is that you can have *multiple* `include` statements in a single PHP script, as well as have it display different templates under various circumstances!

A PHP script that responds to a browser request by selecting one of several

PHP templates to fill in and send back is commonly called a “controller”. A **controller** contains the logic that controls which template is sent to the browser.

Let’s revisit one more example from earlier in this chapter—the welcome form that prompts a visitor for a first and last name.

We’ll start with the PHP template for the form. For this, we can just reuse the `name.html` file we created earlier. Create a directory `templates` inside `default` if you haven’t already, and save a copy of `name.html` called `form.html.php` into this directory. The only code you need to change in this file is the action attribute of the form tag.

Example: PHP-Form-Controller[28]

```
<!doctype html>
<html>
  <head>
    <title>Enter your name</title>
    <link rel="stylesheet" href="form.css" />
    <meta charset="utf-8">
  </head>
  <body>
  <form action="" method="post">
    <label for="firstname">First name:</label>
    <input type="text" name="firstname" id="firstname">

    <label for="lastname">Last name:</label>
    <input type="text" name="lastname" id="lastname">

    <input type="submit" value="GO">
  </form>
  </body>
</html>
```

As you can see, we’re leaving the action attribute blank. This tells the browser to submit the form back to the same URL it received it from—in this case, the URL of the controller that included this template file.

---

28. https://github.com/spbooks/phpmysql7/tree/PHP-Form-Controller"

Let's take a look at the controller for this example. Create an `index.php` file inside the `public` directory that contains the following code:

```
<?php
if (!isset($_POST['firstname'])) {
    include  __DIR__ . '/../templates/form.html.php';
} else {
    $firstName = $_POST['firstname'];
    $lastName = $_POST['lastname'];

    if ($firstName == 'Tom' && $lastName == 'Butler') {
        $output = 'Welcome, oh glorious leader!';
    } else {
        $output = 'Welcome to our website, ' .
      htmlspecialchars($firstName, ENT_QUOTES, 'UTF-8') . ' ' .
      htmlspecialchars($lastName, ENT_QUOTES, 'UTF-8') . '!';
    }

    include  __DIR__ . '/../templates/welcome.html.php';
}
```

This code should look quite familiar at first glance. It's a lot like the `name.php` script we wrote earlier. Let me explain the differences:

- The controller's first task is to decide whether the current request is a submission of the form in `form.html.php` or not. You can do this by checking if the request contains a `firstname` variable. If it does, PHP will have stored the value in `$_POST['firstname']`.

  `isset` is a built-in PHP function that will tell you if a particular variable (or array element) has been assigned a value or not. If `$_POST['firstname']` has a value, `isset($_POST['firstname'])` will be true. If `$_POST['firstname']` is unset, `isset($_POST['firstname'])` will be false.

  For the sake of readability, I like to put the code that sends the form in my controller first. We need this `if` statement to check if `$_POST['firstname']` *isn't* set. To do this, we use the not operator (`!`). By putting this operator before the name of a function, you reverse the value

that function returns—from true to false, or from false to true.

Thus, if the request *doesn't* contain a `firstname` variable, then `!isset($_POST['firstname'])` will return true, and the body of the `if` statement will be executed.

- If the request isn't a form submission, the controller includes the `form.html.php` file to display the form.
- If the request *is* a form submission, the body of the `else` statement is executed instead.

  This code pulls the `firstname` and `lastname` variables out of the `$_POST` array, and then generates the appropriate welcome message for the name submitted.
- Instead of `echo`ing the welcome message, the controller stores the welcome message in a variable named `$output`.
- After generating the appropriate welcome message, the controller includes the `welcome.html.php` template, which will display that welcome message.

All that's left is to write the `welcome.html.php` file into the `templates` directory. Here it is:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Form Example</title>
  </head>
  <body>
    <p>
      <?php echo $output; ?>
    </p>
```

```
  </body>
</html>
```

That's it! Fire up your browser and point it at `https://v.je/index.php`. You'll be prompted for your name, and when you submit the form, you'll see the appropriate welcome message. The URL should stay the same throughout this process.

You'll have noticed I asked you to name the file `index.php` instead of `name.php` or similar. The reason I used `index.php` is because it has a special meaning. `index.php` is known as a **directory index**. If you don't specify a filename when you visit the URL in your browser, the server will look for a file named `index.php` and display that. Try typing just `https://v.je` into your browser and you'll see the index page.

Directory Index

Web servers can have different configurations and specify a different file to be the directory index. However, `index.php` will work on most web servers without any further configuration.

One of the benefits of maintaining the same URL throughout this process of prompting the user for a name and displaying the welcome message is that the user can bookmark the page at any time during this process and gain a sensible result. Whether it's the form page or the welcome message that's bookmarked, when the user returns, the form will be present once again. In the previous version of this example, where the welcome message had its own URL, returning to that URL without submitting the form would have generated a broken welcome message ("Welcome to our website, !"), or a PHP error message if, like ours, the server is running with error reporting enabled.

Sessions

In Chapter 11, where we discuss “sessions”, I show you how to remember the user’s name between visits.

Bring On the Database

In this chapter, we’ve seen the PHP server-side scripting language in action as we’ve explored all the basic language features: statements, variables, operators, comments, and control structures. The sample applications we’ve seen have been reasonably simple, but we’ve still taken the time to ensure they have attractive URLs, and that the HTML templates for the pages they generate are uncluttered by the PHP code that controls them.

As you may have begun to suspect, the real power of PHP is in its hundreds (even thousands) of built-in functions that let you access data in a MySQL database, send email, dynamically generate images, and even create Adobe Acrobat PDF files on the fly.

In Chapter 3, we’ll delve into the MySQL functions built into PHP, and then create a database of jokes. Then, in Chapter 4, we’ll see how to publish that joke database to the Web. These tasks will set the scene for the ultimate goal of this book: to create a complete content management system for your website in PHP and MySQL.

Chapter 3

Introducing MySQL

As I explained in the last chapter, PHP is a server-side scripting language that lets you insert instructions into your web pages that your web server software will execute before it sends those pages to browsers that request them. We've looked at a few basic examples, including generating random numbers and using forms to capture input from a user.

Now, that's all well and good, but it *really* gets interesting when a database is added to the mix. In this chapter, we'll learn what a database is, and how to work with your own databases using Structured Query Language (SQL).

An Introduction to Databases

A **database server** is a program that can store large amounts of information in an organized format that's easily accessible through programming languages like PHP. For example, you could tell PHP to look in the database for a list of jokes that you'd like to appear on your website.

In this example, the jokes would be stored entirely in the database. The advantage of this approach is twofold. First, instead of writing an HTML page for each joke, you could write a single PHP script designed to fetch any joke from the database and display it by generating an HTML page for it on the fly. Second, adding a joke to your website would be a simple matter of inserting the joke into the database. The PHP code would take care of the rest, automatically displaying the new joke along with the others when it fetched the list from the database.

Let's run with this example as we look at how data is stored in a database. A database is composed of one or more **tables**, each of which contains a list of **items**, or *things*. For our joke database, we'd probably start with a table called `joke` that would contain a list of jokes. Each table in a database has one or more **columns**, or **fields**. Each column holds a certain piece of information about each item in the table. In our example, our `joke` table might have one column for the text of the jokes, and another for the dates on which the jokes were added to the database. Each joke stored in this way would be said to be a **row** or **entry** in the table. These rows and columns form a table that's represented in the image below.

column column column

id	joketext	jokedate
1	Why did the chicken ...	2012-04-01
2	Knock-knock! Who's ...	2012-04-01

row
row

3-1. A typical database table containing a list of jokes

If you've ever created a spreadsheet, this will look familiar to you. A database table is similar, in that data is stored in rows and columns. The only difference is that, unlike Excel—where the columns are named `A`, `B`, `C`, and so on—when you create a database table you choose a name for each column.

Notice that, in addition to columns for the joke text (`joketext`) and the date of the joke (`jokedate`), there's also a column named `id`. As a matter of good design, a database table should always provide a means by which each row can be identified uniquely. Since it's possible that two identical jokes could be entered on the same date, we can't rely on the `joketext` and `jokedate` columns to tell all the jokes apart. The function of the `id` column, therefore, is to assign a unique number to each joke, so that we have an easy way to refer to them and to keep track of which joke is which. We'll take a closer look at database design issues like this in Chapter 5.

Combining Columns as a Unique Identifier

It's also possible to use a combination of columns as a unique identifier—such as manufacturer name and product name together. One manufacturer will likely have more than one product, and two manufacturers may have products with the same name. By combining the two names, it's possible to uniquely identify each product.

To review, the table pictured above is a three-column table with two rows (or entries). Each row in the table contains three fields, one for each column in the table: the joke's ID, its text, and the date of the joke. With this basic terminology under your belt, you're ready to dive into creating a database yourself.

MySQL

The title of this book is *PHP and MySQL: Novice to Ninja*—where *MySQL* refers to the database we're using. However, if you peruse the `docker-compose.yml` file you downloaded as part of setting up the Docker environment, you'll notice that it actually installs a database called *MariaDB*.

In 2009, MySQL was bought by Oracle, a massive software company. Unsure about MySQL's future, Michael Widenius—one of the founders of the original MySQL database—decided to fork MySQL to create a new database called MariaDB. (**Forking** means creating a new project from an existing project, using the original project's source code as a basis.) As well as not being controlled by Oracle, MariaDB has some performance advantages over MySQL, which makes it a great choice.

MariaDB is a drop-in replacement for MySQL, and any tutorials you follow that teach you how to use MySQL will work exactly the same way with MariaDB. As a PHP developer, you won't notice any difference between the two, and it's possible to swap one out for the other.

MySQL and MariaDB

Over time, the differences in MySQL and MariaDB have grown slightly, but the fundamental commands, tools and techniques you'll use will be the same. There are some minor differences when it comes to some of the more advanced features supported by the two databases.

Despite this happening over ten years ago, most developers and package

management systems use the two interchangeably. On Arch Linux, for example, if you install the `mysql` package, it actually installs MariaDB instead, and the XAMPP package I discussed in the first chapter installs MariaDB instead of MySQL.

Despite this, if you start developing PHP code using MariaDB, you'll see constant references to MySQL, not MariaDB. That's because client software (anything that connects to the server to interact with the database) doesn't know if it's connecting to MySQL or MariaDB. To this client software, MySQL is a *protocol*. In the same way you can plug in a keyboard or mouse via a USB port on your computer, you can connect to a MySQL server or MariaDB server via the MySQL protocol.

So when you hear a developer talk about "adding records to a MySQL database", they're often referring to using the *MySQL protocol* to manage a database, regardless of the specific implementation being used.

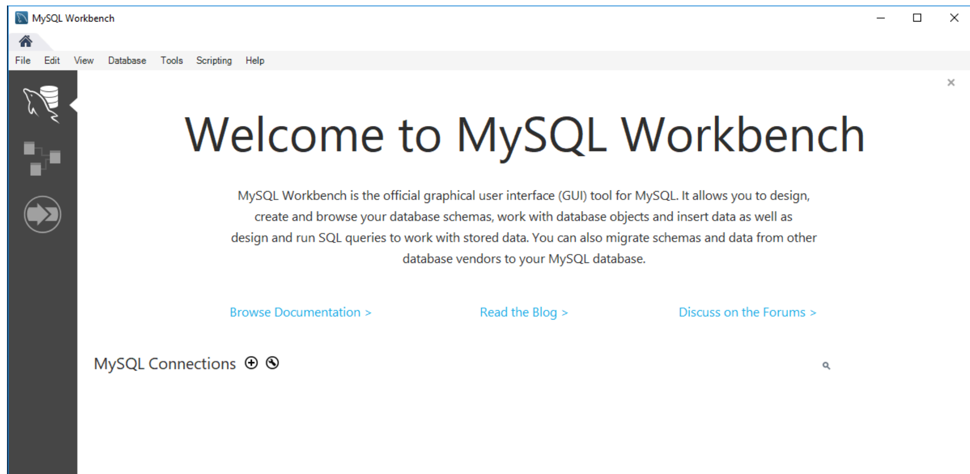
You'll find developers frequently using the term "MySQL" even though they're actually using MariaDB. For consistency's sake, I'm going to do the same in this book. Rather than refer to the server as MariaDB and use the term MySQL when discussing connecting from PHP, I'll just use MySQL throughout.

Using MySQL Workbench to Run SQL Queries

Just as a web server is designed to respond to requests from a **client** (a web browser), a database server responds to requests from **client programs**. Later in this book, we'll write our own MySQL client programs in the form of PHP scripts, but for now we can use a client program written by the same people who write MySQL: MySQL Workbench. You can download MySQL Workbench for free from https://www.mysql.com/products/workbench/.

There are many different MySQL clients available to use, and earlier editions of this book used phpMyAdmin, a web-based MySQL client that has many of the same features. However, it's not as easy to use as MySQL Workbench, and can often be very slow.

Once you've downloaded and installed MySQL Workbench, open it up, and you should see a screen like the one shown below.



3-2. If you can see this, you have MySQL Workbench running

Before you can add any data to your database, you need to connect to it. A MariaDB server is running in the Docker Environment you downloaded in Chapter 1, and you can connect to it using a MySQL client such as MySQL Workbench.

Connecting to the database requires three pieces of information:

- a server address
- a username
- a password

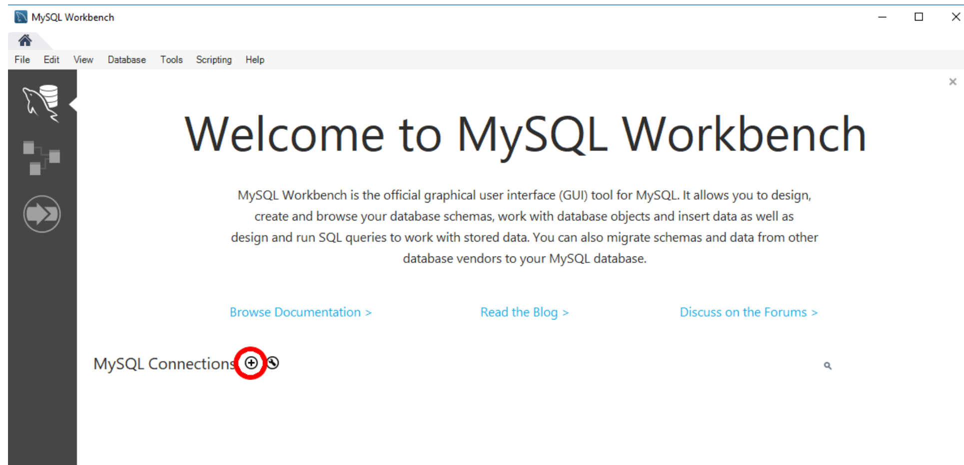
For the Docker environment we're using, the information is:

- **Server**: `v.je`
- **Username**: `v.je`
- **Password**: `v.je`

You'll notice that the server name is identical to the URL you've been connecting to in your web browser to view your PHP scripts. The Docker

environment is running both the web server and the database server, so you only need to remember a single address.

To connect to a database in MySQL Workbench, press the **+** button next to the "MySQL Connections" label in the centre of the window. (Admittedly, it isn't very clearly labeled, and its purpose isn't very clear, but never mind!)



3-3. Add a connection

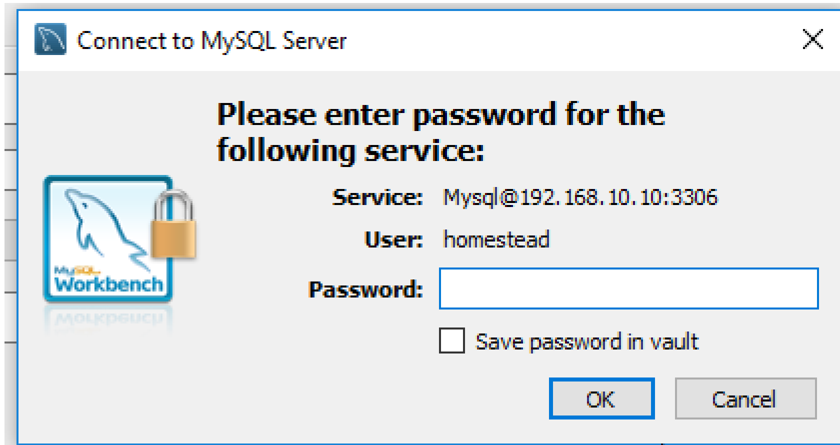
When you press the **+** button, you'll see a new window.

3-4. Add a connection

Enter the server address and username. You'll also need to give your connection a name. I've called it `v.je`, but you can call it whatever you like. This is just a name it's listed as for future reference in MySQL Workbench.

Once you've entered the username and server, you can try connecting to the database by pressing the **Test Connection** button at the bottom of the window.

You should get a password prompt box.



3-5. Password prompt

If you don't, follow these steps:

1. Double-check that your environment is running. (If you've rebooted your PC since setting it up, you may need to run `docker-compose up` again in the project's folder to get it running!)
2. Make sure the username, password and server address are correct.

Up versus Start

You can probably use the command `docker-compose start` here. `start` will start any existing containers, while `up` will create them if they don't exist and then start them. Depending on what you've done on your computer between the chapters of this book, the containers (and other things like network connections) created by Docker will be recreated if needed. If you've moved between computers, performed a system restore or certain software updates, it's possible the containers may need to be created again.

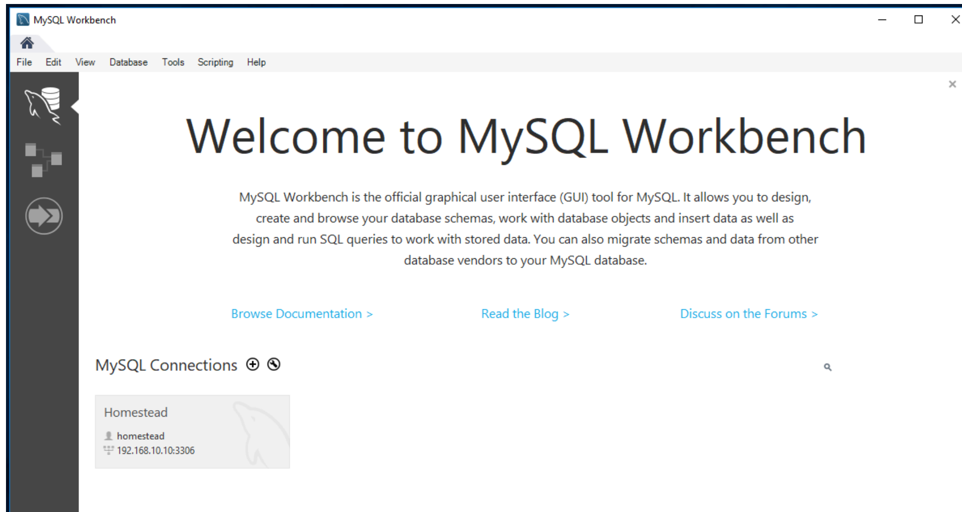
As such, `up` will work regardless of changes on your system, while `start` may or may not work depending on what's happened since last time you ran it.

Case Sensitivity

Usernames and passwords are case-sensitive, so make sure you type them both in lowercase!

Enter the password `v.je` into the box and tick the box that says "Save password". By checking the box, you won't have to enter the password each time you connect. Then press **OK**.

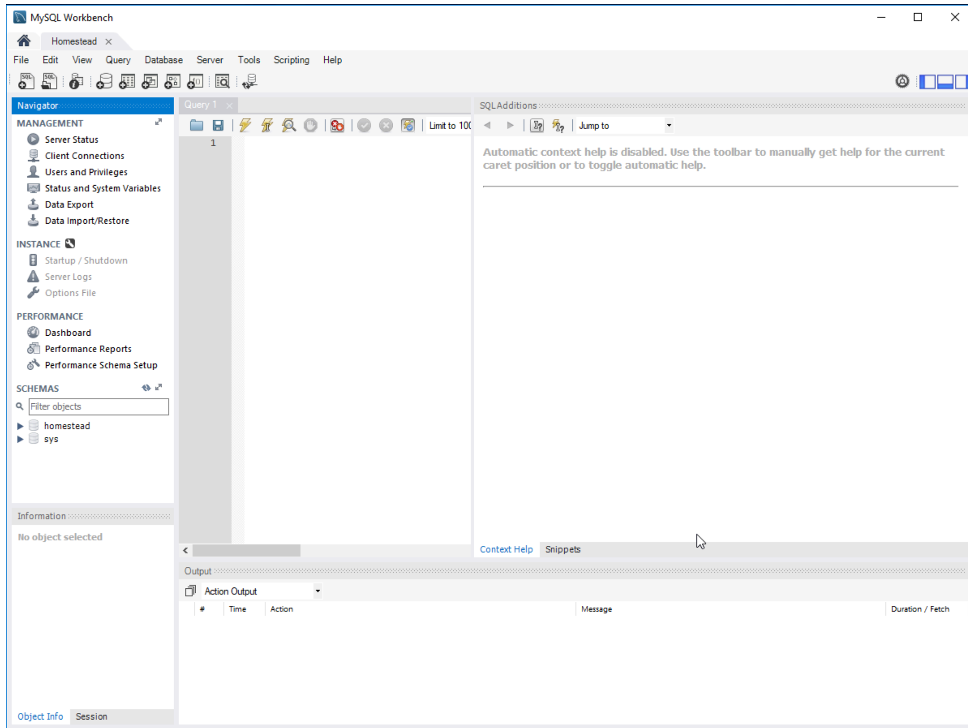
If the password was entered correctly, you'll see a message telling you the connection was successful. Press **OK** in the "Set up new connection" window and you'll see a box appear in the main MySQL window with some of the information you entered.



3-6. The main MySQL window showing information that was entered

Now that the connection is set up, it will be there each time you open MySQL workbench. You won't need to add the connection each time.

You're finally ready to actually connect to the database. To do this, simply double-click on the newly created box representing your connection and you'll be presented with a different screen.



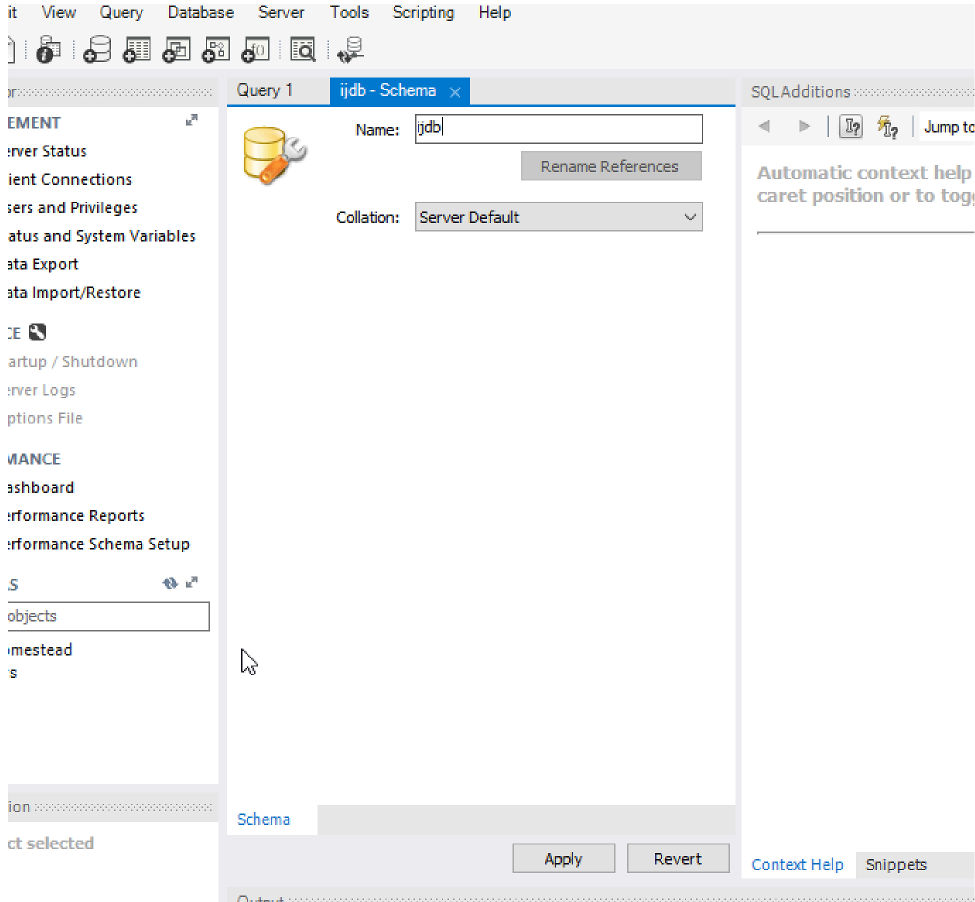
3-7. Initial screen

This looks a little daunting at first, as there are lots of different buttons and panels all representing different things. Down the left-hand side is a menu with lots of different options. The only one you need to worry about is the bottom section titled “SCHEMAS”.

Schema is just a fancy word for “database”. MySQL is a **database server**. In practical terms, this means that it can host lots of different databases, similarly to how a web server can host lots of different websites.

Creating a Database

Before you can add any information to a database, you need to create one. To create a database, right-click in the SCHEMAS panel and select **Create schema**. This gives you a window with several options, but you only need to enter one: the schema name.



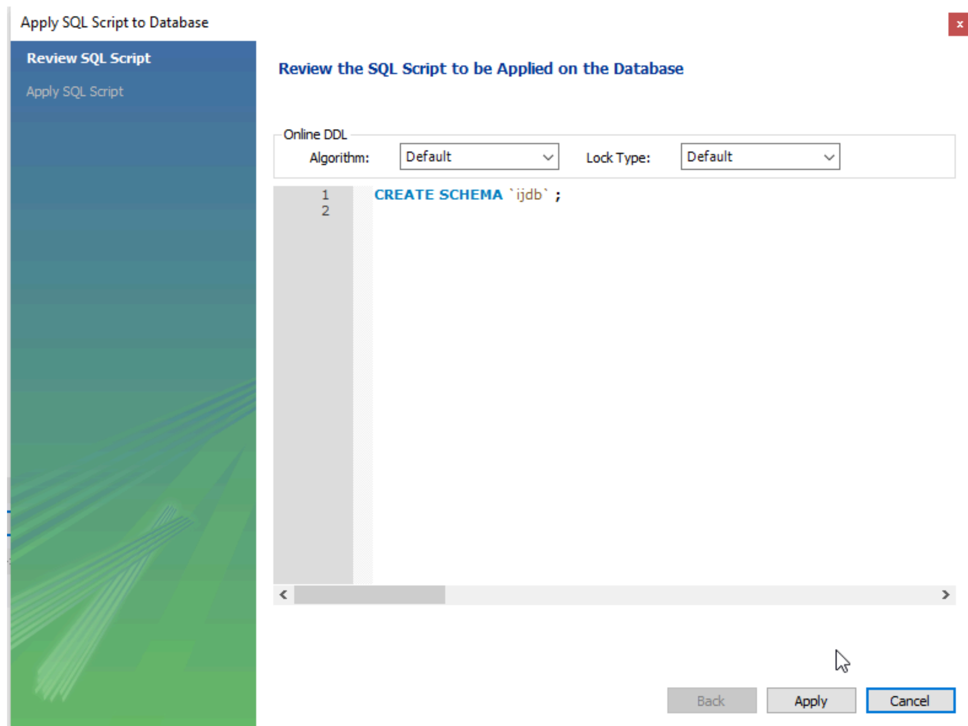
3-8. Creating a schema

I chose to name the database `ijdb`, for Internet Joke Database (with a tip of the hat to the Internet Movie Database[1]), because that fits with the example I gave at the beginning of this chapter: a website that displays a database of jokes. Feel free to give the database any name you like, though. (You'll need to type it out frequently as you progress through this book, so don't pick anything too complicated!)

Once you've typed a name, you can safely leave the other options at their default values and press **Apply**. When you do this, MySQL Workbench will ask you to confirm your action. (Get used to these dialogs. MySQL Workbench

[1] http://www.imdb.com

insists on confirmation for almost everything you do!) Press **Apply** again on the screen shown below.



3-9. Confirming the schema creation

Once you've pressed **Apply**, you'll need to press **Finish** on the next screen. This is one of the annoying things about MySQL Workbench: it forces you to confirm and then **Finish** every action. However, it's better than the alternative, as we'll see shortly!

In the screenshot above, you'll see a white panel with the words `CREATE SCHEMA `ijdb``. This is an SQL Query, and you'll see a lot more of these throughout this book. You could have typed out this command yourself and run it, avoiding the GUI and saving yourself going through MySQL Workbench's confirmation dialogs. And for a command as simple as `CREATE SCHEMA `ijdb``, the GUI is probably overkill. However, as you'll see shortly, not all of the commands are this simple, and it's a lot easier to use MySQL

Workbench's GUI for some of the more complex queries.

If you want to be able to delete databases (and this is probably a good ability to have, given the amount of experimentation I'm going to encourage you to do in this book), MySQL Workbench makes this easy. In the SCHEMAS panel in the main window, right-click on the schema you want to delete and select **DROP Schema**. MySQL uses the word `DROP` for deleting things. (Somewhat inconsistently, **Delete** is also used for some things!)

Structured Query Language

Like the `CREATE SCHEMA` command we just saw, the commands we'll use to direct MySQL throughout the rest of this book are part of a standard called **Structured Query Language**, or **SQL** (pronounced as either "sequel" or "ess-cue-ell", so take your pick). Commands in SQL are also referred to as **queries**, and I'll use these two terms interchangeably.

SQL is the standard language for interacting with most databases, so even if you move from MySQL to a database like Microsoft SQL Server in the future, you'll find that the majority of commands are identical. It's important that you understand the distinction between SQL and MySQL. MySQL is database server software that you're using—and MariaDB, which you're using, follows the same standards. SQL is the language that you use to interact with that database.

Most of these commands can be generated by MySQL Workbench, and that's what we'll use to create the structure of our database. However, you'll need to learn some commands, as you'll be executing them from your PHP scripts rather than MySQL Workbench!

Dive Deeper into SQL

In this book, I'll teach you the essentials of SQL that every PHP developer needs to know. If you decide to make a career out of building database-driven websites, it pays to know some of the more advanced details of SQL, especially when it comes to making your sites run as quickly and smoothly as possible. To dive deeper into SQL, I highly recommend the book *Simply SQL*[2], by Rudy Limeback. It's also worth checking out *Jump Start MySQL*[3], by Timothy Boronczyk.

Case Sensitivity and Convention

Most MySQL commands are not case-sensitive, which means you can type `CREATE DATABASE`, `create database`, or even `CrEaTe DaTaBaSe`, and it will know what you mean. Database names and table names, however, are case-sensitive when the MySQL server is running on an operating system with a case-sensitive file system (such as Linux, macOS, or when running inside Docker).

Additionally, table, column, and other names must be spelled exactly the same when they're used more than once in the same query.

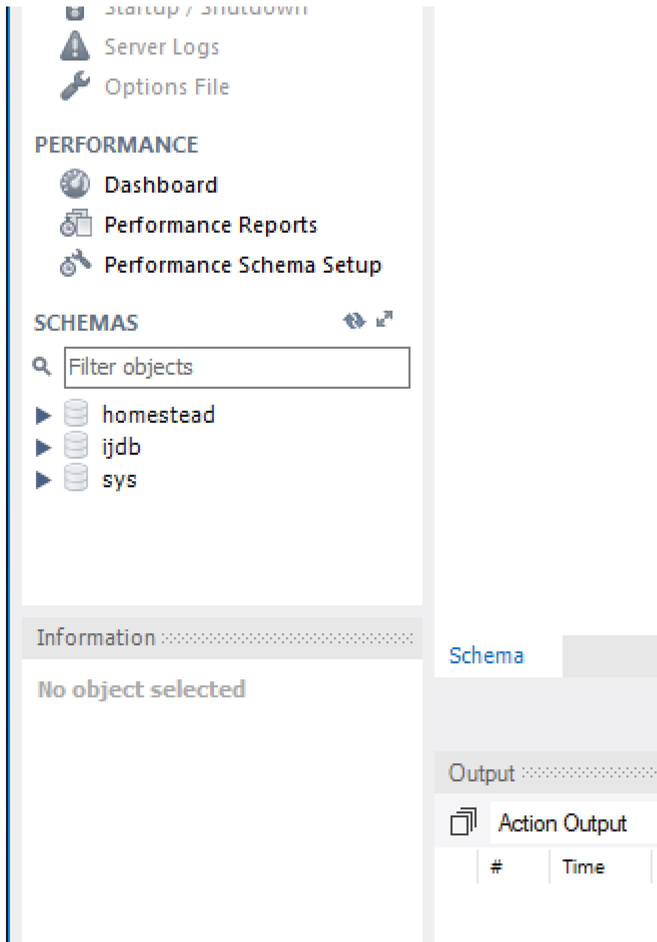
For consistency, this book will respect the accepted convention of typing database commands in all capitals, and database entities (databases, tables, columns, and so on) in all lowercase.

This also makes it easier for people (like you!) to read the queries. MySQL doesn't care, but you'll be able to identify a command quickly and easily because it's in capitals, and a reference to a table, column or database because it's in lowercase.

---

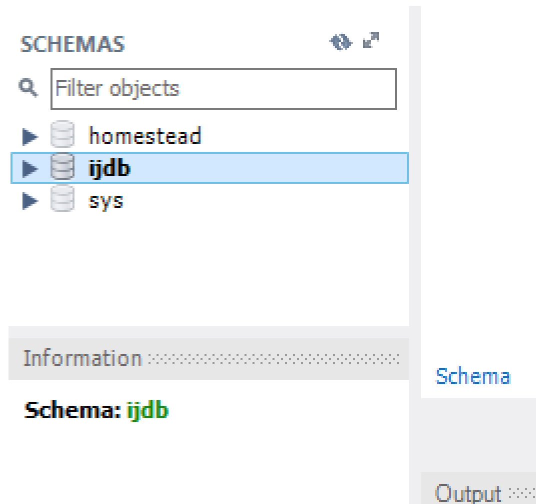
2. https://www.sitepoint.com/premium/books/simply-sql/
3. https://www.sitepoint.com/premium/books/jump-start-mysql/

Once your database has been created, it will appear in the SCHEMAS list on the left-hand side.



3-10. The Internet Joke Database schema

Now that you have a database, you need to tell MySQL Workbench that you want to use it. To do this, simply double-click the newly created schema and its name will go bold. You can only have one schema selected at a time, and you need to tell MySQL Workbench which you'd like to use.

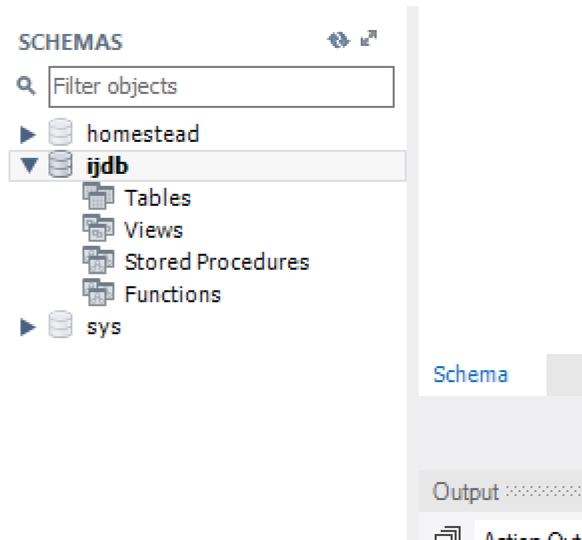


3-11. The ijdb schema selected

You're now ready to use your database. Since a database is empty until you add tables to it, our first order of business is to create a table that will hold your jokes. (Now might be a good time to think of some!)

Creating a Table

If you expand your newly created `ijdb` schema by pressing the arrow next to the name, you'll see a few entries.



3-12. The ijdb schema expanded

The only one we're concerned with for the purposes of this book is the Tables entry. Because your schema has just been created, it doesn't have any tables.

A table describes the format of your data. You'll need to know the *structure* of the data you'd like to store. Before creating a table, you need to think about exactly what you want to store. For the jokes example, we want to store these pieces of information:

- the text of the joke
- the date it was added

Along with the text and date, we'll also need some way to identify each joke. To do this, we'll give each joke a unique ID.

Each piece of information is placed in a field in the table, and each field has a "type". **Types** can be used to store data in different formats like numbers, text and dates.

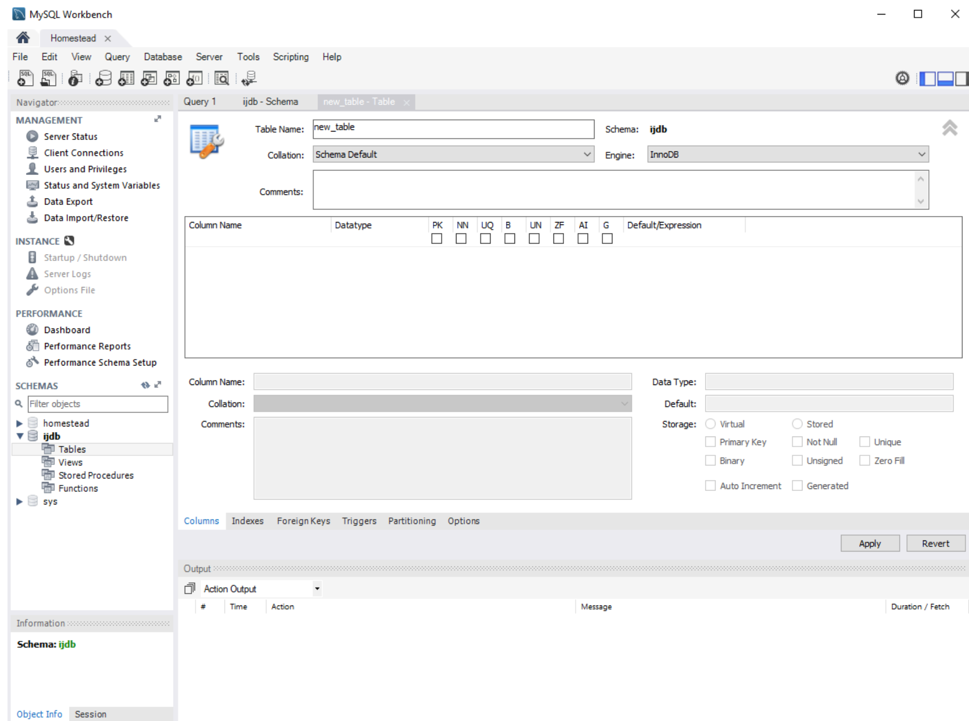
There are three main kinds of types that you'll encounter:

- numbers, for storing numeric values
- text, for storing strings
- dates/times, for storing timestamps

There are lots of column types in MySQL, but you only really need an understanding of three for most purposes!

To create a table using MySQL Workbench, expand the database in the SCHEMAS list, then right-click on the Tables entry and select **Create Table**.

The middle panel of the window will change to show you something like what's pictured below.



3-13. MySQL Workbench New Table window

Every table is given a name to identify it and a series of columns. Firstly, enter the table's name as "joke" and add the following columns in the column list:

- `id` , which will act as a unique identifier for each joke so we can retrieve it later
- `joketext` , which will store the text of the joke
- `jokedate` , which will store the date the joke was added

3-14. Creating the joke table

You'll notice there's a second column called Datatype. Each column in a database table must be assigned a type. The three types we will need are:

- `INT` , meaning "integer" for the numeric `jokeid`
- `TEXT` , to store some text for the joke
- `DATE` , to store the date the joke was published

This helps to keep your data organized, and allows you to compare the values within a column in powerful ways, as we'll see later.

If we were to stop setting up the table at this point, you could start adding records to the table (and I'll show you how to do that very shortly!). However, you'd have to provide all three pieces of information: the joke ID, the joke text, and the joke date. This means that, to add the next joke, you'd need to keep

track of how many were in there in order to assign the next ID.

This sounds like extra work, but fortunately MySQL Workbench provides a convenient way of avoiding it. Along with the name of the column and data type it stores, you'll notice there's a series of checkboxes for each field in the table.

There are three we're interested in here for our ID field:

- `PK`. This means "primary key". Ticking this box specifies that this column is to act as a unique identifier for the entries in the table, so all values in this column must be unique, ensuring that no two jokes will share the same ID.
- `NN`. This stands for "Not Null", and means that when a record is added, a value must be placed in the field. For our ID column, ticking this box tells MySQL not to accept jokes that don't have an ID.
- `AI`. This is the clever bit that will save us work. No, "AI" is not "Artificial Intelligence", some kind of computer-brain doing our work for us. In this case, it stands for "Auto Increment", and by checking this box (and it's only allowed on `INT` fields!), whenever a record (in our case, a joke) is added to the table, it will automatically be assigned the next available ID. This is a real time saver and a feature worth remembering.

Your table should now look like the one pictured below.

3-15. The joke table complete

Press the **Apply** button, and the joke table will be created. You'll see the following query appear in the window:

```
CREATE TABLE `ijdb`.`joke` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `joketext` TEXT NULL,
  `jokedate` DATE NULL,
  PRIMARY KEY (`id`));
```

You'll notice a lot of the same information has been repeated that we entered into the GUI. The GUI just generates this code for us, which is a much quicker and easier way of creating tables than remembering all of the syntax and vocabulary needed to write the query yourself.

As a developer, you don't need to create tables often. But you *will* need to interact with them—adding and removing records and retrieving them from the database—so it's worth spending time learning how to write queries to do this. For creating tables, however, it's usually a lot quicker and easier to use the MySQL Workbench GUI, because once a table has been created, you won't need to write another create table statement.

We need to look at just one more task: deleting a table. This task is frighteningly easy, so be careful! If you delete a table, you can't get it back.

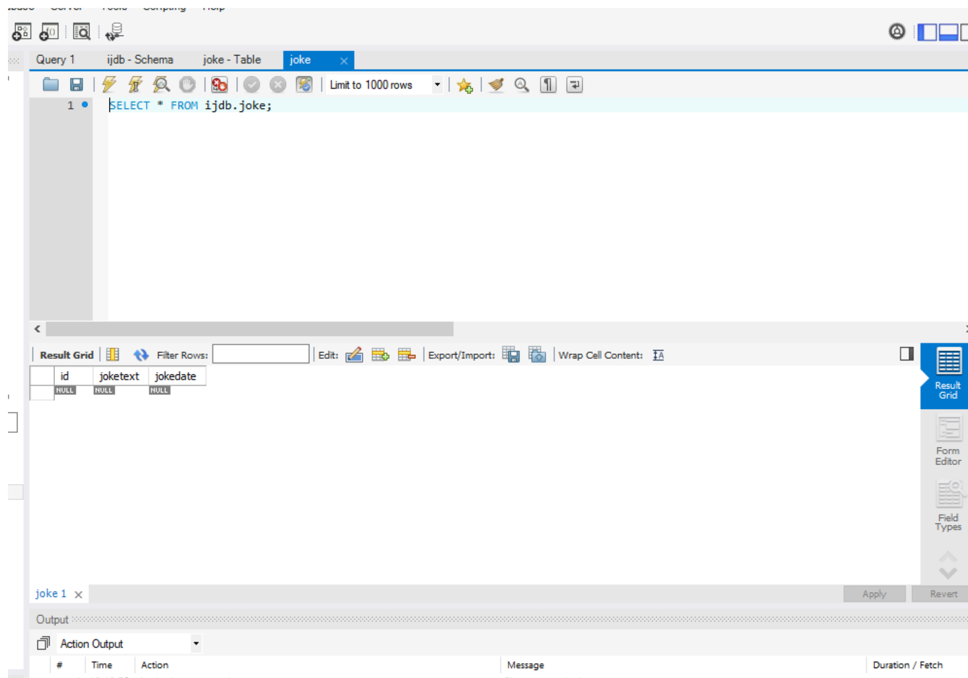
In the SCHEMAS list, right-click on the table you want to delete and select **Drop Table**. *Don't* run this command with your joke table unless you actually do want to be rid of it. If you really want to try it, be prepared to recreate your joke table from scratch. When you delete a table, the table is removed permanently, along with any data stored inside it. *There's no way to recover the data after the table has been dropped*, so be very careful when using this command!

Adding Data

Now that the table has been created, it's time to add some data to it. Although this can be done using MySQL Workbench's GUI, this time we're going to write the query ourselves. Eventually, we'll need to be able to write our own database queries directly from PHP, so it's good to get some practice writing them.

To run a query, you need to open up a query window. The simplest way to do this is to expand your database in the SCHEMAS list. Expand the Tables entry, and you'll see the `joke` table that you just created. Right-click on the table and click on the topmost option "Select Rows - Limit 1000".

This will give you a slightly different screen that's split into two panels horizontally.



3-16. A new query panel

The top half is a text box into which you can type commands to ask your database server questions or make it perform tasks. The bottom half is the result of that query. You'll see there's already a query in the top panel:

```
SELECT * FROM `ijdb`.`joke`;
```

We'll come back to what this means shortly. Along with this query in the top panel, there's a list of rows in the bottom panel—or rather there would be, if there were anything in the table! Because the table was just created, it's currently empty. Before you can view the contents of the table, you need to add some records.

All that's left is to put some jokes into the database. The command that inserts data into a database is called, appropriately enough, `INSERT`. This command can take two basic forms:

```
INSERT INTO tableName SET
  column1Name = column1Value,
  column2Name = column2Value,
  …
```

```
INSERT INTO tableName
  (column1Name, column2Name, …)
  VALUES (column1Value, column2Value, …)
```

So, to add a joke to our table, we can use either of these commands:

```
INSERT INTO joke SET
joketext = "A programmer was found dead in the shower. The instructions read:
↪lather, rinse, repeat.",
jokedate = "2021-10-29"
```

```
INSERT INTO joke
(joketext, jokedate) VALUES (
"A programmer was found dead in the shower. The instructions read: lather, rinse,
↪repeat.",
"2021-10-29")
```

Note that the order of the column/value pairs isn't important, but pairing the right values with the right columns, position-wise, is. If the first column mentioned in the first set of parentheses is `joketext`, then the first entry in the `VALUES` list must be the text that's going to be placed in the `joketext` column. The second column name in the first parentheses gets its values from the same position in the `VALUES` list. Otherwise, the order of the columns isn't important. Go ahead and swap the order of the column and value pairs and try the query.

As you typed the query, you'll have noticed that we used double quotes (`"`) to mark where the text of the joke started and ended. A piece of text enclosed in quotes this way is called a **text string**, and this is how you represent most data values in SQL. For instance, the dates are typed as text strings, too, in the form `"YYYY-MM-DD"`.

If you prefer, you can type text strings surrounded with single quotes (`'`) instead of double quotes:

```
INSERT INTO joke SET
joketext = '',
jokedate = '2021-10-29'
```

You might be wondering what happens when there are quotes used within the joke's text. Well, if the text contains single quotes, surround it with double quotes. Conversely, if the text contains double quotes, surround it with single quotes.

If the text you want to include in your query contains both single *and* double quotes, you'll have to escape the conflicting characters within your text string. You escape a character in SQL by adding a backslash (`\`) immediately before it (which, conveniently, is the same as in PHP). This tells MySQL to ignore any "special meaning" this character might have. In the case of single or double quotes, it tells MySQL not to interpret the character as the end of the text string.

To make this as clear as possible, here's an example of an `INSERT` command for a joke containing single quotes, even though single quotes have been used to mark the string:

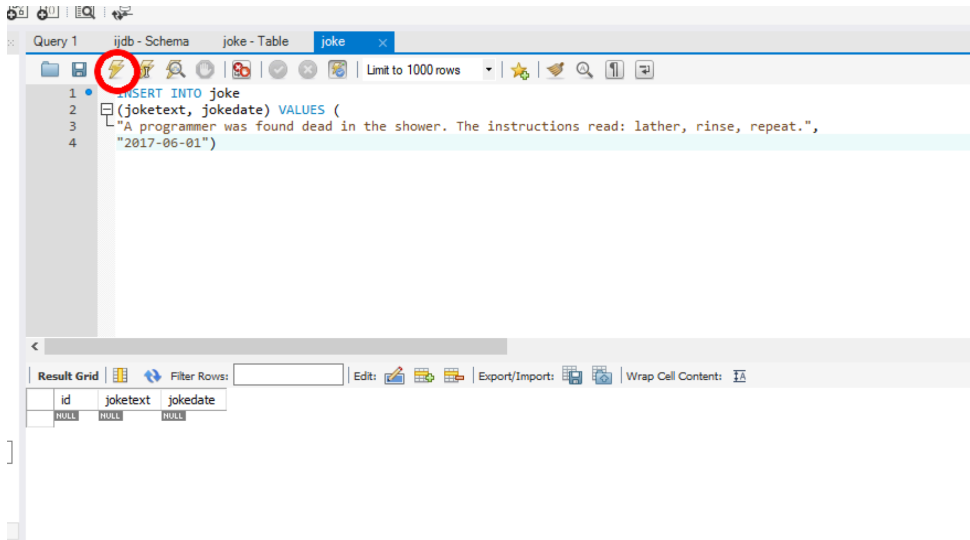
```
INSERT INTO joke
(joketext, jokedate) VALUES (
'!false - it\'s funny because it\'s true',
"2021-10-29")
```

As you can see, I've marked the start and end of the text string for the joke text using single quotes. I've therefore had to escape the two single quotes (the apostrophes) within the string by putting backslashes before them. MySQL would see these backslashes and know to treat the single quotes as characters within the string, rather than end-of-string markers.

If you're especially clever, you might now be wondering how to include actual

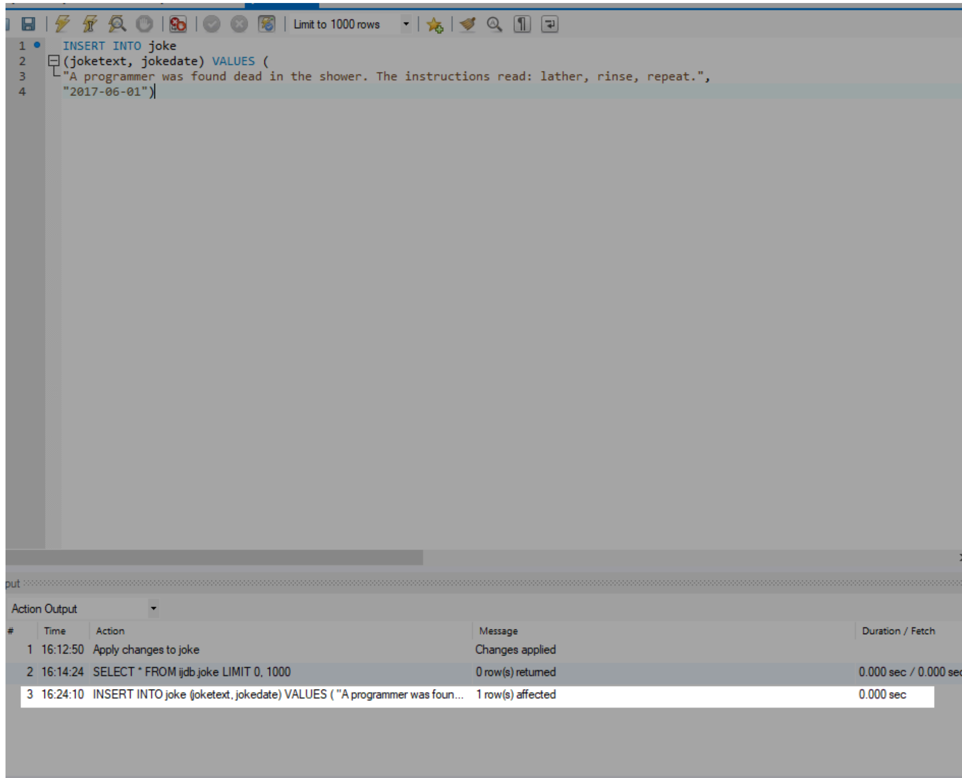
backslashes in SQL text strings. The answer is to type a double-backslash (`\\`), which MySQL will treat as a single backslash in the string of text.

Write your insert query into the top text box in MySQL Workbench and press the yellow lightning bolt icon above it to execute the query.



3-17. Executing an INSERT query using MySQL Workbench

When the query executes, a panel will appear at the bottom of the screen telling you if the query was executed successfully.



3-18. Checking a query has executed successfully

If you get an error and the query isn't successful, take a look at the error message. It should give you a hint where to look. Double-check your syntax, and check your quotes and parentheses are in the right place.

Displaying the Bottom Panel on Low-res Screens

If you have a lower screen resolution than it's expecting, MySQL Workbench hides the bottom panel. To display it, hover your mouse just below the scroll bar at the bottom of the window and you'll get a resize cursor. You can then drag the panel into view.

Add both the jokes (and any others you can think of!) to the database using `INSERT` queries. Now that you know how to add entries to a table, let's see how we can view those entries.

A Word of Warning

You'll have noticed something slightly peculiar about the queries that have been generated by MySQL Workbench. We don't get a query that looks like this:

```
SELECT * FROM joke
```

Instead, this is the query that's generated:

```
SELECT * FROM `joke`
```

There are strange quotes around `` `joke` ``. Those aren't actually quotes, or even apostrophes like we've been using to designate strings. They're **backticks**.

This is a safety precaution. There are lots of words in SQL that have meaning to the language. You've seen a few already: `SELECT`, `FROM` and `INSERT`. But there are hundreds of others, known as **reserved words**. Imagine if you called your table `SELECT`. The query you would need to run would look like this:

```
SELECT * FROM SELECT
```

Unfortunately, this can cause MySQL to get a little confused. It may see `SELECT` as a command rather than as a table name. What's worse, `date` is one of these words, and it's not improbable that you might think to create a column in one of your tables called `date`. What would you expect to happen when the following query runs?

```
INSERT INTO joke
(joketext, date) VALUES (
'!false - it\'s funny because it\'s true',
"2021-04-01")
```

Because the word `date` already has meaning in SQL, it may not be seen as a

column name but as part of the query, like `VALUES` or `INTO`.

MySQL is *usually* good at guessing whether you're referring to a table/column name or a command it needs to follow, but there are times when it isn't able to make that distinction. To avoid this kind of confusion, it's good practice to surround all table and column names with backticks. The backticks tell MySQL to treat the string as a *name* rather than an *instruction*. It's good to get into the habit of doing this from the very start, as it avoids issues later on that often aren't immediately obvious.

From now on, I'll surround all table, schema and column names with backticks. This will also help you—as a programmer—to distinguish between commands and column names. For instance, the `INSERT` query above would be written like this:

```
INSERT INTO `joke`
(`joketext`, `date`) VALUES (
'!false - it\'s funny because it\'s true',
"2021-04-01")
```

Where's the Darned Backtick Key?

On many English-language keyboard layouts, the backtick key is the one to the immediate left of the numeric `1` key and below `Esc`. Its location may differ on non-English keyboards and/or on various devices such as laptops and tablets.

Viewing Stored Data

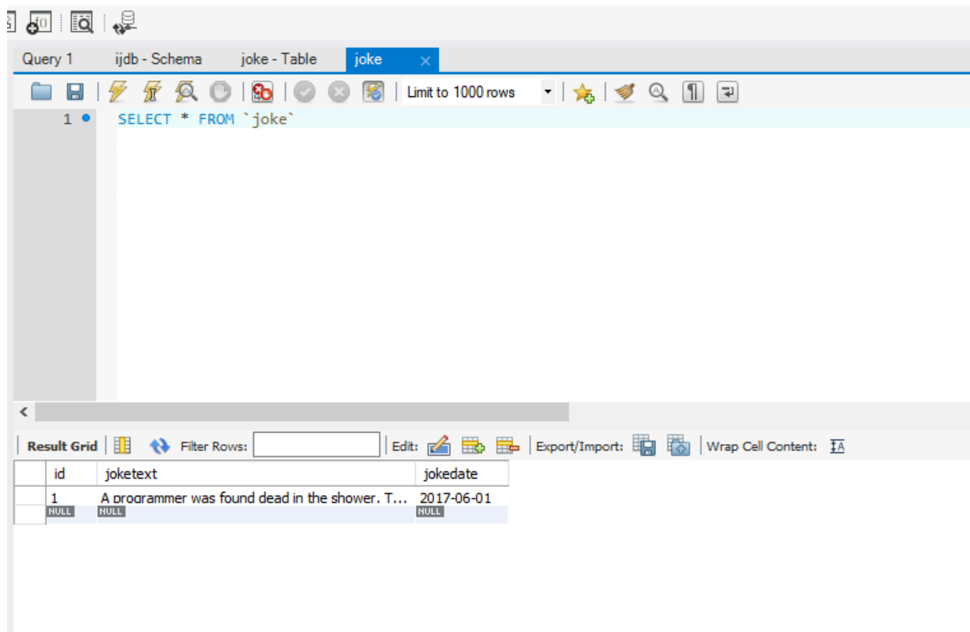
The command that we use to view data stored in database tables is `SELECT`. You saw an example `SELECT` query generated for you by MySQL Workbench earlier. The `SELECT` query is the most complicated command in SQL. The reason for this complexity is that the chief strength of a database is its flexibility in data retrieval. At this early point in our experience with databases,

we need only focus on fairly simple lists of results, so let's consider the simpler forms of the `SELECT` command here.

This command will list everything that's stored in the `joke` table:

```
SELECT * FROM `joke`
```

This command says "select everything from joke", with the `*` meaning "all columns". By default, a `SELECT` query will return every record in the table. If you try this command, your results will resemble the image below.



3-19. MySQL Workbench results

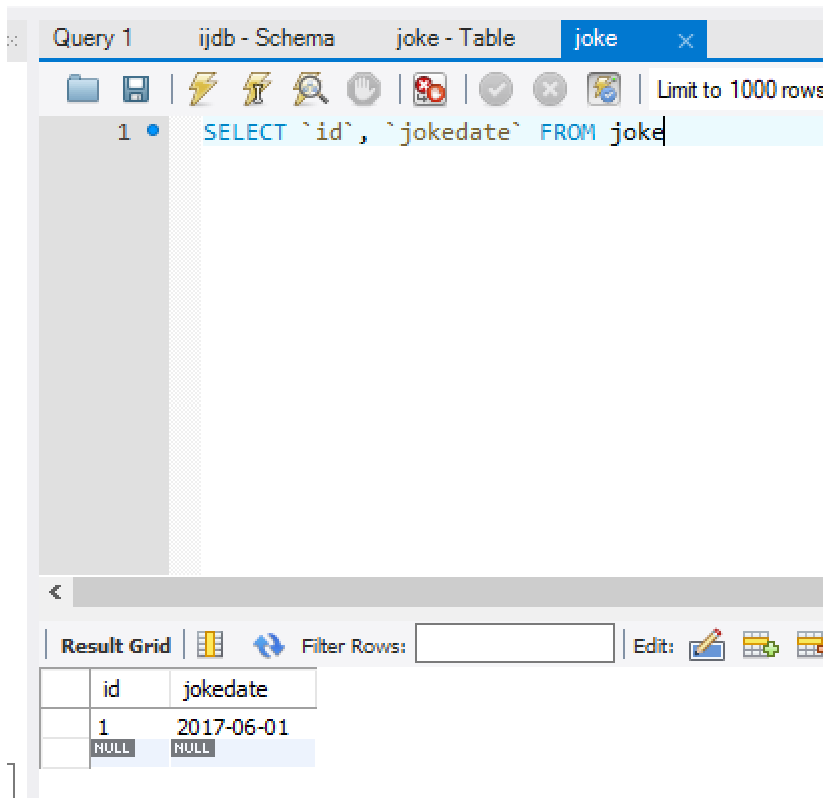
Notice that there are some values in the `id` column, even though you didn't specify them in the `INSERT` queries you ran earlier. MySQL has automatically assigned an ID to the joke. This is because you checked the "AI" (Auto Increment) checkbox when you created the table. If you hadn't checked the box, you'd have needed to specify the ID for each joke you inserted.

If you were doing serious work on such a database, you might be tempted to

stop and read all the hilarious jokes in the database at this point. To save yourself the distraction, you might want to tell MySQL to omit the `joketext` column. The command for doing this is as follows:

```
SELECT `id`, `jokedate` FROM joke
```

This time, instead of telling it to “select everything”, we told it precisely which columns we wanted to see. The result should look like the image below.



3-20. You can select only what you need

What if we’d like to see *some* of the joke text? As well as being able to name specific columns that we want the `SELECT` command to show us, we can use functions to modify each column’s display. One function, called `LEFT`, enables us to tell MySQL to display a column’s contents up to a specified number of characters. For example, let’s say we wanted to see only the first 20

characters of the `joketext` column. Here's the command we'd use:

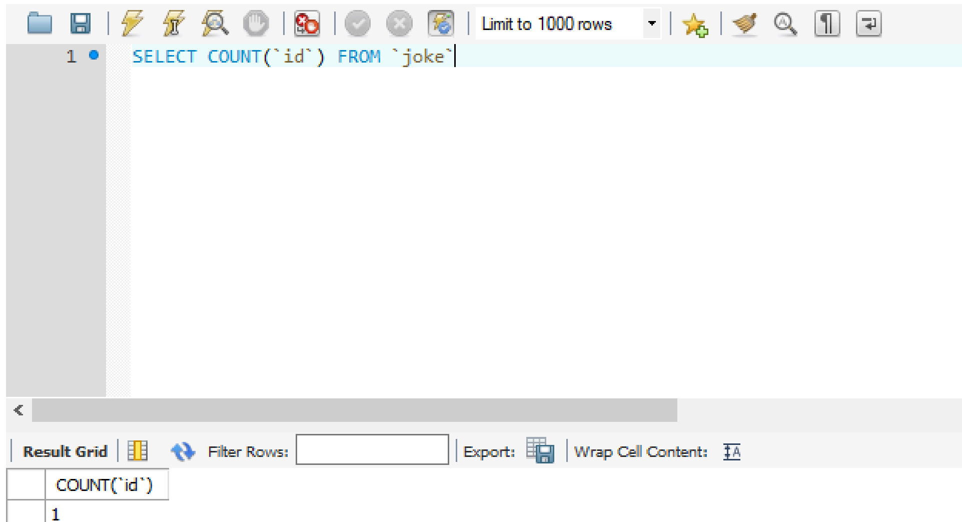
```
SELECT `id`, LEFT(`joketext`, 20), `jokedate` FROM `joke`
```

3-21. The LEFT function trims the text to a specified length

See how that worked? Another useful function is `COUNT`, which lets us count the number of results returned. If, for example, you wanted to find out how many jokes were stored in your table, you could use the following command:

```
SELECT COUNT(`id`) FROM `joke`
```

As you can see in the image below, you have just two jokes in your table.



3-22. The COUNT function counts the rows

Using * Instead

You can use `COUNT(*)` for the same result, but this is slower, as all the columns will be selected from the table. By using the primary key, only one column needs to be retrieved.

So far, the examples we've looked at have fetched all the entries in the table. However, you can limit your results to only those database entries that have the specific attributes you want. You set these restrictions by adding what's called a `WHERE` clause to the `SELECT` command. Consider this example:

```
SELECT COUNT(*) FROM `joke` WHERE `jokedate` >= "2021-01-01"
```

This query will count the number of jokes that have dates greater than or equal to January 1, 2021. In the case of dates, "greater than or equal to" means "on or after". Another variation on this theme lets you search for entries that contain a certain piece of text. Check out this query:

```
SELECT `joketext` FROM `joke` WHERE `joketext` LIKE "%programmer%"
```

This query displays the full text of all jokes containing the text "programmer" in their `joketext` column. The `LIKE` keyword tells MySQL that the named column must match the given pattern. In this case, the pattern we've used is `"%programer%"`. The `%` signs (called **wildcards**) indicate that the text "programmer" may be preceded and/or followed by any string of text. (Interestingly, `LIKE` is case-insensitive, so this pattern will also match a joke that contains "Programmer", or even "FuNkYProGRammeR".)

Conditions may also be combined in the `WHERE` clause to further restrict results. For example, to display knock-knock jokes from April 2021 only, you could use the following query:

```
SELECT `joketext` FROM `joke` WHERE
`joketext` LIKE "%knock%" AND
`jokedate` >= "2021-04-01" AND
`jokedate` < "2021-05-01"
```

Enter a few more jokes into the table. (For example, "Why did the programmer quit his job? He didn't get arrays.") Then experiment with `SELECT` queries.

You can do a lot with the `SELECT` command, so I'd encourage you to become quite familiar with it. We'll look at some of its more advanced features later, when we need them.

Modifying Stored Data

Having entered data into a database table, you might find that you'd like to change it. Whether you're correcting a spelling mistake, or changing the date attached to a joke, such alterations are made using the `UPDATE` command. This command contains elements of the `SELECT` and `INSERT` commands, since the command both picks out entries for modification and sets column values. The general form of the `UPDATE` command is as follows:

```
UPDATE `tableName` SET
  `colName` = newValue, …
WHERE conditions
```

So, for example, if we wanted to change the date on the joke we entered earlier, we'd use the following command:

```
UPDATE `joke` SET `jokedate` = "2021-04-01" WHERE id = "1"
```

Here's where that `id` column comes in handy, enabling you to easily single out a joke for changes. The `WHERE` clause used here works just as it did in the `SELECT` command. This next command, for example, changes the date of all entries that contain the word "programmer":

```
UPDATE `joke` SET `jokedate` = "2021-04-01"
WHERE `joketext` LIKE "%programmer%"
```

WHERE is Optional

Believe it or not, the `WHERE` clause in the `UPDATE` command is optional. Consequently, you should be very careful when typing this command! If you leave the `WHERE` clause out, the `UPDATE` command will then apply to *all entries in the table*.

The following command will set the date for all the records in the table!

```
UPDATE `joke` SET `jokedate` = "2021-04-01"
```

Deleting Stored Data

Deleting entries in SQL is dangerously easy, which you've probably noticed is a recurring theme. Here's the command syntax:

```
DELETE FROM `tableName` WHERE conditions
```

To delete all programmer jokes from your table, you'd use the following query:

```
DELETE FROM `joke` WHERE `joketext` LIKE "%programmer%"
```

Again, WHERE is Optional

As with `UPDATE`, the `WHERE` clause in the `DELETE` command is optional. Consequently, you should be very careful when using it. If you leave the `WHERE` clause out, the `DELETE` command will then apply to *all entries in the table*.

The following command will empty the joke table in one fell swoop:

```
DELETE FROM `joke`
```

Scary, huh?

Let PHP Do the Typing

There's a lot more to the MySQL database server software and SQL than the handful of basic commands I've presented here, but these commands are by far the most commonly used and most useful!

At this stage, you might be thinking that databases seem a little cumbersome. SQL can be tricky to type, as its commands tend to be long and verbose compared with those of other computer languages. You're probably dreading the thought of typing in a complete library of jokes in the form of `INSERT` commands.

Don't sweat it! As we proceed through this book, you'll be surprised how few SQL queries you actually type by hand. Generally, you'll be writing PHP scripts that type your SQL for you. For example, if you want to be able to insert a

bunch of jokes into your database, you'll typically create a PHP script for adding jokes that includes the necessary `INSERT` query, with a placeholder for the joke text. You can then run that PHP script whenever you have jokes to add. The PHP script prompts you to enter your joke, then issues the appropriate `INSERT` query to your MySQL server.

For now, however, it's important to develop a good feel for typing SQL by hand. It will give you a strong sense of the inner workings of MySQL databases, and will make you appreciate all the more the work that PHP will save you from having to do!

To date, we've only worked with a single table, but to realize the true power of a relational database, you'll need to learn how to use multiple tables together to represent potentially complex relationships between the items stored in your database. I'll cover all this and more in Chapter 5, in which I'll discuss database design principles and show off some more advanced examples.

In the meantime, we've accomplished our objective, and you can comfortably interact with MySQL using the MySQL Workbench query window. In Chapter 4, the fun continues as we delve into the PHP language, and use it to create several dynamically generated web pages.

If you like, you can practice with MySQL a little before you move on, by creating a decent-sized joke table (for our purposes, five should be enough). This library of jokes will come in handy when you reach Chapter 5.

Chapter

4

Publishing MySQL Data on the Web

This is it—the stuff you signed up for! In this chapter, you'll learn how to take information stored in a MySQL database and display it on a web page for all to see.

So far, you've written your first PHP code and learned the basics of MySQL, a relational database engine, and PHP, a server-side scripting language.

Now you're ready to learn how to use these tools together to create a website where users can view data from the database and even add their own.

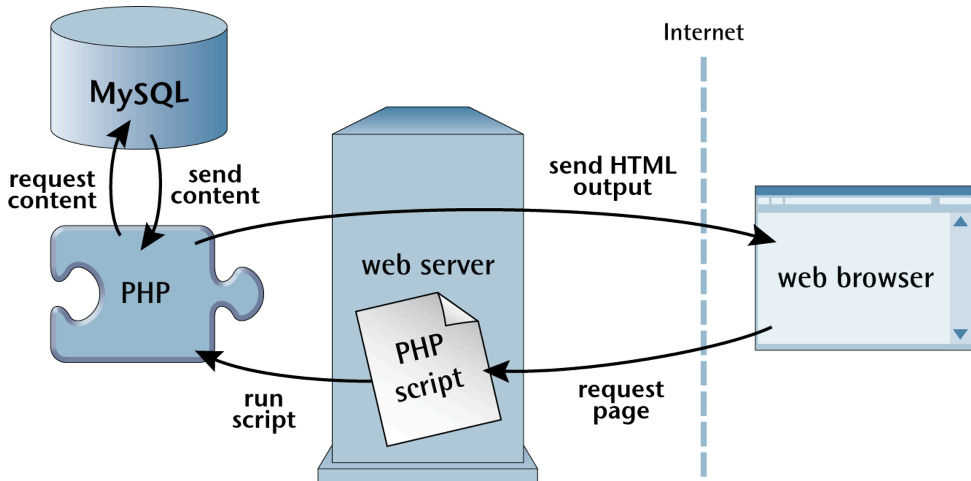
MySQL and MariaDB

As in Chapter 3, I'm using "MySQL" here to refer to the database protocol. Your PHP scripts will do the same. There are numerous references in this chapter—and in the PHP code you'll write—to "MySQL", even though we're actually connecting to a MariaDB database.

The Big Picture

Before we leap forward, it's worth taking a step back for a clear picture of our ultimate goal. We have two powerful tools at our disposal: the PHP scripting language and the MySQL database engine. It's important to understand how these will fit together.

The purpose of using MySQL for our website is to allow the content to be pulled dynamically from the database to create web pages for viewing in a regular browser. So, at one end of the system you have a visitor to your site using a web browser to request a page. That browser expects to receive a standard HTML document in return. At the other end you have the content of your site, which sits in one or more tables in a MySQL database that only understands how to respond to SQL queries (commands).



4-1. Relationships between the web server, browser, PHP and MySQL

As shown in the image above, the PHP scripting language is the go-between that speaks both languages. It processes the page request and fetches the data from the MySQL database using SQL queries just like those you used to create a table of jokes in Chapter 3. It then spits it out dynamically as the nicely formatted HTML page that the browser expects.

Just so it's clear and fresh in your mind, this is what happens when there's a visitor to a page on your website:

1. The visitor's web browser requests the web page from your web server.

2. The web server software (typically Apache or NGINX) recognizes that the requested file is a PHP script, so the server fires up the PHP interpreter to execute the code contained in the file.

3. Certain PHP commands (which will be the focus of this chapter) connect to the MySQL database and request the content that belongs in the web page.

4. The MySQL database responds by sending the requested content to the PHP script.

5. The PHP script stores the content into one or more PHP variables, then uses `echo` statements to output the content as part of the web page.

6. The PHP interpreter finishes up by handing a copy of the HTML it has created to the web server.

7. The web server sends the HTML to the web browser as it would a plain HTML file, except that instead of coming directly from an HTML file, the page is the output provided by the PHP interpreter. The browser has no way of knowing this, however. As far as the browser is concerned, it's requesting and receiving a web page like any other.

Creating a MySQL User Account

In order for PHP to connect to your MySQL database server, it will need to use a username and password. So far, all that your joke database contains is a number of pithy *bon mots*, but before long it may contain sensitive information like email addresses and other private details about the users of your website. For this reason, MySQL is designed to be very secure, giving you tight control over what connections it will accept and what those connections are allowed to do.

The Docker environment already contains a MySQL user in Chapter 3, which you've already used to log in to the MySQL server.

You *could* connect to the database from your PHP script using the same username (`v.je`) and password (`v.je`), but it's useful to create a new account—because if you have a web server, you may want to use it to host more than one website. By giving each website its own user account, you'll have more control over who has access to the data for any given site. If you're working with other developers, you can give them access to the sites they're working on, but no more.

You should create a new user account with only the specific privileges it needs to work on the `ijdb` database that your website depends upon. Let's do that now.

To create a user, open up MySQL Workbench and connect to your server. Then run the following queries:

```
CREATE USER 'ijdbuser'@'%' IDENTIFIED BY 'mypassword';
GRANT ALL PRIVILEGES ON `ijdb`.* TO 'ijdbuser'@'%';
```

The first query is fairly self explanatory: It creates a user called `ijdbuser` with the password `mypassword`. The `%` sign after the username indicates that the database can be connected to from any location. The second query gives the user full acces to the `ijdb` schema, as a result this user can see and modify all the tables, columns and data in the `ijdb` schema but has no access to anything outside it.

Now that the user `ijdbuser` has been created, we can use it to connect to the database. It's possible to set up a connection in MySQL Workbench with this user, but since the permissions are limited, it's better to keep MySQL Workbench using the `v.je` account. Instead, we're going to use the new user when connecting from a PHP script.

Connecting to MySQL with PHP

Before you can retrieve content from your MySQL database for inclusion in a web page, you must know how to establish a connection to MySQL from inside a PHP script. So far, you've used an application called MySQL Workbench to connect to your database. Just as MySQL Workbench can connect directly to a running MySQL server, so too can your own PHP scripts.

Although this chapter talks entirely about connecting to MySQL from PHP, we're actually connecting to the MariaDB database discussed in the previous chapter. PHP can't see any difference between MySQL and MariaDB, as they're interchangeable. I'll refer to the database as MySQL throughout, because all of the commands used could be used to connect to a MySQL or MariaDB database server.

The original MySQL database provided a standardized method for *clients* such as MySQL Workbench and PHP to communicate with the server. MariaDB

copied that standard, and all the commands in PHP use the name *MySQL*, so to keep things simple, I'll use the term *MySQL* throughout this chapter to refer to the database.

There are three methods of connecting to a MySQL server from PHP:

- the MySQL library
- the MySQLi library
- the PDO library

These all essentially do the same job—connecting to the database and sending queries to it—but they use different code to achieve it.

The MySQL library is the oldest method of connecting to the database and was introduced in PHP 2.0. The features it contains are minimal, and it was superseded by MySQLi as of PHP 5.0 (released in 2004).

To connect and query the database using the old MySQL library, functions such as `mysql_connect()` and `mysql_query()` are used. These functions have been **deprecated**—meaning they should be avoided—since PHP 5.5, and have been removed from PHP entirely since PHP 7.0.

Although most developers saw the reason for the change as soon as PHP 5.0 was released, there are still hundreds of articles and code examples on the Web using these now non-existent `mysql_*` functions—despite the fact that MySQLi has effectively been the preferred library for fifteen years.

If you come across a code example that contains the line `mysql_connect()`, check the date of the article. It's probably from the early 2000s, and in programming, you should never trust anything that old. Things change all the time—which is why this book is on its seventh edition!

In PHP 5.0, the MySQLi library—standing for "MySQL Improved"—was released to address some of the limitations in the original MySQL library. You can easily identify the use of MySQLi, because the code will use functions such as `mysqli_connect()` and `mysqli_query()`.

Shortly after the release of the MySQLi library in PHP 5.0, PHP 5.1 was released, with a significant number of changes that helped shape the way we write PHP today (mostly to do with object-oriented programming, which you'll see plenty of later in this book). One of the major changes in PHP 5.1 was that it introduced a third library, PDO (PHP Data Objects), for connecting to MySQL databases.

There are several differences between PDO and MySQLi, but the main one is that you can use the PDO library to connect to almost any database server—such as an Oracle server, or Microsoft SQL Server. For developers, the biggest advantage of this generic approach is that, once you've learned how to use the library to interact with a MySQL database, it's very simple to interact with another database server.

Arguably, it's simpler to write code for PDO, and there are some nuances that can make PDO code more readable—named parameters in prepared statements being the main benefit. (Don't worry, I'll explain what that means later on.)

For these reasons, most recent PHP projects use the PDO library, and it's the library I'm going to show you how to use in this book. For more information on the differences, take a look at the SitePoint article “>Re-introducing PDO – the Right Way to Access Databases in PHP”[1].

After that little history lesson, you're probably eager to get back to writing code. Here's how you use PDO to establish a connection to a MySQL server:

```
new PDO('mysql:host=hostname;dbname=database', 'username',
  'password')
```

For now, think of `new PDO` as a built-in function, just like the `rand` function we used in Chapter 2. If you're thinking “Hey, functions can't have *spaces* in their names!”, you're smarter than the average bear, and I'll explain exactly what's

---

1. https://www.sitepoint.com/re-introducing-pdo-the-right-way-to-access-databases-in-php/

going on here in a moment. In any case, it takes three arguments:

- a string specifying the type of database (`mysql:`), the hostname of the server (`host=hostname;`), and the name of the database (`dbname=database`)
- the MySQL username you want PHP to use
- the MySQL password for that username

You may remember from Chapter 2 that PHP functions usually return a value when they're called. This `new PDO` "function" returns a value called a `PDO` object that identifies the connection that's been established. Since we intend to make use of the connection, we should hold onto this value by storing it in a variable. Here's how that looks, with the necessary values filled in to connect to your database:

```
$pdo = new PDO('mysql:host=mysql;dbname=ijdb', 'ijdbuser',
  'mypassword');
```

You can probably see what's going on with the last two arguments: they're the username and password you created earlier in this chapter.

The first argument is a little more complicated. The `dbname=ijdb` part tells PDO to use the database (also referred to as a *schema*) called `ijdb` . Any query run from PHP will default to tables in that schema. `SELECT * FROM joke` will select records from the `joke` table in the `ijdb` schema.

Even if you're familiar with PHP, PDO and MySQL already, the `host=mysql` part looks confusing. Normally, this would be `host=localhost` (referring to the local computer, the same machine running PHP) or pointing to a specific domain name where the database is hosted, such as `host=sitepoint.com` .

Why is it `host=mysql` , and what does `mysql` refer to here? In Docker, each *service* is given a name. If you examine the `docker-compose.yml` file that configures the server, the database service is called `mysql` , and in Docker, one service can connect to another using the other service's name.

Arguments aside, what's important to see here is that the value returned by `new PDO` is stored in a variable named `$pdo`.

The MySQL server is a completely separate piece of software from the web server. Therefore, we must consider the possibility that the server may be unavailable or inaccessible due to a network outage, or because the username/password combination you provided is rejected by the server, or because you just forgot to start your MySQL server! In such cases, `new PDO` won't run, and will throw a PHP exception.

Turning Off Exception Throwing

At least by default, PHP can be configured so that no exception is thrown and it simply won't connect. This isn't generally desirable behavior, as it makes it much more difficult to work out what went wrong.

If you're wondering what it means to "throw a PHP exception", brace yourself! You're about to discover some more features of the PHP language.

A PHP **exception** is what happens when you tell PHP to perform a task and it's unable to do it. PHP will try to do what it's told, but will fail; and in order to tell you about the failure, it will throw an exception at you. An exception is little more than PHP just crashing with a specific error message. When an exception is thrown, PHP stops. No lines of code after the error will be executed.

As a responsible developer, it's your job to catch that exception and do something about it so the program can continue.

Uncaught Exceptions

If you don't catch an exception, PHP will stop running your PHP script and display a spectacularly ugly error message. That error message will even reveal the code of your script that threw the error. In this case, that code contains your MySQL username and password, so it's especially important to avoid the error message being seen by users!

To catch an exception, you should surround the code that might throw an exception with a `try … catch` statement:

```
try {
  ⋮ do something risky
}
catch (ExceptionType $e) {
  ⋮ handle the exception
}
```

You can think of a `try … catch` statement like an `if … else` statement, except that the second block of code is what happens if the first block of code fails to run.

Confused yet? I know I'm throwing (no pun intended) a lot of new concepts at you, but it will make more sense if I put it all together and show you what we have:

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb', 'ijdbuser',
    'mypassword');
  $output = 'Database connection established.';
}
catch (PDOException $e) {
  $output = 'Unable to connect to the database server.';
}

include  __DIR__ . '/../templates/output.html.php';
```

As you can see, this code is a `try … catch` statement. In the `try` block at the top, we attempt to connect to the database using `new PDO`. If this succeeds, we store the resulting PDO object in `$pdo` so that we can work with our new database connection. If the connection is successful, the `$output` variable is set to a message that will be displayed later.

Importantly, inside a `try … catch` statement, any code after an exception has been thrown won't get executed. In this case, if connecting to the database throws an exception (maybe the password is wrong, or the server isn't responding), the `$output` variable will never get set to "Database connection established".

If our database connection attempt fails, PHP will throw a `PDOException`, which is the type of exception that `new PDO` throws. Our `catch` block, therefore, says that it will catch a PDOException (and store it in a variable named `$e`). Inside that block, we set the variable `$output` to contain a message about what went wrong.

However, this error message isn't particularly useful. All it tells us is that PDO couldn't connect to the database server. It would be better to have some information about why that was—for example, because the username and password were invalid.

The `$e` variable contains details about the exception that occurred, including an error message describing the problem. We can add this to the output variable using concatenation:

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb', 'ijdbuser',
    'mypassword');
   $output = 'Database connection established.';
}
catch (PDOException $e) {
  $output = 'Unable to connect to the database server: ' . $e->getMessage();
}
```

```
include  __DIR__ . '/../templates/output.html.php';
```

The `$e` Variable

The `$e` variable isn't a string, but an *object*. We'll come to what that means shortly. For now, though, all you need to know is that the code `$e->getMessage()` gets the error message based on the specific exception that occurred.

Like an `if … else` statement, one of the two branches of a `try … catch` statement is guaranteed to run. Either the code in the `try` block will execute successfully, or the code in the `catch` block will run. Regardless of whether the database connection was successful, there will be a message in the `$output` variable—either the error message, or the message saying the connection was successful.

Finally, regardless of whether the `try` block was successful, or the `catch` block runs, the template `output.html.php` is included. This is a generic template that just displays some text to the page:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Script Output</title>
  </head>
  <body>
      <?php echo $output; ?>
  </body>
</html>
```

The complete code can be found in **Example: MySQL-Connect**[2].

When the template is included, it will display either the error message or the

2. https://github.com/spbooks/phpmysql7/tree/MySQL-Connect

"Database connection established" message.

I hope the aforementioned code is now making some sense to you. Feel free to go back to the start of this section and read it all again if you're lost, as there were some tricky concepts in there. Once you have a firm grip on the code, however, you'll probably realize that I've still left one mystery unexplained: PDOs. Just what exactly is `new PDO`, and when I said it returns a "PDO object", just what exactly is an object?

Schemas

All downloaded sample code includes a schema called `ijdb_sample` and a user called `ijdb_sample`, so that you're able to run it regardless of what you called your schema and user. A file containing the database is provided as `database.sql`, which you can import.

If you use the web-based sample code viewer provided, the `idbj_sample` database will be created as you load a sample, but any changes to this schema will be lost when you view another sample. (You can mess things up, and switching to another sample and back will reset it, but if you want to keep any changes you make, make them in the schema you created.)

If you want to load the sample data into your schema using MySQL Workbench, import `database.sql` from the `project` directory by selecting **Data Import/Restore**. Then select **Import from self-contained file**, browse to `database.sql`, and select your schema name in **default** target schema. If you have created any tables with the same name, they'll be overwritten and all records lost.

A Crash Course in Object-oriented Programming

You may have noticed the word "object" beginning to creep into my vocabulary in the previous section. PDO is the PHP Data *Objects* extension, and `new PDO` returns a PDO *object*. In this section, I'd like to explain what

objects are all about.

Perhaps you've come across the term **object-oriented programming (OOP)** in your own explorations of PHP or of programming in general. OOP is an advanced style of programming that's especially suited to building really complex programs with a lot of parts. Most programming languages in active use today support OOP. Some of them even *require* you to work in an OOP style. PHP is a little more easygoing about it, and leaves it up to the developer to decide whether or not to write their scripts in the OOP style.

So far, we've written our PHP code in a simpler style called **procedural programming**, and we'll continue to do so for now, with a more detailed look at objects later on. Procedural style is well suited to the relatively simple projects we're tackling at the moment. However, almost all complex projects you'll come across use OOP, and I'll cover it in more detail later in this book.

That said, the PDO extension we'll use to connect to and work with a MySQL database is designed in the object-oriented programming style. This means that, rather than simply calling a function to connect to MySQL and then calling other functions that use that connection, we must first create a PDO *object* that will represent our database connection, and then use the features of that object to work with the database.

Creating an object is a lot like calling a function. In fact, you've already seen how to do it:

```
$pdo = new PDO('mysql:host=mysql;dbname=ijdb', 'ijdbuser',
  'mypassword');
```

The `new` keyword tells PHP that you want to create a new object. You then leave a space and specify a **class name**, which tells PHP what type of object you want to create. A **class** is a set of instructions that PHP will follow to create an object. You can think of a class as being a recipe, such as for a cake, and an object being the actual cake that's produced from following the recipe. Different classes can produce different objects, just as different recipes can produce different dishes.

Just as PHP comes with a bunch of built-in functions that you can call, PHP comes with a library of classes that you can create objects from. `new PDO`, therefore, tells PHP to create a new `PDO` object—that is, a new object of the built-in `PDO` class.

In PHP, an object is a value, just like a string, number, or array. You can store an object in a variable or pass it to a function as an argument—all the same stuff you can do with other PHP values. Objects, however, have some useful additional features.

First of all, an object behaves a lot like an array, in that it acts as a container for other values. As we saw in Chapter 2, you can access a value inside an array by specifying its index (for example, `$birthdays['Kevin']`). When it comes to objects, the concepts are similar but the names and code are different. Rather than accessing the value stored in an array index, we say that we're accessing a **property** of the object. Instead of using square brackets to specify the name of the property we want to access, we use **arrow notation** (`->`)—for instance, `$myObject->someProperty`:

```
$myObject = new SomeClass();    // create an object
$myObject->someProperty = 123;  // set a property's value
echo $myObject->someProperty;   // get a property's value
```

Whereas arrays are normally used to store a list of *similar* values (such as an array of birthdays), objects are used to store a list of *related* values (for example, the properties of a database connection). Still, if that's all objects did, there wouldn't be much point to them: we might just as well use an array to store these values, right? Of course, objects do more.

In addition to storing a collection of properties and their values, objects can contain a group of functions designed to bring us more useful features. A function stored in an object is called a **method** (one of the more confusing names in the programming world, if you ask me). A method is just a function inside a class. More confusingly, when we get onto writing our own classes, methods are defined using the `function` keyword! Even experienced developers often wrongly use *function* and *method* interchangeably.

To call a method, we again use arrow notation— `$myObject->someMethod()` :

```
$myObject = new SomeClass();    // create an object
$myObject->someMethod();        // call a method
```

Just like standalone functions, methods can take arguments and return values.

At this stage, this is probably all sounding a little complicated and pointless, but trust me: pulling together collections of variables (properties) and functions (methods) into little bundles called objects results in much tidier and easier-to-read code for certain tasks—working with a database being just one of them. One day, you may even want to develop custom classes that you can use to create objects of your own devising.

For now, however, we'll stick with the classes that come included with PHP. Let's keep working with the `PDO` object we've created, and see what we can do by calling one of its methods.

Configuring the Connection

So far, I've shown you how to create a `PDO` object to establish a connection with your MySQL database, and how to display a meaningful error message when something goes wrong:

```
<?php
try {
    $pdo = new PDO('mysql:host=mysql;dbname=ijdb', 'ijdbuser', 'mypassword');
    $output = 'Database connection established.';
} catch (PDOException $e) {
    $output = 'Unable to connect to the database server: ' . $e->getMessage();
}

include  __DIR__ . '/../templates/output.html.php';
```

Assuming the connection succeeds, though, you need to configure it before use. You can configure your connection by calling some methods of your new

`PDO` object.

Before sending queries to the database, we'll need to configure the character encoding of our database connection. As I mentioned briefly in Chapter 2, you should use UTF-8 encoded text in your websites to maximize the range of characters users have at their disposal when filling in forms on your site. By default, when PHP connects to MySQL, it uses the simpler ISO-8859-1 (or Latin-1) encoding instead of UTF-8. If we were to leave it as is, we wouldn't easily be able to insert Chinese, Arabic or most non-English characters.

Even if you're 100% sure that your website will only be used by English speakers, there are other problems caused by not setting the character set. If your web page is not set to UTF-8, you'll run into problems when people write certain characters such as curly quotes `”` into a text box, because they'll appear in the database as a different character.

Therefore, we now need to set our new `PDO` object to use the UTF-8 encoding.

We can instruct PHP to use UTF-8 when querying the database by appending `;charset=utf8mb4` to the connection string. There are no downsides to doing this, provided your PHP script is also being sent to the browser as `UTF-8` (which is the default in recent PHP versions):

```
$pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  'mypassword');
```

Other Ways to Set the Charset

If you go searching, you'll find different ways to set the charset, and earlier editions of this book instructed you to use this code:

```
$pdo->exec('SET NAMES "utf8"');
```

This is because, until PHP 5.3.6, the charset option was not correctly applied by PHP. Since this is fixed in any PHP version you're actually going to be using, setting the charset as part of the connection string is the preferred option.

The complete code we use to connect to MySQL and then configure that connection, therefore, is shown below.

Example: MySQL-Connect-Complete[3]

```
<?php
try {
    $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
    ↪'mypassword');
    $output = 'Database connection established.';
} catch (PDOException $e) {
    $output = 'Unable to connect to the database server: ' . $e->getMessage();
}

include  __DIR__ . '/../templates/output.html.php';
```

Fire up this example in your browser. (If you've placed your database code in `index.php` inside the `public` directory and the `output.html.php` file in the `templates` directory, the URL for the page will be `https://v.je/`.)

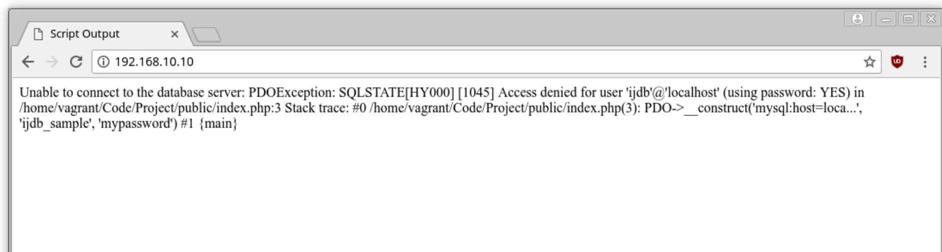
If your server is up and running, and everything is working properly, you should see a message indicating success.

---

3. https://github.com/spbooks/phpmysql7/tree/MySQL-Connect-Complete

4-2. A successful connection

If PHP is unable to connect to your MySQL server, or if the username and password you provided are incorrect, you'll instead see a similar screen to that shown below. To make sure your error-handling code is working properly, you might want to misspell your password intentionally to test it out.



4-3. A connection failure

Thanks to our `catch` block, the error message from the database has been included on the page:

```
catch (PDOException $e) {
  $output = 'Unable to connect to the database server: ' . $e->getMessage();
}
```

The method `getMessage()` returns a message describing the exception that occurred. There are some other methods—including `getFile()` and `getLine()` —for returning the file name and line number that the exception was thrown on. You can generate a very detailed error message like this:

```
catch (PDOException $e) {
```

```
  $output = 'Unable to connect to the database server: ' . $e->getMessage() .
  ↪ ' in ' .
  $e->getFile() . ':' . $e->getLine();
}
```

This is incredibly useful if you have a large website with dozens of include files. The error message will tell you exactly which file to look in and which line the error occurred on.

If you're curious, try inserting some other mistakes in your database connection code (for example, a misspelled database name) and observe the detailed error messages that result. When you're done, and your database connection is working correctly, go back to the simple error message. This way, your visitors won't be bombarded with technical gobbledygook if a genuine problem emerges with your database server.

With a connection established and a database selected, you're ready to begin using the data stored in the database.

What Happens after the Script Has Finished?

You might be wondering what happens to the connection with the MySQL server after the script has finished executing. If you really want to, you can force PHP to disconnect from the server by discarding the `PDO` object that represents your connection. You do this by setting the variable containing the object to `null`:

```
$pdo = null;  // disconnect from the database server
```

That said, PHP will automatically close any open database connections when it finishes running your script, so you can usually just let PHP clean up after you.

Sending SQL Queries with PHP

In Chapter 3, we connected to the MySQL database server using MySQL

Workbench, which allowed us to type SQL queries (commands) and view the results of those queries immediately. The `PDO` object offers a similar mechanism—the exec method:

```
$pdo->exec($query)
```

Here, `$query` is a string containing whatever SQL query you want to execute.

As you know, if there's a problem executing the query (for instance, if you made a typing mistake in your SQL query), this method will throw a `PDOException` for you to catch.

Consider the following example, which attempts to produce the joke table we created in Chapter 3.

Example: MySQL-Create[4]

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  ➥ 'mypassword');

  $sql = 'CREATE TABLE joke (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    joketext TEXT,
    jokedate DATE NOT NULL
  ) DEFAULT CHARACTER SET utf8 ENGINE=InnoDB';

  $pdo->exec($sql);

  $output = 'Joke table successfully created.';
}
catch (PDOException $e) {
  $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/output.html.php';
```

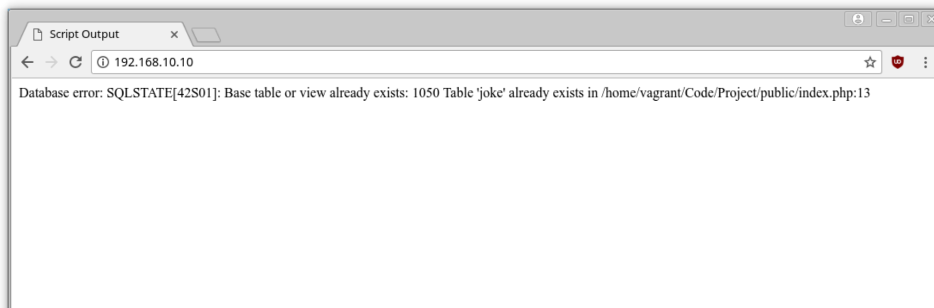
4. https://github.com/spbooks/phpmysql7/tree/MySQL-Create

Note that we use the same `try … catch` statement technique to handle possible errors produced by the query. It would be possible to use multiple `try … catch` blocks to display different error messages—one for the connection and one for the query—but this can result in a considerable amount of extra code.

Instead, I've opted to use the same `try` statement to contain both the connection and the query. The `try … catch` block will stop executing code once an error occurs, so if an error occurs during the database connection, the `$pdo->exec($run)` line will never run, ensuring that, if a query is sent to the database, a connection must have been established.

This approach gives us a little less control over the error message that's displayed, but saves typing a `try … catch` statement for each database operation. Later in this book, we'll break these up into different blocks, but for now, keep all the database operations in the same `try` block.

This example also uses the `getMessage` method to retrieve a detailed error message from the MySQL server. The following image shows the error that's displayed when, for example, the joke table already exists.



4-4. The CREATE TABLE query fails because the table already exists

For `DELETE`, `INSERT`, and `UPDATE` queries (which serve to modify stored data), the `exec` method returns the number of table rows (entries) that were affected by the query. Consider the following SQL command, which we used in Chapter 3 to set the dates of all jokes that contained the word "programmer".

Example: MySQL-Update[5]

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  ↪ 'mypassword');

  $sql = 'UPDATE joke SET jokedate="2021-04-01"
      WHERE joketext LIKE "%programmer%"';

  $affectedRows = $pdo->exec($sql);

  $output = 'Updated ' . $affectedRows .' rows.';
}
catch (PDOException $e) {
  $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/output.html.php';
```

By storing the value returned from the exec method in `$affectedRows`, we can use the variable in the `$output` variable for printing in the template.

The image below shows the output of this example, assuming there's only one "programmer" joke in your database.

4-5. The number of database records updated is displayed

If you refresh the page to run the same query again, you should see the message change, as shown in the following image. It indicates that no rows

5. https://github.com/spbooks/phpmysql7/tree/MySQL-Update

were updated, since the new date being applied to the jokes is the same as the existing date.

4-6. MySQL lets you know when you're wasting its time

`SELECT` queries are treated a little differently, as they can retrieve a lot of data, and PHP provides ways to handle that information.

Handling `SELECT` Result Sets

For most SQL queries, the `exec` method works just fine. The query does something to your database, and you get the number of affected rows (if any) from the method's return value. `SELECT` queries, however, require something a little fancier than `exec`. You'll recall that `SELECT` queries are used to view stored data in the database. Instead of only affecting the database, `SELECT` queries have results—and we need a method to return them.

The query method looks just like `exec`, in that it accepts an SQL query as an argument to be sent to the database server. What it returns, however, is a `PDOStatement` object, which represents a **result set** containing a list of all the rows (entries) returned from the query:

```
<?php
try {
    $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
    ↪ 'mypassword');
```

```
    $sql = 'SELECT `joketext` FROM `joke`';
    $result = $pdo->query($sql);
} catch (PDOException $e) {
    $error = 'Unable to connect to the database server: ' . $e->getMessage() .
    ➥ ' in ' .
  $e->getFile() . ':' . $e->getLine();
}
```

Provided that no error was encountered in processing the query, this code will store a result set (in the form of a `PDOStatement` object) into the variable `$result`. This result set contains the text of all the jokes stored in the `joke` table. As there's no practical limit on the number of jokes in the database, the result set can be quite big.

I mentioned back in Chapter 2 that the `while` loop is a useful control structure when we need to loop but don't know how many times. We can't use a `for` loop, because we don't know how many records the query returned. Indeed, you could use a `while` loop here to process the rows in the result set one at a time:

```
while ($row = $result->fetch()) {
  ⋮ process the row
}
```

The condition for the `while` loop is probably different from the conditions you're used to, so let me explain how it works. Consider the condition as a statement all by itself:

```
$row = $result->fetch();
```

The `fetch` method of the `PDOStatement` object returns the next row in the result set as an array (we discussed arrays in Chapter 2). When there are no more rows in the result set, `fetch` returns `false` instead. (This is one case where asking a PDO object to do something it can't do—as `fetch` can't return the next row when there are no rows left in the result set—will *not* throw a `PDOException`. If it did, we'd be unable to use the `fetch` method in a `while`

loop condition the way we do here.)

Now, the above statement assigns a value to the `$row` variable, but, at the same time, the statement as a whole takes on that same value. This is what lets you use the statement as a condition in the `while` loop. Since a `while` loop will keep looping until its condition evaluates to `false`, this loop will occur as many times as there are rows in the result set, with `$row` taking on the value of the next row each time the loop executes. All that's left to figure out is how to retrieve the values out of the `$row` variable each time the loop runs.

Rows of a result set returned by `fetch` are represented as associative arrays, with the indices named after the table columns in the result set. If `$row` is a row in our result set, `$row['joketext']` is the value in the `joketext` column of that row.

Our goal in this code is to store away the text of all the jokes so that we can display them in a PHP template. The best way to do this is to store each joke as a new item in an array, `$jokes`:

```
while ($row = $result->fetch()) {
  $jokes[] = $row['joketext'];
}
```

With the jokes pulled out of the database, we can now pass them along to a PHP template `jokes.html.php`.

To summarize, here's the code of the controller for this example so far:

```
<?php

try {
    $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
    ↪ 'mypassword');

    $sql = 'SELECT `joketext` FROM `joke`';
```

```
    $result = $pdo->query($sql);

    while ($row = $result->fetch()) {
        $jokes[] = $row['joketext'];
    }
} catch (PDOException $e) {
    $error = 'Unable to connect to the database server: ' . $e->getMessage() .
    ↪' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/jokes.html.php';
```

The `$jokes` variable is an array that stores a list of jokes. If you wrote out the contents of the array in PHP, it would look something like this:

```
$jokes = [];
$jokes[0] = 'A programmer was found dead in the shower. The instructions read:
↪lather, rinse, repeat.';
$jokes[1] = '!false - it\'s funny because it\'s true';
$jokes[2] = 'A programmer\'s wife tells him to go to the store and "get a gallon
↪ of milk, and if they have eggs, get a dozen." He returns with 13 gallons of milk.';
```

However, the data has been retrieved from the database rather than being typed out manually in the code.

You'll have noticed that there are two different variables being set— `$jokes` and `$error` —depending on whether or not the `try` block executed successfully.

In the `jokes.html.php` template, we need to display the contents of the `$jokes` array or the error message contained in the `$error` variable.

To check whether or not a variable has been assigned a value, we can use the `isset` function that we used earlier for checking if a form has been submitted. The template can include an `if` statement to determine whether to display the error or the list of jokes:

```
if (isset($error)) {
  ?>
  <p>
  <?php
  echo $error;
  ?>
  </p>
}
else {
  : display the jokes
}
```

There's nothing new here, but to display the jokes, we need to display the contents of the `$jokes` array. Unlike other variables we've used up to this point, the `$jokes` array contains more than just a single value.

The most common way to process an array in PHP is to use a loop. We've already seen `while` loops and `for` loops. The `foreach` loop is particularly helpful for processing arrays:

```
foreach (array as $item) {
  ⋮ process each $item
}
```

Instead of a condition, the parentheses at the top of a `foreach` loop contain an array, followed by the keyword `as`, and then the name of a new variable that will be used to store each item of the array in turn. The body of the loop is then executed once for each item in the array. Each time that item is stored in the specified variable, so that the code can access it directly.

It's common to use a `foreach` loop in a PHP template to display each item of an array in turn. Here's how this might look for our `$jokes` array:

```
<?php
foreach ($jokes as $joke) {
    ?>
  ⋮ HTML code to output each $joke
```

```
<?php
}
?>
```

With this blend of PHP code to describe the loop and HTML code to display it, the code looks rather untidy. Because of this, it's common to use an alternative way of writing the `foreach` loop when it's used in a template:

```
foreach (array as $item):
  ⋮ process each $item
endforeach;
```

The two pieces of code are functionally identical, but the latter looks more friendly when mixed with HTML code. Here's how this form of the code looks in a template:

```
<?php foreach ($jokes as $joke): ?>
  ⋮ HTML code to output each $joke
<?php endforeach; ?>
```

The same thing can be done with the `if` statement, making it nicer to look at inside HTML templates by avoiding the braces:

```
<?php if (isset($error)): ?>
  <p>
  <?php echo $error; ?>
  </p>
<?php else: ?>
  : display the jokes
<?php endif; ?>
```

With these new tools in hand, we can write our template to display the list of jokes.

Example: MySQL-ListJokes[6]

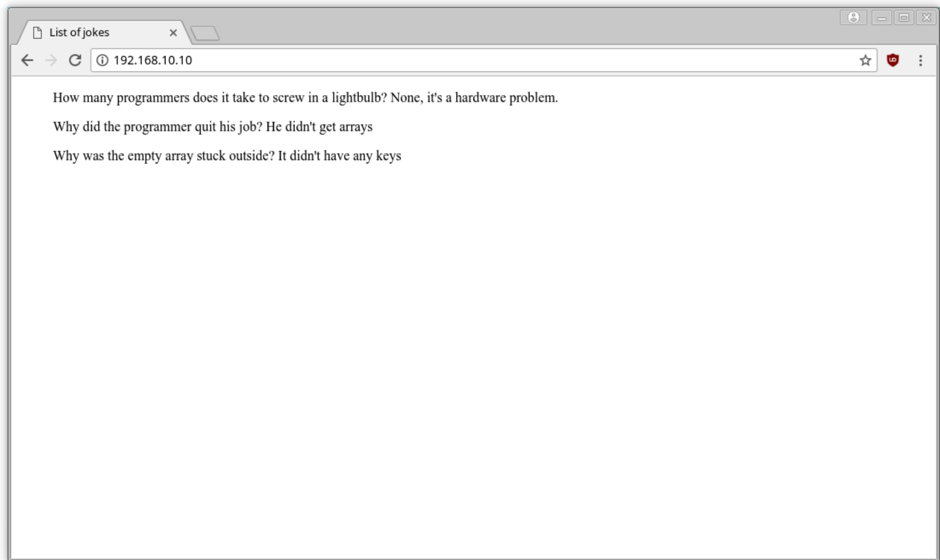
6. https://github.com/spbooks/phpmysql7/tree/MySQL-ListJokes

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>List of jokes</title>
  </head>
  <body>
  <?php if (isset($error)): ?>
  <p>
    <?php echo $error; ?>
  </p>
  <?php else: ?>
  <?php foreach ($jokes as $joke): ?>
  <blockquote>
    <p>
    <?php echo htmlspecialchars($joke, ENT_QUOTES, 'UTF-8') ?>
    </p>
  </blockquote>
  <?php endforeach; ?>
  <?php endif; ?>
  </body>
</html>
```

Either the `$error` text is displayed on the page or each joke is displayed in a paragraph (`<p>`) contained within a block quote (`<blockquote>`), since we're effectively quoting the author of each joke in this page.

Because jokes might conceivably contain characters that could be interpreted as HTML code (for example, `<` , `>` , or `&`), we must use `htmlspecialchars` to ensure they're translated into HTML character entities (that is, `<` , `>` , and `&`) so that they'll be displayed correctly.

The following image shows what this page looks like once you've added a couple of jokes to the database.



4-7. A list of jokes from the database

Using `foreach`

Remember how we used a `while` loop in our controller to fetch the rows out of the `PDOStatement` result set one at a time?

```
while ($row = $result->fetch()) {
  $jokes[] = $row['joketext'];
}
```

It turns out that `PDOStatement` objects are designed to behave just like arrays when you pass them to a `foreach` loop. You can therefore slightly simplify your database processing code using a `foreach` loop instead of a `while` loop:

```
foreach ($result as $row) {
  $jokes[] = $row['joketext'];
}
```

I'll be using this tidier `foreach` form in the rest of this book.

Another neat tool PHP offers is a shorthand way to call the `echo` command—which, as you've already seen, we need to use frequently. Our `echo` statements look like this:

```
<?php echo $variable; ?>
```

Instead, we can use this:

```
<?=$variable?>
```

This does exactly the same thing. `<?=` means `echo` and gives you a slightly shorter way to print variables. There's a limitation to this, though: if you use `<?=`, you can only print. You can't include `if` statements, `for` statements, and so on, although you can use concatenation, and it can be followed by a function call.

Here's an updated template using the shorthand echo.

Example: MySQL-ListJokes-Shorthand[7]

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>List of jokes</title>
  </head>
  <body>
  <?php if (isset($error)): ?>
  <p>
    <?=$error?>
  </p>
  <?php else: ?>
  <?php foreach ($jokes as $joke): ?>
  <blockquote>
    <p>
    <?=htmlspecialchars($joke, ENT_QUOTES, 'UTF-8')?>
```

7. https://github.com/spbooks/phpmysql7/tree/MySQL-ListJokes-Shorthand

```
    </p>
  </blockquote>
  <?php endforeach; ?>
  <?php endif; ?>
  </body>
</html>
```

I'll be using the shorthand notation when it's applicable from this point on.

Using the Shorthand

In versions of PHP prior to 5.4, this shorthand notation required a fairly uncommon PHP setting to be enabled, so it was discouraged for compatibility reasons. Using the shorthand notation may have caused your code to stop working when moving from a server that had it enabled to one that didn't.

As of PHP 5.4 (so any version you're realistically going to encounter these days), the shorthand echo works regardless of PHP settings, so you can safely use it without worrying that it might not work on all servers.

Thinking Ahead

In the example we just looked at, we created a template, `jokes.html.php`, which contains all the HTML required to display the page. However, as our website grows, we'll add more pages. We'll certainly want a page for people to be able to add jokes to the website, and we'll also need a home page with some introductory text, a page with the owner's contact details, and, as the site grows, perhaps even a page where people can *log in* to the website.

I'm jumping ahead a quite a bit here, but it's always worth considering how a project will grow. If we apply the approach we just used for `jokes.html.php` to the rest of the templates—`addjoke.html.php`, `home.html.php`, `contact.html.php`, `login.html.php` and so on—we'll end up with a lot of repeated code.

Every page on the website will require a template that will look something like this:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>IJDB - Internet Joke Database</title>
  </head>
  <body>
  <?php if (isset($error)): ?>
  <p>
    <?=$error?>
  </p>
  <?php else: ?>
    : do whatever is required for this page: show text,
    : show a form, list records from the database, etc.
  <?php endif; ?>
  </body>
</html>
```

As a programmer, repeating code is one of the worst things you can do. In fact, programmers often refer to the **DRY** principle, which stands for “Don’t repeat yourself”. If you find yourself repeating sections of code, there’s almost certainly a better solution.

All the best programmers are lazy, and repeating code means repeating work. Using this copy/paste approach for templates makes the website very difficult to maintain. Let’s imagine there’s a footer and a navigation section that we want to appear on each page. Our templates would now look like this:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>IJDB - Internet Joke Database</title>
  </head>
  <body>
  <nav>
    <ul>
```

```
      <li><a href="index.php">Home</a></li>
      <li><a href="jokes.php">Jokes List</a></li>
    </ul>
  </nav>

  <main>
  <?php if (isset($error)): ?>
  <p>
    <?=$error?>
  </p>
  <?php else: ?>
    : do whatever is required for this page: show text,
    : show a form, list jokes, etc.
  <?php endif; ?>
  </main>

  <footer>
  &copy; IJDB 2021
  </footer>
  </body>
</html>
```

We'll run into a problem in 2022! If the templates for all the pages on the website—for example, `jokes.html.php` `addjoke.html.php`, `home.html.php`, `contact.html.php` and `login.html.php`—contain code in the structure above, to update the year in the copyright notice to "2022" you'd need to open each of the templates and change the date.

We could be clever and have the date dynamically read from the server's clock (`echo date('Y');` if you're curious!) to avoid this issue, but what if we wanted to add a `<script>` tag that was included on every page? Or add a new link to the menu? We'd still need to open every template file and change it!

Changing five or six templates may be slightly annoying, but it's not going to pose much of a problem. However, what if the website grows to dozens or hundreds of pages? Each time you wanted to add a link to the menu you'd have to open every single template and change it.

This problem *could* be solved with a series of `include` statements. For

example:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>IJDB - Internet Joke Database</title>
  </head>
  <body>
  <nav>
    <?php include 'nav.html.php'; ?>
  </nav>

  <main>
  <?php if (isset($error)): ?>
  <p>
    <?=$error?>
  </p>
  <?php else: ?>
    : do whatever is required for this page: show text,
    : show a form, list jokes, etc.
  <?php endif; ?>

  </main>

  <footer>
  <?php include 'footer.html.php'; ?>
  </footer>
  </body>
</html>
```

But this method requires clairvoyance: we need to anticipate exactly what changes might need to be made in the future and use relevant `include` statements in the places we foresee changes will happen.

In the example above, for example, it's easy to add new menu entries by adding them to `nav.html.php`, but adding a `<script>` tag to every page, or even something as trivial as adding a CSS class to the `nav` element, still means opening every template to make the change.

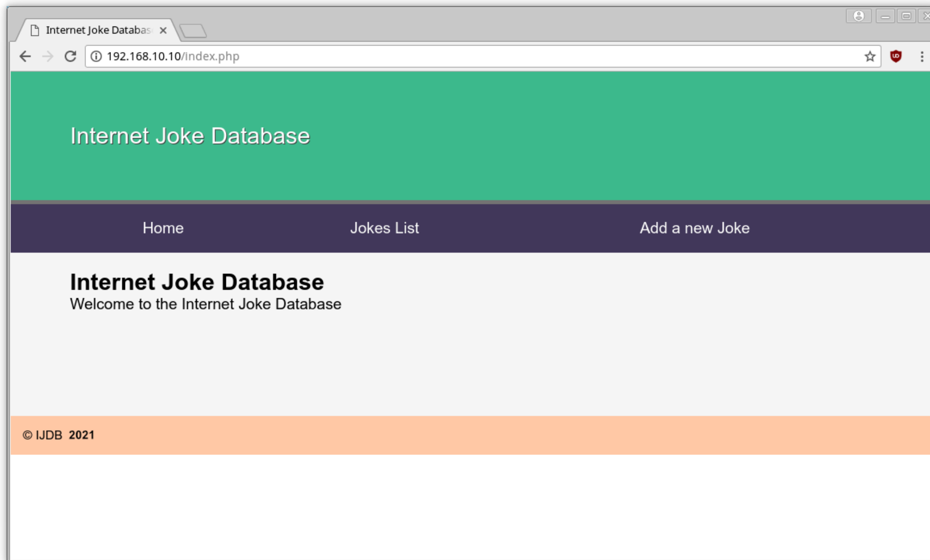
There's no way to accurately predict all the changes that might be needed

over the lifetime of the website, so instead the approach I showed you at the beginning of this chapter is actually better:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="jokes.css">
    <title><?=$title?></title>
  </head>
  <body>

  <header>
    <h1>Internet Joke Database</h1>
  </header>
  <nav>
    <ul>
      <li><a href="index.php">Home</a></li>
      <li><a href="jokes.php">Jokes List</a></li>
    </ul>
  </nav>

  <main>
  <?=$output?>
  </main>

  <footer>
  &copy; IJDB 2021
  </footer>
  </body>
</html>
```

If we always include this template, which we'll call `layout.html.php`, it's possible to set the `$output` variable to some *HTML code* and have it appear on the page with the navigation and footer. The benefit of this is that, to change the date on every page of the website, we'll only need to change it in one location.

I've also snuck in a `$title` variable so each controller can define a value that appears between the `<title>` and `</title>` tags along with some CSS (available as `jokes.css` in the sample code) to make the page a little prettier.



4-8. IJDB—now with CSS styles

Any controller can now use `include __DIR__ . '/../templates/layout.html.php';` and provide values for `$output` and `$title`.

Our `jokes.php` using `layout.html.php` is coded as shown below.

Example: MySQL-ListJokes-Layout-1[8]

```
<?php

try {
    $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
    ➥'mypassword');

    $sql = 'SELECT `joketext` FROM `joke`';
    $result = $pdo->query($sql);

    while ($row = $result->fetch()) {
        $jokes[] = $row['joketext'];
    }
```

8. https://github.com/spbooks/phpmysql7/tree/MySQL-ListJokes-Layout-1

```
    $title = 'Joke list';

    $output = '';

    foreach ($jokes as $joke) {
        $output .= '<blockquote>';
        $output .= '<p>';
        $output .= $joke;
        $output .= '</p>';
        $output .= '</blockquote>';
    }
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

But wait! What's going on with `$output` in the `try` block? The `$output` variable actually contains some HTML code: the loop builds a string containing the HTML code for the jokes list.

In principle, this is what we want to happen: the `$output` variable contains the HTML code that's going to be inserted between the navigation and the footer in `layout.html.php`, but I think you'll agree the code is incredibly ugly.

I already showed you how to avoid mixing HTML and PHP code via the `include` statement. Like we did earlier, it would be good to move the HTML for displaying the jokes to its own file—but this time, only the HTML code that's unique to the joke list page.

`jokes.html.php` in the `templates` directory should contain this code:

```
<?php foreach ($jokes as $joke): ?>
<blockquote>
  <p>
```

```
    <?=htmlspecialchars($joke, ENT_QUOTES, 'UTF-8')?>
    </p>
  </blockquote>
  <?php endforeach; ?>
```

Importantly, this is only the code for displaying the jokes. It doesn't contain the navigation, footer, `<head>` tag or anything we want repeated on every page; it's only the HTML code that's unique to the joke list page.

To use this template, you might try the following:

```
    while ($row = $result->fetch()) {
      $jokes[] = $row['joketext'];
    }

    $title = 'Joke list';

    include 'jokes.html.php';
  }
```

Or if you're very clever:

```
    while ($row = $result->fetch()) {
      $jokes[] = $row['joketext'];
    }

    $title = 'Joke list';

    $output = include 'jokes.html.php';
  }
```

With this approach, your logic would be entirely sound. We need to include `jokes.html.php`. Unfortunately, the `include` statement just executes the code from the included file at the point it's called. If you run the code above, the output will actually be something like this:

```
<blockquote>
  <p>
```

```
  A programmer was found dead in the shower. The instructions read: lather,
  ↪rinse, repeat.
  </p>
</blockquote>
<blockquote>
  <p>
  !false - it's funny because it's true
  </p>
</blockquote>
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Joke List</title>
  </head>
  <body>
  …
```

Because `jokes.html.php` is included first, it's sent to the browser first. What we need to do is load `jokes.html.php`, but instead of sending the output straight to the browser, we need to capture it and store it in the `$output` variable so that it can be used later by `layout.html.php`.

The `include` statement doesn't return a value, so `$output = include 'jokes.html.php';` doesn't have the desired effect, and PHP doesn't have an alternative statement to do that. However, that doesn't mean that it's not possible.

PHP does have a useful feature called "output buffering". It might sound complicated, but the concept is actually very simple: when you use `echo` to print something, or `include` to include a file that contains HTML, usually it's sent directly to the browser. By making use of output buffering, instead of having the output being sent straight to the browser, the HTML code is stored on the server in a "buffer", which is basically just a string containing everything that's been printed so far.

Even better, PHP lets you turn on the buffer and read its contents at any time.

There are two functions we need:

- `ob_start()`, which starts the output buffer. After calling this function, anything printed via `echo` or HTML printed via `include` will be stored in a buffer rather than sent to the browser.
- `ob_get_clean()`, which returns the contents of the buffer and clears it.

As you've probably guessed, "ob" in the function names stands for "output buffer".

To capture the contents of an included file, we just need to make use of these two functions:

```
while ($row = $result->fetch()) {
   $jokes[] = $row['joketext'];
}

$title = 'Joke list';

// Start the buffer

ob_start();

// Include the template. The PHP code will be executed,
// but the resulting HTML will be stored in the buffer
// rather than sent to the browser.

include  __DIR__ . '/../templates/jokes.html.php';

// Read the contents of the output buffer and store them
// in the $output variable for use in layout.html.php

$output = ob_get_clean();
}
```

When this code runs, the `$output` variable will contain the HTML that was generated in the `jokes.html.php` template.

We'll use this approach from now on. Each page will be made up of two

templates:

- `layout.html.php`, which contains all of the common HTML needed by every page
- a unique template that contains only the HTML code that's unique to that particular page

The complete `jokes.php` looks like this:

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  ↪ 'mypassword');

  $sql = 'SELECT `joketext` FROM `joke`';
  $result = $pdo->query($sql);

  while ($row = $result->fetch()) {
     $jokes[] = $row['joketext'];
  }

  $title = 'Joke list';

  ob_start();

  include  __DIR__ . '/../templates/jokes.html.php';

  $output = ob_get_clean();
}
catch (PDOException $e) {
  $title = 'An error has occurred';

  $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

Let's make the "Home" link work by adding an `index.php` file. We could put anything on this page: the latest jokes, the best joke of the month, or whatever we like. For now, though, we'll keep it simple and just have a message that says "Welcome to the Internet Joke Database".

Create a file called `home.html.php` in the `templates` folder:

```
<h2>Internet Joke Database</h2>

<p>Welcome to the Internet Joke Database</p>
```

Our `index.php` is considerably simpler than `jokes.html.php`. It doesn't get any information from the database, so it doesn't need a database connection and we don't need a `try … catch` statement, so we'll just load the two templates and set the `$title` and `$output` variables.

Example: MySQL-ListJokes-Layout-3[9]

```
<?php

$title = 'Internet Joke Database';

ob_start();

include  __DIR__ . '/../templates/home.html.php';

$output = ob_get_clean();

include  __DIR__ . '/../templates/layout.html.php';
```

Only Connect to the Database Where Necessary

It's good practice to only connect to the database if you need to. Databases are the most common performance bottleneck on many websites, so making as few connections as possible is preferred.

Test that both pages work in your browser. You should have a list of jokes visible when you visit `https://v.je/jokes.php` and the welcome message on `https://v.je/jokes.php`. Both pages should contain the navigation and the footer.

---

9. https://github.com/spbooks/phpmysql7/tree/MySQL-ListJokes-Layout-3

Try amending `layout.html.php`. The changes you make will appear on both pages. If the site had dozens of pages, changes to the layout would affect every page.

Inserting Data into the Database

In this section, I'll demonstrate how to use the tools at your disposal to enable site visitors to add their own jokes to the database.

If you want to let your site visitors enter new jokes, you'll obviously need a form. Here's a template for a form that will fit the bill:

```
<form action="" method="post">
  <label for="joketext">Type your joke here:</label>
  <textarea id="joketext" name="joketext" rows="3" cols="40">
  </textarea>
  <input type="submit" name="submit" value="Add">
</form>
```

Save this as `addjoke.html.php` in the `templates` directory.

The most important part of the `<form>` element is the `action` attribute. The `action` attribute tells the browser where to send the data once the form is submitted. This can be the name of a file, such as `"addjoke.php"`.

However, if you leave the attribute empty by setting it to `""`, the data provided by the user will be sent back to the page you're currently viewing. If the browser's URL shows the page as `addjoke.php`, that's were the data will be sent when the user presses the **Add** button.

Let's tie this form into the preceding example, which displayed the list of jokes in the database. Open up `layout.html.php` and add a URL to the “Add a new Joke” link that goes to `addjoke.php`:

```
<!doctype html>
<html>
```

```
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="jokes.css">
    <title><?=$title?></title>
  </head>
  <body>
  <nav>
    <header>
      <h1>Internet Joke Database</h1>
    </header>
    <ul>
      <li><a href="index.php">Home</a></li>
      <li><a href="jokes.php">Jokes List</a></li>
      <li><a href="addjoke.php">Add a new Joke</a></li>
    </ul>
  </nav>

  <main>
  <?=$output?>
  </main>

  <footer>
  &copy; IJDB 2021
  </footer>
  </body>
</html>
```

While you have `layout.html.php` open, include the `form.css` stylesheet from Chapter 2 as I have above. Now, any form displayed inside the layout will have the styles we used before.

When this form is submitted, the request will include a variable—`joketext`—that contains the text of the joke as typed into the text area. This variable will then appear in the `$_POST` array created by PHP.

Let's create `addjoke.php` in the `public` directory. The basic logic for this controller is:

- If no `joketext` POST variable is set, display a form.
- Otherwise, insert the supplied joke into the database.

Create this skeleton `addjoke.php`:

```
<?php
if (isset($_POST['joketext'])) {
    try {
        $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ↪'ijdbuser', 'mypassword');
    } catch (PDOException $e) {
        $title = 'An error has occurred';

        $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
    }
} else {
    $title = 'Add a new joke';

    ob_start();

    include  __DIR__ . '/../templates/addjoke.html.php';

    $output = ob_get_clean();
}
include  __DIR__ . '/../templates/layout.html.php';
```

This opening `if` statement checks if the `$_POST` array contains a variable called `joketext`. If it's set, the form has been submitted. Otherwise, the form from `addjoke.html.php` is loaded into the `$output` variable for displaying in the browser.

If you do open `addjoke.php` in your browser at this point, you'll see the form, but typing in a joke and pressing **Add** won't work, because we haven't yet done anything with the data contained in `$_POST['joketext']`.

To insert the submitted joke into the database, we must execute an `INSERT` query using the value stored in `$_POST['joketext']` to fill in the `joketext` column of the `joke` table. This might lead you to write some code like this:

```
$sql = 'INSERT INTO `joke` SET
  `joketext` ="' . $_POST['joketext'] . '"',
```

```
  `jokedate` ="2021-02-04"';

$pdo->exec($sql);
```

There's a serious problem with this code, however: the contents of `$_POST['joketext']` are entirely under the control of the user who submitted the form. If a malicious user were to type some nasty SQL code into the form, this script would feed it to your MySQL server without question. This type of attack is called an **SQL injection attack**, and in the early days of PHP it was one of the most common security holes that hackers found and exploited in PHP-based websites. (In many programming niches, SQL injection attacks are still surprisingly effective, as developers don't expect them. Consider this remarkable attempt to cause traffic cameras to drop their databases: "SQL Injection License Plate Hopes to Foil Euro Traffic Cameras"[10].)

A user might type this into the text box:

```
How many programmers does it take to screw in a lightbulb? None. It's a hardware
  ↪ problem.
```

The query sent to the database would be as follows:

```
INSERT INTO `joke` SET
  `joketext` ="How many programmers does it take to screw in a lightbulb? None.
  ↪It's a hardware problem.",
  `jokedate` ="2021-02-04
```

But what if the user types in the following:

```
A programmer's wife tells him to go to the store and "get a gallon of milk, and
↪if they have eggs, get a dozen." He returns with 13 gallons of milk.
```

In this case, the query sent to the database will be this:

---

10. https://gizmodo.com/sql-injection-license-plate-hopes-to-foil-euro-traffic-5498412

```
INSERT INTO `joke` SET
  `joketext`="A programmer's wife tells him to go to the store and "get a gallon
  ↪of milk, and if they have eggs, get a dozen." He returns with 13 gallons of
  ↪milk.",
  `jokedate`="2021-02-04
```

Because the joke contains a quote character, MySQL will return an error, as it will see the quote before “get” as the end of the string.

To make this a valid query, we need to escape all quotes in the text so that the query sent to the database becomes this:

```
INSERT INTO `joke` SET
  `joketext`="A programmer's wife tells him to go to the store and \"get a gallon
  ↪of milk, and if they have eggs, get a dozen.\" He returns with 13 gallons of
  ↪milk.",
  `jokedate`="2021-02-04
```

Data not being inserted if it contains a quote is an annoying problem for users. They’ll lose whatever they typed in. But malicious users are able to abuse this. In older versions of PHP, it was possible to run multiple queries from PHP by separating them with a semicolon (`;`).

Imagine if the user were to type this into the box:

```
"; DELETE FROM `joke`; --
```

This would send the following queries to the database:

```
INSERT INTO `joke` SET
        `joketext`="";

DELETE FROM `joke`;

--`jokedate`="2021-02-04
```

`--` is a single line comment in MySQL, so the last line would be ignored, and the `INSERT` query would run, followed by the `DELETE` query the user had

typed into the box. In fact, the user could type any query they liked into the box and it would be run on the database!

Magic Quotes

In the early days of PHP, SQL injection attacks were so feared that the team behind PHP added some built-in protections against SQL injections to the language. Firstly, they disabled the ability to send multiple queries at once. Secondly, they added something called **magic quotes**. This protective feature of PHP automatically analyzed all values submitted by the browser and inserted backslashes (`\`) in front of any “dangerous” characters like apostrophes—which can cause problems if they’re included in an SQL query inadvertently.

The problem with the magic quotes feature is that it caused as many problems as it prevented. First of all, the characters that it detected, and the method it used to sanitize them (prefixing them with a backslash), were only valid in some circumstances. Depending on the character encoding of your site and the database server you were using, these measures could be completely ineffective.

Second, when a submitted value was used for some purpose *other* than creating an SQL query, those backslashes could be really bothersome. The magic quotes feature would insert a spurious backslash into the user’s last name if it contained an apostrophe.

In short, the magic quotes feature was a bad idea—so much so that it was removed from PHP from version 5.4. However, due to PHP’s age and the amount of code out there, you might come across some references to it, so it’s worth having a basic understanding of what the magic quotes feature was supposed to do.

Once magic quotes was identified as a bad idea, the advice from PHP developers was to turn it off. However, this meant that there were some web servers with it turned off and others with it turned on. This was a headache for developers: they either had to instruct everyone who was ever going to use

their code to turn it off—which wasn't possible on some shared servers—or write extra code to account for it.

Most developers chose the latter, and you may come across some code like this:

```
if (get_magic_quotes_gpc()) {
  // code here
}
```

If you see an `if` statement like this in legacy code you've been given to work with, you can safely delete the entire block, as no code inside the `if` statement will ever be executed on recent PHP versions.

If you do see code like this, it means the original developer understood the problems with magic quotes and was doing their best to prevent it. As of PHP 5.4 (which you should never come across, as it's no longer supported), `get_magic_quotes_gpc()` will always return `false` and the code will never be executed.

All you really need to know about magic quotes is that it was a bad solution to the problem at hand. Of course, without magic quotes, you need to find a different solution to the problem. Luckily, the PDO class can do all the hard work for you, by using something called "prepared statements".

Prepared Statements

A **prepared statement** is a special kind of SQL query that you've sent to your database server ahead of time, giving the server a chance to prepare it for execution—but not actually execute it. Think of it like writing a `.php` script. The code is there, but it doesn't actually get run until you visit the page in your web browser. The SQL code in prepared statements can contain placeholders that you'll supply the values for later, when the query *is* to be executed. When filling in these placeholders, PDO is smart enough to guard against "dangerous" characters automatically.

Here's how to prepare an `INSERT` query and then execute it safely with `$_POST['joketext']` as the text of the joke:

```
$sql = 'INSERT INTO `joke` SET
  `joketext` = :joketext,
  `jokedate` = "today's date"';

$stmt = $pdo->prepare($sql);

$stmt->bindValue(':joketext', $_POST['joketext']);
$stmt->execute();
```

Let's break this down, one statement at a time. First, we write our SQL query as a PHP string and store it in a variable (`$sql`) as usual. What's unusual about this `INSERT` query, however, is that no value is specified for the `joketext` column. Instead, it contains a placeholder for this value (`:joketext`). Don't worry about the `jokedate` field just now; we'll circle back to it in a moment.

Next, we call the prepare method of our PDO object (`$pdo`), passing it our SQL query as an argument. This sends the query to the MySQL server, asking it to *prepare* to run the query. MySQL can't run it yet, as there's no value for the `joketext` column. The `prepare` method returns a `PDOStatement` object (yes, the same kind of object that gives us the results from a `SELECT` query), which we store in `$stmt` .

Now that MySQL has prepared our statement for execution, we can send it the missing value(s) by calling the `bindValue` method of our `PDOStatement` object (`$stmt`). We call this method once for each value to be supplied (in this case, we only need to supply one value—the joke text), passing as arguments the placeholder that we want to fill in (`':joketext'`) and the value we want to fill it with (`$_POST['joketext']`). Because MySQL knows we're sending it a discrete value, rather than SQL code that needs to be parsed, there's no risk of characters in the value being interpreted as SQL code. When using prepared statements, SQL injection vulnerabilities simply aren't possible!

Finally, we call the `PDOStatement` object's `execute` method to tell MySQL to

execute the query with the value(s) we've supplied. (Yes, this `PDOStatement` method is called `execute`, unlike the similar method of `PDO` objects, which is called `exec`. PHP has many strengths, but consistency isn't one of them!)

One interesting thing you'll notice about this code is that we never placed quotes around the joke text. `:joketext` exists inside the query without any quotes, and when we called `bindValue` we passed it the plain joke text from the `$_POST` array. When using prepared statements, you don't need quotes because the database (in our case, MySQL) is smart enough to know that the text is a string and it will be treated as such when the query is executed.

The lingering question in this code is how to assign today's date to the `jokedate` field. We *could* write some fancy PHP code to generate today's date in the `YYYY-MM-DD` format that MySQL requires, but it turns out that MySQL itself has a function to do this—`CURDATE`:

```
$sql = 'INSERT INTO `joke` SET
  `joketext` = :joketext,
  `jokedate` = CURDATE()';

$stmt = $pdo->prepare($sql);
$stmt->bindValue(':joketext', $_POST['joketext']);
$stmt->execute();
```

The MySQL `CURDATE` function is used here to assign the current date as the value of the `jokedate` column. MySQL actually has dozens of these functions, but I'll introduce them only as required.

Now that we have our query, we can complete the `if` statement we started earlier to handle submissions of the "Add Joke" form:

```
if (isset($_POST['joketext'])) {
  try {
    $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
    ➥ 'mypassword');
```

```
    $sql = 'INSERT INTO `joke` SET
      `joketext` = :joketext,
      `jokedate` = CURDATE()';

    $stmt = $pdo->prepare($sql);

    $stmt->bindValue(':joketext', $_POST['joketext']);

    $stmt->execute();
  }
  catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
  }
}
```

But wait! This `if` statement has one more trick up its sleeve. Once we've added the new joke to the database, instead of displaying the PHP template as previously, we want to redirect the user's browser back to the list of jokes. That way, users are able to see the newly added joke among them. That's what the two lines at the end of the `if` statement above do.

In order to achieve the desired result, your first instinct might be to allow the controller to simply fetch the list of jokes from the database after adding the new joke and displaying the list using the `jokes.html.php` template as usual. The problem with doing this is that the list of jokes, from the browser's perspective, would be the result of having submitted the "Add Joke" form. If the user were then to refresh the page, the browser would resubmit that form, causing another copy of the new joke to be added to the database! This is rarely the desired behavior.

Instead, we want the browser to treat the updated list of jokes as a normal web page that's able to be reloaded without resubmitting the form. The way to do this is to answer the browser's form submission with an HTTP redirect—a special response that tells the browser to navigate to a different page. (HTTP stands for HyperText Transfer Protocol, and is the language that describes the request/response communications that are exchanged between the visitor's

web browser and your web server.)

The PHP `header` function provides the means of sending special server responses like this one, by letting you insert specific headers into the response sent to the browser. In order to signal a redirect, you must send a `Location` header with the URL of the page you want to direct the browser to:

```
header('Location: URL');
```

In this case, we want to send the browser to `jokes.php`. Here are the two lines that redirect the browser back to our controller after adding the new joke to the database:

```
header('Location: jokes.php');
```

Below is the complete code of the `addjoke.php` controller.

Example: MySQL-AddJoke[11]

```
<?php
if (isset($_POST['joketext'])) {
    try {
        $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ↪'ijdbuser', 'mypassword');

        $sql = 'INSERT INTO `joke` SET
      `joketext` = :joketext,
      `jokedate` = CURDATE()';

        $stmt = $pdo->prepare($sql);

        $stmt->bindValue(':joketext', $_POST['joketext']);

        $stmt->execute();

        header('location: jokes.php');
```

11. https://github.com/spbooks/phpmysql7/tree/MySQL-AddJoke

```
    } catch (PDOException $e) {
        $title = 'An error has occurred';

        $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
    }
} else {
    $title = 'Add a new joke';

    ob_start();

    include  __DIR__ . '/../templates/addjoke.html.php';

    $output = ob_get_clean();
}
include  __DIR__ . '/../templates/layout.html.php';
```

As you review this to ensure it all makes sense to you, note that the code that connects to the database by creating a `new PDO` object must come before any of the code that runs database queries. But a database connection isn't required for displaying the "Add Joke" form. The connection is only made when the form has been submitted.

Load this up and add a new joke or two to the database via your browser.

There you have it: you're able to view existing jokes in—and add new jokes to—your MySQL database.

Deleting Data from the Database

In this section, we'll make one final enhancement to our joke database site. Next to each joke on the jokes page (`jokes.php`), we'll place a button labeled **Delete**. When clicked, it will remove that joke from the database and display the updated joke list.

If you like a challenge, you might want to take a stab at writing this feature yourself before you read on to see my solution. Although we're implementing a brand new feature, we'll mainly be using the same tools as employed in the

previous examples in this chapter. Here are a few hints to start you off:

- You'll need a new controller (`deletejoke.php`).
- The SQL `DELETE` command will be required, which I introduced in Chapter 3.
- To delete a particular joke in your controller, you'll need to identify it uniquely. The `id` column in the `joke` table was created to serve this purpose. You're going to have to pass the ID of the joke to be deleted with the request to delete a joke. The easiest way to do this is to use a hidden form field.

At the very least, take a few moments to think about how you'd approach this. When you're ready to see the solution, read on!

To begin with, we need to modify the `SELECT` query that fetches the list of jokes from the database. In addition to the `joketext` column, we must also fetch the `id` column so that we can identify each joke uniquely:

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  ↪ 'mypassword');

  $sql = 'SELECT `id`, `joketext` FROM `joke`';
  $result = $pdo->query($sql);
  // …
```

We also have to modify the `while` loop that stores the database results into the `$jokes` array. Instead of simply storing the text of each joke as an item in the array, we store both the ID and text of each joke. One way to do this is to make each item in the `$jokes` array an array in its own right:

```
while ($row = $result->fetch()) {
  $jokes[] = ['id' => $row['id'], 'joketext' => $row['joketext']];
}
```

Using a `foreach` Loop Instead

If you've already switched to using a `foreach` loop to process your database result rows, that will work just fine too:

```
foreach ($result as $row) {
  $jokes[] = array('id' => $row['id'], 'joketext' =>
    $row['joketext']);
}
```

Once this loop runs its course, we'll have the `$jokes` array, each item of which is an associative array with two items: the ID of the joke and its text. For each joke (`$jokes[n]`), we can therefore retrieve its ID (`$jokes[n]['id']`) and its text (`$jokes[n]['text']`).

Our next step is to update the `jokes.html.php` template to retrieve each joke's text from this new array structure, as well as provide a **Delete** button for each joke:

```
<?php foreach ($jokes as $joke): ?>
<blockquote>
  <p>
  <?=htmlspecialchars($joke['joketext'], ENT_QUOTES, 'UTF-8')?>
  <form action="deletejoke.php" method="post">
    <input type="hidden" name="id" value="<?=$joke['id']?>">
    <input type="submit" value="Delete">
  </form>
  </p>
</blockquote>
<?php endforeach; ?>
```

Here are the highlights of this updated code:

- Each joke will be displayed with a form, which, if submitted, will delete that joke. We signal this to a new controller, `deletejoke.php` , using the form's `action` attribute.

- Since each joke in the `$jokes` array is now represented by a two-item array instead of a simple string, we must update this line to retrieve the text of the joke. We do this using `$joke['text']` instead of just `$joke`.

- When we submit the form to delete this joke, we send along the ID of the joke to be deleted. To do this, we need a form field containing the joke's ID, but we'd prefer to keep this field hidden from the user; that's why we use a hidden form field (`input type="hidden"`). The name of this field is `id`, and its value is the ID of the joke to be deleted (`$joke['id']`).

  Unlike the text of the joke, the ID is not a user-submitted value, so there's no need to worry about making it HTML-safe with `htmlspecialchars`. We can rest assured it will be a number, since it's automatically generated by MySQL for the `id` column when the joke is added to the database.

- The submit button (`input type="submit"`) submits the form when clicked. Its value attribute gives it a label of **Delete**.

- Finally, we close the form for this joke.

Why Aren't the Form and Input Tags Outside the Blockquote?

If you know your HTML, you're probably thinking the form and input tags belong outside of the blockquote element, since they aren't a part of the quoted text (the joke).

Strictly speaking, that's true: the form and its inputs should really be either before or after the blockquote. Unfortunately, making that tag structure display clearly requires a little CSS (cascading style sheets) code that's really beyond the scope of this book.

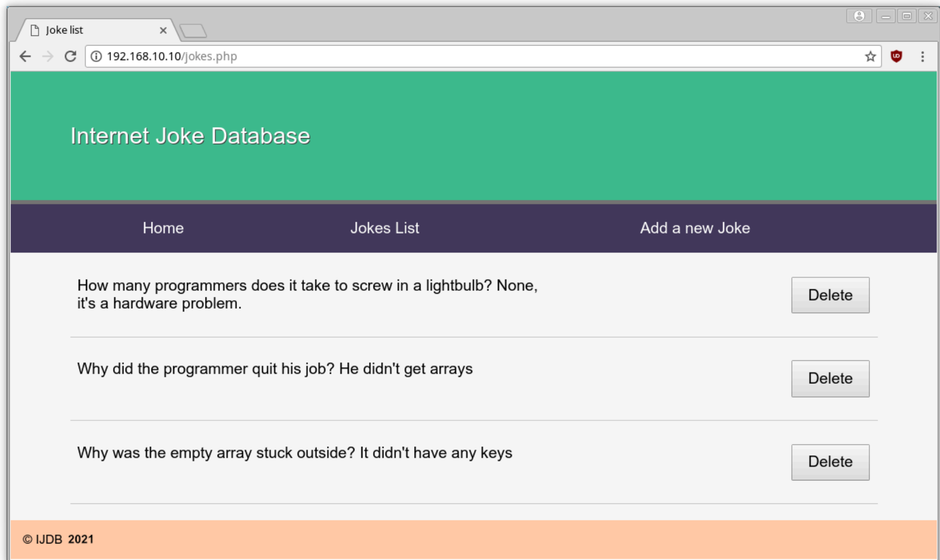
Rather than teach you CSS layout techniques in a book about PHP and MySQL, I've decided to go with this imperfect markup. If you plan to use this code in the real world, you should invest some time into learning CSS (or at least secure the services of a CSS guru). That way, you can take complete control of your HTML markup without worrying about the CSS required to make it look nice. If you want to learn more about CSS layouts, take a look at *CSS Master, 3rd Edition*[12], by Tiffany Brown.

Add the following CSS to `jokes.css` to make the buttons appear to the right of the jokes and draw a line between them:

```
blockquote {display: table; margin-bottom: 1em; border-bottom: 1px solid #ccc;
  ↪ padding: 0.5em;}
blockquote p {display: table-cell; width: 90%; vertical-align: top;}
blockquote form {display: table-cell; width: 10%;}
```

The following image shows what the joke list looks like with the **Delete** buttons added.

12. https://www.sitepoint.com/premium/books/css-master-3rd-edition"



4-9. Each button can delete its respective joke

But wait! Before we move on to making the **Delete** button work, let's briefly step back and take a careful look at this line:

```
$jokes[] = ['id' => $row['id'], 'joketext' => $row['joketext']];
```

Here, we're looping over the `PDOStatement` object, which gives us a `$row` variable containing the keys `id` and `joketext` along with corresponding values, and we're using that to build another array with the same keys and values.

You may have already realized this is terribly inefficient. We could achieve the same thing using this code:

```
while ($row = $result->fetch()) {
  $jokes[] = $row;
}
```

But as we know, this can also be achieved with a `foreach` loop:

```
foreach ($result as $row) {
 $jokes[] = $row;
}
```

In this instance, we're using `foreach` to iterate over the records from the database and build an array. We're then looping over the array with another `foreach` loop in the template. We could just write this:

```
$jokes = $result;
```

Now, when `$jokes` is iterated over in the template, it's not an array but a `PDOStatement` object. However, that has no effect on the output and saves us some code. In fact, we can omit the `$result` variable altogether and load the `PDOStatement` object directly into the `$jokes` variable. The complete `jokes.php` controller now looks like this:

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  ↪ 'mypassword');

  $sql = 'SELECT `joketext`, `id` FROM joke';

  $jokes = $pdo->query($sql);

  $title = 'Joke list';

  ob_start();

  include  __DIR__ . '/../templates/jokes.html.php';

  $output = ob_get_clean();
}
catch (PDOException $e) {
  $title = 'An error has occurred';

  $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}
```

```
include  __DIR__ . '/../templates/layout.html.php';
```

Now we don't even have a `while` loop iterating over the records in the controller, but just iterate over the records directly in the template, saving some code and making the page execute slightly faster, as it now only loops over the records once.

Back to our new **Delete** button: all that remains to make this new feature work is to add a relevant `deletejoke.php` to issue a `DELETE` query to the database:

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  ➥ 'mypassword');

  $sql = 'DELETE FROM `joke` WHERE `id` = :id';

  $stmt = $pdo->prepare($sql);

  $stmt->bindValue(':id', $_POST['id']);
  $stmt->execute();

  header('location: jokes.php');
}
catch (PDOException $e) {
  $title = 'An error has occurred';

  $output = 'Unable to connect to the database server: ' . $e->getMessage() .
  ➥ ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

The complete code for the updated `jokes.php` and `deletejoke.php` is available as **Example: MySQL-DeleteJoke**[13].

This chunk of code works exactly like the one we added to process the "Add Joke" code earlier in the chapter. We start by preparing a `DELETE` query with a

13. https://github.com/spbooks/phpmysql7/tree/MySQL-DeleteJoke

placeholder for the joke ID that we want to delete.

Prepared Statements Necessary?

You might think that a prepared statement is unnecessary in this instance to protect our database from SQL injection attacks, since the joke ID is provided by a hidden form field invisible to the user. In fact, *all* form fields—even hidden ones—are ultimately under the user's control. There are widely distributed browser add-ons, for example, that will make hidden form fields visible and available for editing by the user. Remember: any value submitted by the browser is ultimately suspect when it comes to protecting your site's security.

We then bind the submitted value of `$_POST['id']` to that placeholder and execute the query. Once that query is achieved, we use the PHP `header` function to ask the browser to send a new request to view the updated list of jokes.

Don’t Use Hyperlinks to Perform Actions

If you tackled this example yourself, your first instinct might have been to provide a **Delete** hyperlink for each joke, instead of going to the trouble of writing an entire HTML form containing a **Delete** button for each joke on the page. Indeed, the code for such a link would be much simpler:

```
<?php foreach ($jokes as $joke): ?>
  <blockquote>
    <p>
      <?=htmlspecialchars($joke['joketext'], ENT_QUOTES, 'UTF-8')?>
      <a href="deletejoke.php&amp;id=<?=$joke['id']?>">Delete</a>
    </p>
  </blockquote>
<?php endforeach; ?>
```

In short, hyperlinks should never be used to perform *actions* (such as deleting a joke). They must only be used to provide a link to some related content. The same goes for forms with `method="get"`, which should only be used to perform queries of existing data. Actions must only ever be performed as a result of a form with `method="post"` being submitted.

The reason for this is that forms with `method="post"` are treated differently by browsers and related software. If you were to submit a form with `method="post"` and then click the refresh button in your browser, for example, the browser would ask if you’re certain you want to resubmit the form. Browsers have no similar protection against resubmission when it comes to links and forms with `method="get"`.

Search engines and other web crawlers will also follow all the links on your site in order to work out when to show your site’s pages in search results. If your site deleted a joke as a result of a hyperlink being followed, you could find your jokes being deleted whenever a search engine finds your site.

Mission Accomplished

In this chapter, you learned all about PHP Data Objects (PDO), a collection of built-in PHP classes (`PDO` , `PDOException` , and `PDOStatement`) that allow you to interface with a MySQL database server by creating objects and then calling the methods they provide. While you were at it, you also picked up the basics of object-oriented programming (OOP)—which is no mean feat for a PHP beginner!

Using PDO objects, you built your first database-driven website, which published the `ijdb` database online and allowed visitors to add and delete jokes.

In a way, you could say this chapter achieved the stated mission of this book: to teach you how to build a database-driven website. Of course, the example in this chapter contained only the bare essentials. In the rest of the book, I'll show you how to flesh out the skeleton you learned to build in this chapter.

In Chapter 5, we'll return to the SQL Query window in MySQL Workbench. We'll learn how to use relational database principles and advanced SQL queries to represent more complex types of information, and give our visitors credit for the jokes they add!

Chapter 5

Relational Database Design

Since Chapter 3, we've worked with a very simple database of jokes, composed of a single table named (appropriately enough) `joke`. While this database has served us well as an introduction to MySQL databases, there's more to relational database design than can be understood from this simple example. In this chapter, we'll expand on this database and learn a few new features of MySQL, in an effort to realize and appreciate the real power that relational databases have to offer.

Be forewarned that I'll cover several topics only in an informal, non-rigorous sort of way. As any computer science major will tell you, database design is a serious area of research, with tested and mathematically provable principles that, while useful, are beyond the scope of this book.

For more complete coverage of database design concepts and SQL in general, pick up a copy of *Jump Start MySQL*[1].

Giving Credit Where Credit Is Due

To start off, let's recall the structure of our `joke` table. It contains three columns: `id`, `joketext`, and `jokedate`. Together, these columns allow us to identify jokes (`id`), and keep track of their text (`joketext`) and the date they were entered (`jokedate`). For your reference, here's the SQL code that creates this table and inserts a couple of entries:

```
# Code to create a simple joke table

CREATE TABLE `joke` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `joketext` TEXT,
  `jokedate` DATE NOT NULL
) DEFAULT CHARACTER SET utf8 ENGINE=InnoDB;

# Adding jokes to the table

INSERT INTO `joke` SET
```

1. https://www.sitepoint.com/premium/books/jump-start-mysql

```
`joketext` = 'Why was the empty array stuck outside? It didn\'t have any keys',
`jokedate` = '2021-04-01';

INSERT INTO `joke`
(`joketext`, `jokedate`) VALUES (
'!false - It\'s funny because it\'s true',
"2021-04-01"
);
```

Recreating a Database from Scratch

If you ever need to re-create your database from scratch, you can use MySQL Workbench to drop all the tables and then go to Data Import/Restore and select the `database.sql` from the sample code. Note that the `database.sql` file will change depending on which sample you're viewing. In this way, you can use the `.sql` files in this book's code archive as database snapshots to load up whenever you need them.

Now, let's say we wanted to track another piece of information about our jokes: the names of the people who submitted them. It would be natural to add a new column to our `joke` table for this. The SQL `ALTER TABLE` command (which we've yet to see) lets us do exactly that.

As I demonstrated earlier, you can either type out these queries yourself or have a tool such as MySQL Workbench do it for you. When you use the GUI of MySQL Workbench to interact with the database, it generates the queries for you. It even shows them to you before applying them to the database. If you're not quite sure how to write a query, you can always get MySQL Workbench to generate it for you and then amend it to suit your needs.

SQL Queries fall into two categories:

- **Data definition language (DDL) queries**. These are the queries that describe *how* the data will be stored. These are the `CREATE TABLE` and

`CREATE DATABASE` queries that I showed you in Chapter 3, along with the aforementioned `ALTER TABLE` query.
- **Data manipulation language (DML) queries**. These are the queries that you use to manipulate the data in the database. You've seen some of these already: `INSERT`, `UPDATE`, `DELETE` and `SELECT`.

It's worthwhile for a PHP developer to learn the syntax and different variations of the DML queries, as they regularly need to be typed out in PHP scripts. It's also useful to know what's going on behind the scenes. That said, it's not going to be detrimental to your progress as a developer to have MySQL Workbench generate your DDL queries rather than typing them out yourself. Having the queries generated for you can be a time saver, as it gives you a lot less syntax learn, and the DDL query format can be considerably more difficult to follow than most of the DML queries.

To add a new column to a database table, open up MySQL Workbench as you did in Chapter 4, connect to your database, double-click on the schema we created called `ijdb`, and expand the Tables entry. You'll see the `joke` table.

If You've Logged Out or Rebooted

If you've logged out or rebooted your PC since the last chapter, you'll need to boot your server as you did in Chapter 1 using the `docker-compose up` command.

To add a new column, right-click on the table name and select **Alter Table**. This will bring up the familiar table editing screen. From here, you can add columns in the same way you did in Chapter 3.

Add a new column called `authorname`, which is going to store the name of the joke's author along with each joke. Set the type to `VARCHAR(255)`. The type declared is a **variable-length character string** of up to 255 characters, `VARCHAR(255)`—which is plenty of space for even very esoteric names. Let's also add a column for the authors' email addresses, set the column name to `authoremail`, and the type to `VARCHAR(255)`.

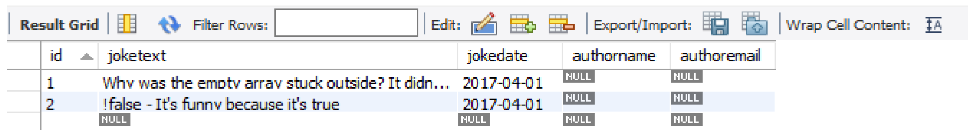
Once you press **Apply**, you'll see a confirmation dialog appear with the following DDL queries:

```
ALTER TABLE `joke` ADD COLUMN `authorname` VARCHAR(255)
ALTER TABLE `joke` ADD COLUMN `authoremail` VARCHAR(255)
```

You could have typed these in yourself, but the GUI in MySQL Workbench provides some useful error checks, and it will always generate valid queries.

Just to make sure the two columns were added properly, right-click on the table name in the SCHEMAS panel and select **Rows - Limit 1000**. You should see the two extra columns listed. Of course, at the moment, none of your jokes have values for either of these fields.

This should give you a table of results like the one pictured below.



5-1. Our joke table now contains five columns

Looks good, right? Obviously, to accommodate this expanded table structure, we'd need to make changes to the HTML and PHP form code we wrote in Chapter 4 that allowed us to add new jokes to the database. Using `UPDATE` queries, we could now add author details to all the jokes in the table. But before we spend too much time on such changes, we should stop and consider whether this new table design was the right choice here. In this case, it turns out it wasn't.

Rule of Thumb: Keep Entities Separate

As your knowledge of database-driven websites continues to grow, you may decide that a personal joke list is too limited. In fact, you might receive more submitted jokes than you have original jokes of your own. Let's say you decide

to launch a website where people from all over the world can share jokes with each other. Adding the author's name and email address to each joke certainly makes a lot of sense, but the method we used above leads to potential problems:

- What if a frequent contributor to your site named Joan Smith changed her email address? She might begin to submit new jokes using the new address, but her old address would still be attached to the jokes she'd submitted in the past. Looking at your database, you might simply think there were two people named Joan Smith who had submitted jokes. She might inform you of the change of address, and you may try to update all the old jokes with the new address, but if you missed just one joke, your database would still contain incorrect information. Database design experts refer to this sort of problem as an **update anomaly**.

- It would be natural for you to rely on your database to provide a list of all the people who've ever submitted jokes to your site. In fact, you could easily obtain a mailing list using the following query:

  ```
  SELECT DISTINCT `authorname`, `authoremail`
  FROM `joke`
  ```

  The word `DISTINCT` in the above query stops MySQL from outputting duplicate result rows. For example, if Joan Smith submits 20 jokes to your site, using the `DISTINCT` option would cause her name to only appear once in the list instead of 20 times.

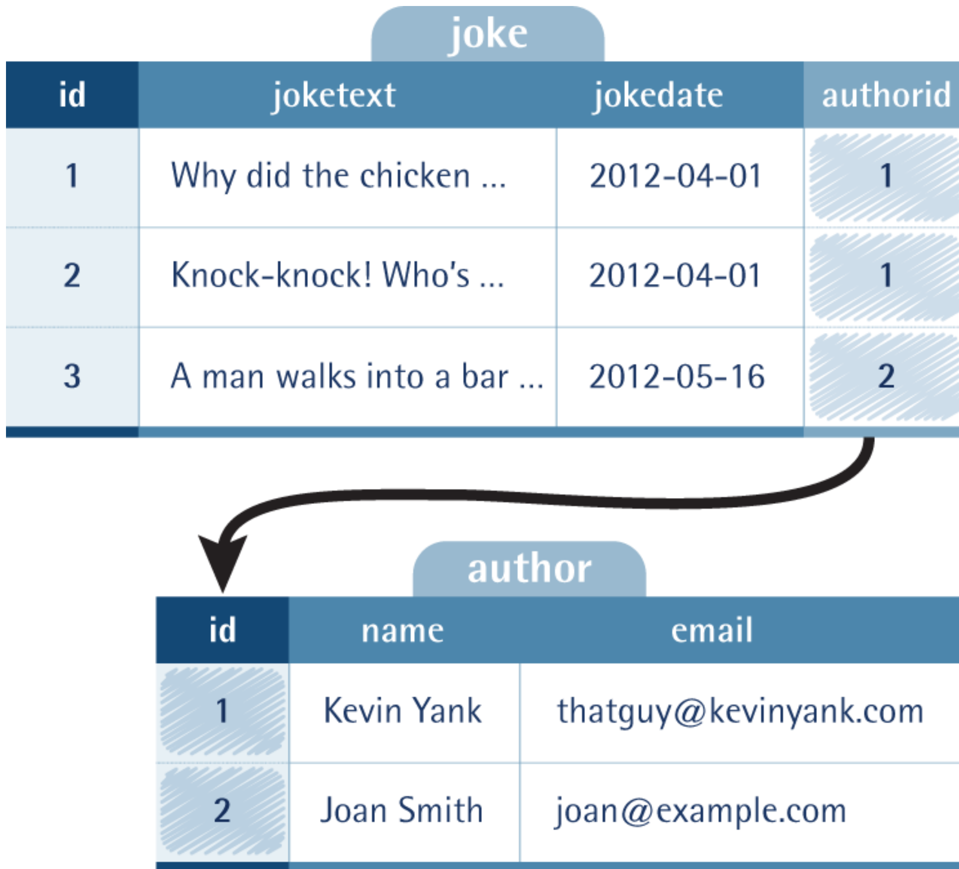
  Then, if for some reason you decided to remove all the jokes that a particular author had submitted to your site, you'd remove any record of this person from the database in the process, and you'd no longer be able to email them with information about your site! Database design experts call this a **delete anomaly**.

  As your mailing list might be a major source of income for your site, it's unwise to go throwing away an author's email address just because you

disliked the jokes that person submitted.

- You have no guarantee that Joan Smith will enter her name the same way each time. Consider the variations: Joan Smith, J. Smith, Smith, Joan—you catch my drift. This makes keeping track of a particular author exceedingly difficult, especially if Joan Smith also has several email addresses she likes to use.

These problems—and more—can be dealt with very easily using established database design principles. Instead of storing the information for the authors in the same table as the jokes, let's create an entirely new table for our list of authors. Just as we have an `id` column in our `joke` table to identify each joke with a unique number, we'll use an identically named column in our new table to identify our authors. We can then use those author IDs in our `joke` table to associate authors with their jokes. The complete database layout is pictured below.



5-2. The relationship between the joke and author tables

These tables show that there are three jokes and two authors. The `authorid` column of the `joke` table establishes a relationship between the two tables, indicating that Kevin Yank submitted jokes 1 and 2 and Joan Smith submitted joke 3. Notice that, since each author now only appears once in the database, and independently of the jokes submitted, we've avoided all the potential problems just outlined.

What's really important to note about this database design is that we're storing information about two types of *things* (jokes and authors), so it's most appropriate to have two tables. This is a rule of thumb that you should always keep in mind when designing a database: each type of entity (or "thing") about which you want to be able to store information *should be given its own table*.

To set up the aforementioned database from scratch is fairly simple (involving just two `CREATE TABLE` queries), but since we'd like to make these changes in a nondestructive manner (that is, without losing any of our precious jokes), we'll use MySQL Workbench to remove the `authorname` and `authoremail` columns from the `joke` table. To do this, right-click on the `joke` table in the SCHEMAS list and select **Alter Table**. Once again, it will give you an editable grid containing all the columns in the table. To delete the two columns, right-click on the column name and select **Delete Selected**. You'll need to do this for both columns. Once you click **Apply**, you'll see MySQL Workbench has generated this query for you:

```
ALTER TABLE `ijdb`.`joke`
DROP COLUMN `authoremail`,
DROP COLUMN `authorname`;
```

This is a DDL `ALTER TABLE` query for removing columns. As with all of these DDL queries, you could have typed this into the query panel manually and executed it, but we've used the GUI to avoid having to remember all the different commands.

Now, we need to create a new table to store the authors. To do this, follow the same procedure you used to create the `joke` table: right-click on the Tables entry in the SCHEMAS panel and select **Create Table**.

Set the table name to `author` and add the following fields:

- `id`, and check the `PK`, `AI` and `NN` boxes
- `name`, `VARCHAR(255)`
- `email`, `VARCHAR(255)`

Click **Apply**, and MySQL Workbench will generate a `CREATE TABLE` query similar to this:

```
CREATE TABLE `author` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
```

```
  `name` VARCHAR(255),
  `email` VARCHAR(255)
) DEFAULT CHARACTER SET utf8 ENGINE=InnoDB
```

Finally, we add the `authorid` column to our `joke` table. Edit the `joke` table and add a column called `authorid` with the type `INT`.

If you prefer, here are the `CREATE TABLE` commands that will create the two tables from scratch:

```
# Code to create a simple joke table that stores an author ID

CREATE TABLE `joke` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `joketext` TEXT,
  `jokedate` DATE NOT NULL,
  `authorid` INT
) DEFAULT CHARACTER SET utf8 ENGINE=InnoDB;

# Code to create a simple author table

CREATE TABLE `author` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `name` VARCHAR(255),
  `email` VARCHAR(255)
) DEFAULT CHARACTER SET utf8 ENGINE=InnoDB;
```

All that's left to do is add some authors to the new table, and assign authors to all the existing jokes in the database by filling in the `authorid` column. (For now, you'll have to do this manually, but rest assured that, in Chapter 8, we'll see how PHP can insert entries with the correct IDs automatically, reflecting the relationships between them.) Go ahead and do this now if you like, as it will give you some practice with `INSERT` and `UPDATE` queries. If you're rebuilding the database from scratch, however, here's a series of `INSERT` queries that will do the trick:

```
# Adding authors to the database
# We specify the IDs so they're known when we add the jokes below.
```

```
INSERT INTO `author` SET
  `id` = 1,
  `name` = 'Kevin Yank',
  `email` = 'thatguy@kevinyank.com';

INSERT INTO `author` (`id`, `name`, `email`)
VALUES (2, 'Tom Butler', 'tom@r.je');

# Adding jokes to the database

INSERT INTO `joke` SET
  `joketext` = 'How many programmers does it take to screw in a lightbulb?
  ↪ None, it\'s a hardware problem.',
  `jokedate` = '2021-04-01',
  `authorid` = 1;

INSERT INTO `joke` (`joketext`, `jokedate`, `authorid`)
VALUES (
  'Why did the programmer quit his job? He didn\'t get arrays',
  '2021-04-01',
  1
);

INSERT INTO `joke` (`joketext`, `jokedate`, `authorid`)
VALUES (
  'Why was the empty array stuck outside? It didn\'t have any keys',
  '2021-04-01',
  2
);
```

Both Kinds of `INSERT`

I've used this opportunity to refresh your memory about both kinds of `INSERT` query syntax. They both do exactly the same job and have the same result, so it's up to you which you use, and boils down mainly to personal preference rather than any practical reason.

`SELECT` with Multiple Tables

With your data now separated into two tables, it may seem that you're complicating the process of data retrieval. Consider, for example, our original goal: to display a list of jokes with the name and email address of the author next to each joke. In the single-table solution, you could gain all the information needed to produce such a list using a single `SELECT` query in your PHP code:

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  ➥ 'mypassword');
  $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

  $sql = 'SELECT `id`, `joketext` FROM `joke`';

  $jokes = $pdo->query($sql);

  $title = 'Joke list';

  ob_start();

  include  __DIR__ . '/../templates/jokes.html.php';

  $output = ob_get_clean();
}
catch (PDOException $e) {
  $title = 'An error has occurred';

  $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

With our new database layout, this would, at first, no longer seem possible. As the author details of each joke are no longer stored in the `joke` table, you might think that you'd have to fetch those details separately for each joke you wanted to display. The code required would involve a call to the PDO query method for each and every joke to be displayed. This would be messy and

involve a considerable amount of extra code.

Taking all this into account, it would seem that the “old way” was the better solution, despite its weaknesses. Fortunately, relational databases like MySQL are designed to make it easy to work with data stored in multiple tables! Using a new form of the `SELECT` statement, called a **join**, you can have the best of both worlds. Joins allow you to treat related data in multiple tables as if they were stored in a single table. Here’s what the syntax of a simple join looks like:

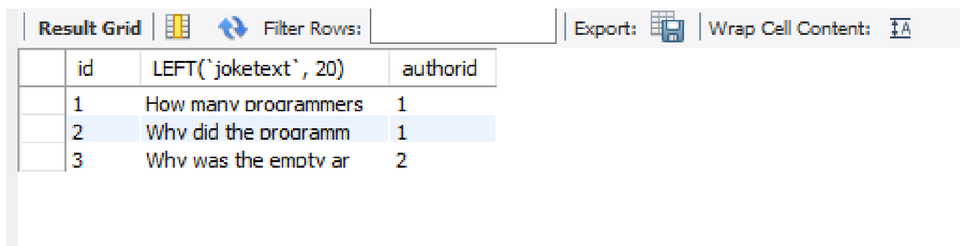
```
SELECT columns
FROM `table1`
INNER JOIN `table2`
  ON condition(s) for data to be related
```

In your case, the columns we’re interested in are `id` and `joketext` in the `joke` table, and `name` and `email` in the `author` table. The condition for an entry in the `joke` table to be related to an entry in the `author` table is that the value of the `authorid` column in the `joke` table is equal to the value of the `id` column in the `author` table.

Let’s look at an example of a join. The first two queries show you what’s contained in the two tables; they’re unnecessary to perform the join. The third query is where the action’s at:

```
SELECT `id`, LEFT(`joketext`, 20), `authorid` FROM `joke`
```

This query should now give the results shown below.



id	LEFT(`joketext`, 20)	authorid
1	How many programmers	1
2	Why did the programm	1
3	Why was the empty ar	2

5-3. Results of the joke table query

```
SELECT * FROM `author`
```

And this query, as you'd expect, will show all the authors.

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

id	name	email
1	Kevin Yank	thatguy@kevinyank.com
2	Tom Butler	tom@r.ie
NULL	NULL	NULL

5-4. All of the authors from the authors table

It's possible to query data from both tables by using the SQL `JOIN` statement:

```
SELECT `joke`.`id`, LEFT(`joketext`, 20), `name`, `email`
FROM `joke` INNER JOIN `author`
  ON `authorid` = `author`.`id`
```

This will display all the data from both tables.

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	LEFT(`joketext`, 20)	name	email
1	How many programmers	Kevin Yank	thatguy@kevinyank.com
2	Why did the programm	Kevin Yank	thatguy@kevinyank.com
3	Why was the empty ar	Tom Butler	tom@r.ie

5-5. The results of your first join

See? The results of the third `SELECT` —a join—group the values stored in the two tables into a single table of results, with related data correctly appearing together. Even though the data is stored in two tables, you can still access all the information you need to produce the joke list on your web page with a single database query. Note in the query that, since there are columns named `id` in both tables, you must specify the name of the table when you refer to either `id` column. The joke table's ID is referred to as `joke.id` , while the author table's ID column is `author.id` . If the table name is unspecified, MySQL won't know which `id` you're referring to, and will produce the following error:

```
Error Code: 1052. Column 'id' in field list is ambiguous
```

Now that you know how to access the data stored in your two tables efficiently, you can rewrite the code for your joke list to take advantage of joins:

```
try {
  $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
  ↪ 'mypassword');
  $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

  $sql = 'SELECT `joke`.`id`, `joketext`, `name`, `email`
    FROM `joke` INNER JOIN `author`
      ON `authorid` = `author`.`id`';

  $jokes = $pdo->query($sql);

  $title = 'Joke list';

  ob_start();

  include  __DIR__ . '/../templates/jokes.html.php';

  $output = ob_get_clean();
}
catch (PDOException $e) {
  $title = 'An error has occurred';

  $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

You can then update your `jokes.html.php` template to display the author information for each joke:

```
<?php foreach($jokes as $joke): ?>
<blockquote>
  <p>
```

```
    <?=htmlspecialchars($joke['joketext'], ENT_QUOTES, 'UTF-8')?>

    (by <a href="mailto:<?php
      echo htmlspecialchars($joke['email'], ENT_QUOTES,
        'UTF-8'); ?>"><?php
      echo htmlspecialchars($joke['name'], ENT_QUOTES,
        'UTF-8'); ?></a>)

    <form action="deletejoke.php" method="post">
      <input type="hidden" name="id" value="<?=$joke['id']?>">
      <input type="submit" value="Delete">
    </form>
    </p>
  </blockquote>
  <?php endforeach; ?>
```

If you run this script, you'll see the following result.

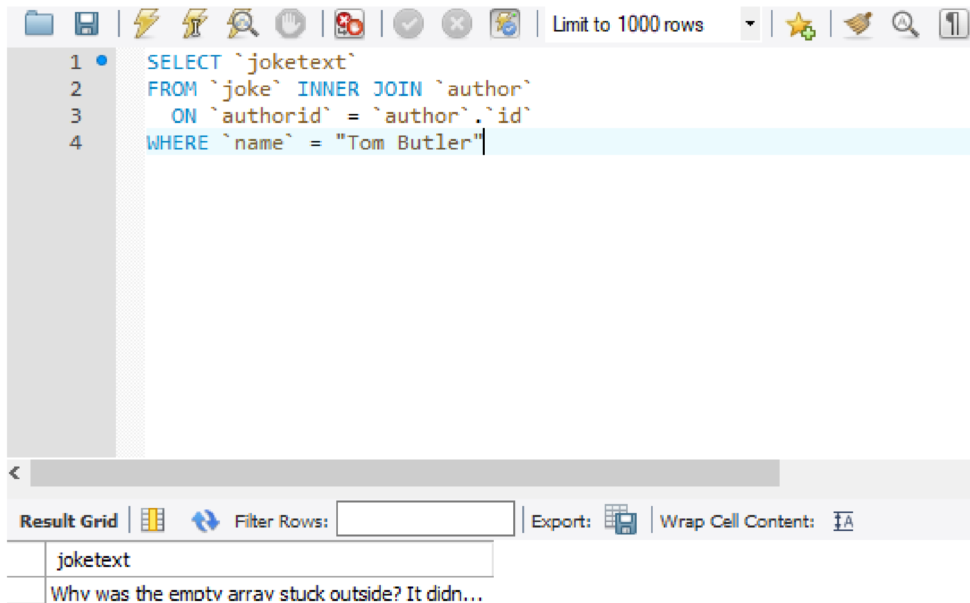
5-6. Jokes with authors

The more you work with databases, the more you'll come to realize the power of combining data from separate tables into a single table of results. Consider,

for example, the following query, which displays a list of all jokes written by Tom Butler:

```
SELECT `joketext`
FROM `joke` INNER JOIN `author`
  ON `authorid` = `author`.`id`
WHERE `name` = "Tom Butler"
```

The results that are output from this query, pictured below, come only from the `joke` table, but the query uses a join to let it search for jokes based on a value stored in the `author` table. There will be plenty more examples of clever queries like this throughout the book, but this example alone illustrates that the practical applications of joins are many and varied, and, in almost all cases, can save you a lot of work!



5-7. Tom Butler's jokes

Simple Relationships

The type of database layout for a given situation is usually dictated by the

form of relationship that exists between the data that it needs to store. In this section, I'll examine the typical relationship types, and explain how best to represent them in a relational database.

In the case of a simple **one-to-one relationship**, a single table is all you'll need. An example of a one-to-one relationship is the email address of each author in our `joke` database. Since there will be one email address for each author, and one author for each email address, there's no reason to split the addresses into a separate table. (There are exceptions to this rule. For example, if a single table grows very large with lots of columns, some of which are rarely used in `SELECT` queries, it can make sense to split those columns out into their own table. This can improve the performance of queries on the now-smaller table.)

A **many-to-one relationship** is a little more complicated, but you've already seen one of these as well. Each joke in our database is associated with just one author, but many jokes may have been written by that one author. This joke–author relationship is many-to-one. I've already covered the problems that result from storing the information associated with a joke's author in the same table as the joke itself. In brief, it can result in many copies of the same data, which are difficult to keep synchronized and waste space. If we split the data into two tables and use an ID column to link them together (making joins possible as shown before), all these problems disappear.

A **one-to-many relationship** is simply a many-to-one relationship seen from the opposite direction. As the joke–author relationship is many-to-one, the author–joke relationship is one-to-many (there's potentially one author for many jokes). This is easy to see in theory, but when you're coming at a problem from the opposite direction, it's less obvious. In the case of jokes and authors, we started with a library of jokes (the many) and then wanted to assign an author to each of them (the one). Let's now look at a hypothetical design problem where we start with the one and want to add the many.

Say we wanted to allow each of the authors in our database (the one) to have multiple email addresses (the many). When an inexperienced person in database design approaches a one-to-many relationship like this one, often the first thought is to try to store multiple values in a single database field, as

pictured below.

id	name	email
1	Kevin Yank	thatguy@kevinyank.com, kyank@example.com
2	Joan Smith	joan@example.com, jsmith@example.com

5-8. A table field overloaded with multiple values

This would work, but to retrieve a single email address from the database, we'd need to break up the string by searching for commas (or whatever special character you chose to use as a separator). It's a not-so-simple, potentially time-consuming operation. Try to imagine the PHP code necessary to remove one particular email address from a specific author! In addition, you'd need to allow for much longer values in the email column, which could result in wasted disk space, because the majority of authors would have just one email address.

Now take a step back, and realize that this one-to-many relationship is just the same as the many-to-one relationship we faced between jokes and authors. The solution, therefore, is also the same: split the new entities (in this case, email addresses) into their own table. The resulting database structure would be as shown below.

email

id	email	authorid
1	thatguy@kevinyank.com	1
2	kyank@example.com	1
3	joan@example.com	2
4	jsmith@example.com	2

author

id	name
1	Kevin Yank
2	Joan Smith

5-9. The authorid field associates each row of email with one row of author

Using a join with this structure, we can easily list the email addresses associated with a particular author:

```
SELECT `email`
FROM `author` INNER JOIN `email`
  ON `authorid` = `author`.`id`
WHERE `name` = "Kevin Yank"
```

Many-to-many Relationships

Okay, you now have a steadily growing database of jokes published on your website. It's growing so quickly, in fact, that the number of jokes has become unmanageable! Your site visitors are faced with a mammoth page that contains hundreds of jokes without any structure whatsoever. We need to make a change.

You decide to place your jokes into the following categories: knock-knock jokes, crossing-the-road jokes, lawyer jokes, light bulb jokes, and political jokes. Remembering our rule of thumb from earlier, you identify joke categories as a new entity, and create a table for them, either through MySQL

Workbench or by issuing a `CREATE TABLE` query:

```
CREATE TABLE `category` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `name` VARCHAR(255)
) DEFAULT CHARACTER SET utf8 ENGINE=InnoDB
```

This table will store a name and an ID for each category in exactly the same way the `joke` table had an `authorid` column to attribute each joke to an author. We could add a `categoryid` table to the joke table to associate each joke with a category. It would even be possible to query all the jokes form a particular category using this:

```
SELECT `joketext`, `jokedate` FROM `joke` WHERE `categoryId` = 2
```

Now you come to the daunting task of assigning categories to your jokes. It occurs to you that a political joke might also be a crossing-the-road joke, and a knock-knock joke might also be a lawyer joke. A single joke might belong to many categories, and each category will contain many jokes. This is a **many-to-many** relationship.

Once again, many inexperienced developers begin to think of ways to store several values in a single column, because the obvious solution is to add a category column to the `joke` table and use it to list the IDs of those categories to which each joke belongs. A second rule of thumb would be useful here: *if you need to store multiple values in a single field, your design is probably flawed.*

The correct way to represent a many-to-many relationship is by using a **lookup table**. This is a table that contains no actual data, but lists pairs of entries that are related. The image below shows what the database design would look like for our joke categories.

joke

id	joketext
1	Why did the chicken ...
2	Knock-knock! Who's ...
3	A man walks into a bar ...
4	How many lawyers ...

jokecategory

jokeid	categoryid
1	2
2	1
3	4
4	3
4	5

category

id	name
1	Knock-knock
2	Cross the road
3	Lawyers
4	Walk the bar
5	Light bulb

5-10. Lookup table

The `jokecategory` table associates joke IDs (`jokeid`) with category IDs (`categoryid`). In this example, we can see that the joke that starts with "How many lawyers ..." belongs to both the lawyers and light bulb categories.

A lookup table is created in much the same way as any other table. The difference lies in the choice of the primary key. Every table we've created so far has had a column named `id` that was designated to be the `PRIMARY KEY` when the table was created. Designating a column as a primary key tells MySQL to forbid two entries in that column from having the same value. It also speeds up join operations based on that column.

In the case of a lookup table, there's no single column that we want to force to have unique values. Each joke ID may appear more than once, as a joke may belong to more than one category, and each category ID may appear more than once, as a category may contain many jokes. What we want to prevent is the same *pair* of values appearing in the table twice. And, since the sole purpose of this table is to facilitate joins, the speed benefits offered by a primary key would come in very handy. For this reason, we usually create lookup tables with a multicolumn primary key as follows:

```
CREATE TABLE `jokecategory` (
```

```
  `jokeid` INT NOT NULL,
  `categoryid` INT NOT NULL,
  PRIMARY KEY (`jokeid`, `categoryid`)
) DEFAULT CHARACTER SET utf8 ENGINE=InnoDB
```

The same can be done in MySQL Workbench by checking the `PK` checkbox for both columns. This creates a table in which the `jokeid` and `categoryid` columns together form the primary key. This enforces the uniqueness that's appropriate to a lookup table, preventing a particular joke from being assigned to a specific category more than once, and speeds up joins that make use of this table. (If you like, you can use the `CREATE TABLE` and `INSERT` commands in Chapter 3 to create the `jokecategory` table from scratch—and others, including the jokes within the tables—to follow along.)

Now that your lookup table is in place and contains category assignments, you can use joins to create several interesting and practical queries. This query lists all jokes in the knock-knock category:

```
SELECT `joketext`
FROM `joke`
INNER JOIN `jokecategory`
  ON `joke`.`id` = `jokeid`
INNER JOIN `category`
  ON `categoryid` = `category`.`id`
WHERE name = "knock-knock"
```

As you can see, this query uses *two* joins. First, it takes the `joke` table and joins it to the `jokecategory` table. Then it takes that joined data and joins it to the category table. As your database structure becomes more complex, multijoin queries like this one become common.

The following query lists the categories that contain jokes beginning with "How many lawyers …":

```
SELECT `name`
FROM `joke`
INNER JOIN `jokecategory`
```

```
  ON `joke`.`id` = `jokeid`
INNER JOIN `category`
  ON `categoryid` = `category`.`id`
WHERE `joketext` LIKE "How many lawyers%"
```

And this query—which also makes use of our `author` table to join together the contents of *four tables*—lists the names of all authors who have written knock-knock jokes:

```
SELECT `author`.`name`
FROM `joke`
INNER JOIN `author`
  ON `authorid` = `author`.`id`
INNER JOIN `jokecategory`
  ON `joke`.`id` = `jokeid`
INNER JOIN `category`
  ON `categoryid` = `category`.`id`
WHERE `category`.`name` = "knock-knock"
```

This is starting to get complicated! Although `JOIN` s can be used, later on in the book we'll see some different approaches that can help reduce this complexity—though sometimes at the expense of efficiency.

We won't add categories to the website just yet, but you now at least have a basic understanding of how the database could be structured to do so.

`INNER` and `OUTER` Joins

There are `INNER` and `OUTER` joins in SQL. They both do the same job, retrieving data from multiple tables in a single query, but in slightly different ways. An `INNER JOIN` will only find records where there are matching records in both tables, while an `OUTER JOIN` will find all the records from one table along with the matching records from another.

We won't be making use of `JOIN`s in this book, but if you want to learn more about relational databases, I recommend *Jump Start MySQL*[2], by Timothy Boronczyk.

One for Many, and Many for One

In this chapter, I've explained the fundamentals of good database design, and covered how MySQL and, for that matter, all relational database management systems provide support for the representation of different types of relationships between entities. From your initial understanding of one-to-one relationships, you should now have expanded your knowledge to include many-to-one, one-to-many, and many-to-many relationships.

In the process, you've learned about some common SQL commands—in particular, how to use a `SELECT` query to join data spread across multiple tables into a single set of results.

With the increased expressiveness that multiple database tables bring, you're now equipped to extend the simple "joke list" site you assembled in Chapter 4 to include authors and categories, and that's exactly what Chapter 9 will be all about. Before you tackle this project, however, you should take some time to add to your PHP skills. Just as you spent this chapter learning some of the finer points of MySQL database design, Chapter 6 will teach you some of the subtleties of PHP programming—which will make the job of building a more complete joke database site much more fun.

---

2. https://www.sitepoint.com/premium/books/jump-start-mysql/

Chapter 6

Structured PHP Programming

Before we plow headlong into the next enhancements of our `joke` database, let's spend a little time honing your "PHP-fu". Specifically, I want to show you a few techniques to better structure your code.

Structured coding techniques are useful in all but the simplest of PHP projects. In Chapter 2, we looked at how to split our PHP code into multiple files, using a controller and a set of associated templates. This lets us keep the server-side logic of our site separate from the HTML code used to display the dynamic content generated by that logic. In order to do this, we looked at how to use the PHP `include` command.

The PHP language offers many such facilities to help you add structure to your code. The most powerful of these is undoubtedly its support for object-oriented programming (OOP), which we touched on briefly in Chapter 4. But there's no need to learn all the complexities of OOP to build complex (and well-structured) applications with PHP. Thankfully, there are also opportunities for structuring your code through the more basic features of PHP.

In this chapter, I'll explore some methods of keeping your PHP code manageable and maintainable. As projects grow in size, so does the code. When you want to make a change to something, you'll need to find the point in the code you want to change. This can be tricky, and sometimes requires editing code in more than one place.

Programmers are lazy, and we don't want to have to make the same change in multiple locations. By placing code in one place, and using it with the `include` statement, it allows us to avoid repetition. If you ever find yourself copying and pasting code, you're almost certainly better off moving that repeated code into its own file and using it in both locations with an `include` statement. This is commonly referred to as the *don't repeat yourself* (DRY) principle. Any time you find yourself reaching for `Ctrl` + `C` for a block of code, there's almost certainly a better way to solve the problem than repeating the same code twice or more.

On the other hand, the computer doesn't care how you structure your code

and will blindly follow any instructions you give it. Programmers structure the code, reduce repetition and break code into small chunks purely to make our own job easier. Code is a lot simpler to manage when broken up into small tasks. Trying to find an error in a 1000-line PHP script that does a dozen different things is a lot more difficult than finding the same error in a 30-line file that only performs a single task.

Include Files

Even very simple PHP-based websites often need the same piece of code in several places. You've already learned to use the PHP `include` command to load PHP templates from inside your controllers. It turns out you can use the same feature to save yourself from having to write the same code again and again, just like you did with `layout.html.php`: you wrote some HTML code and reused it for every page.

Include files (also known just as **includes**) also contain snippets of PHP code that you can load into your other PHP scripts instead of having to retype them.

Including HTML Content

The concept of include files came long before PHP. If you're an old codger like me (which, in the web world, means you're over 30), you may have experimented with server-side includes (SSIs)—a long-forgotten feature that existed in early web servers. SSIs let you put commonly used snippets of HTML (and JavaScript, and CSS) into include files that you can then use in multiple pages—for example, to put the code for the navigation bar in one file and reference that file on every page. The benefit of this is that, to add a link to the navigation bar on every page, you'd only have to edit one file.

PHP supports include files like this. And for someone who's mostly used to working with pure HTML files, these includes are one of the most immediately beneficial tools that PHP has to offer.

In PHP, include files most commonly contain either pure PHP code or, in the case of PHP templates, a mixture of HTML and PHP code. But you don't *have*

to put PHP code in your include files. If you like, an include file can contain strictly static HTML. This is most useful for sharing common design elements across your site, such as a copyright notice at the bottom of every page:

```
<footer>
  The contents of this web page are copyright &copy; 1998&ndash;2021
  Example LLC. All Rights Reserved.
</footer>
```

This file is a **template partial**—an include file to be used by PHP templates. To distinguish this type of file from others in your project, I recommend giving it a name ending with `.html.php` to differentiate from non-template pages. This naming convention is common in projects that use templates in this way.

The `.html.php` Extension

The extension `.html.php` is optional, but it makes managing your files easier. You might, for example, have a template called `jokes.html.php` and a file called `jokes.php` open in different tabs in your editor. By giving the files a different extension, it's easy to quickly identify which is which.

You can then use this partial in any of your PHP templates:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A Sample Page</title>
  </head>
  <body>
  <main>
    This page uses a static include to display a standard
    copyright notice below.
  </main>
  <?php include 'footer.html.php'; ?>
  </body>
```

```
</html>
```

Finally, here's the controller that loads this template:

```
<?php
include 'samplepage.html.php';
?>
```

6-1. The completed page

The screenshot above shows what the page looks like in the browser.

Now all you need to do to update your copyright notice is to edit *footer.html.php*. No more time-consuming, error-prone find-and-replace operations!

Of course, if you *really* want to make your life easier, you can just let PHP do the work for you:

```
<p id="footer">
  The contents of this web page are copyright &copy;
  1998&ndash;<?php echo date('Y'); ?> Example LLC.
  All Rights Reserved.
</p>
```

Including PHP Code

On most websites with a database, almost every controller script must establish a database connection as its first order of business, and finally include the `layout.html.php` file. Our controllers all follow this pattern:

```
<?php
try {
    $pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
    ➥ 'mypassword');

    // do something unique for this page
  // setting the $title and $output variables
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Unable to connect to the database server: ' . $e->getMessage() .
    ➥' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include __DIR__ . '/../templates/layout.html.php';
```

At some 12 lines long, it's only a slightly cumbersome chunk of code, but having to repeat it in every controller script can quickly become annoying. Many new PHP developers will often omit essential error checking to save typing (for example, by leaving out the `try … catch` statement in this code), which can result in a lot of lost time looking for the cause when an error *does* occur. Others will make heavy use of the clipboard to copy pieces of code like this from existing scripts for use in new ones. Some even use features of their text editor software to store useful pieces of code as snippets for frequent use.

Instead of repeating this code at the top of every file, we can move it to an *include file*. Include files are just like normal PHP files, but typically they contain snippets of code that are only useful within the context of a larger script. As such, as with templates, we don't want people to be able to navigate directly to these files by typing the filename into their browser, as they only

contain small snippets that won't produce any meaningful output on their own.

We'll solve this problem the same way we did with templates: by creating a directory outside the `public` directory, so that any files placed in this new directory can only be accessed by other PHP scripts. We'll call this directory `includes`, and use it to store our code snippets.

Inside the new `includes` directory, create a file called `DatabaseConnection.php` and place the database connection code inside it:

```
<?php
$pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
↪ 'mypassword');
```

Naming Files

In previous versions of this book, the database connection was placed in a file named `db.inc.php`, which at the time was the convention: all include files were named with a `.inc.php` extension. This convention has lost popularity, and include files are generally created using camelCase. This is in part due to both the PSR-0 standard[1] and heavy use of object-oriented programming in modern websites.

Now you can put this `DatabaseConnection.php` file to use in your controllers. Amend each of your controllers—`addjoke.php`, `deletejoke.php` and `jokes.php`—to include the new file.

The updated `jokes.php` looks like this.

Example: Structure-Include[2]

1. http://www.php-fig.org/psr/psr-0/
2. https://github.com/spbooks/phpmysql7/tree/Structure-Include

```
<?php

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';

    $sql = 'SELECT `joke`.`id`, `joketext`, `name`, `email`
    FROM `joke` INNER JOIN `author`
      ON `authorid` = `author`.`id`';

    $jokes = $pdo->query($sql);

    $title = 'Joke list';

    ob_start();

    include  __DIR__ . '/../templates/jokes.html.php';

    $output = ob_get_clean();
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

Make the same change to `addjoke.php` and `delete.php` so they use an `include` statement rather than repeating the database connection code.

As you can see, wherever our controller needs a database connection, we can obtain it simply by including the `DatabaseConnection.php` file with an `include` statement. And because the code to do this is a simple one-liner, we can make our code more readable by using a separate `include` statement just before each SQL query in our controller.

When PHP encounters an `include` statement, it puts the current script on hold and runs the specified PHP script. When it's finished, it returns to the original script and picks up where it left off.

Include files are the simplest way to structure PHP code. Because of their simplicity, they're also the most widely used method. Even very simple web applications can benefit greatly from using include files.

You've already seen how you can create a variable such as `$title` in the controller and that it's available in included files such as `layout.html.php`. Here, you can see the inverse is also true. The `$pdo` variable is created in `DatabaseConnection.php`, but can be used in the controller.

An `include` statement can be thought of as an automated copy-and-paste. When PHP encounters the line `include __DIR__ . '/../includes/DatabaseConnection.php';` it effectively reads the code from the file and copies/pastes it into the current code at the position of the `include` statement.

After you've amended all your controllers to have an `include` statement, if the database password gets updated, you only need to edit `DatabaseConnection.php`, rather than having to update the password in each of your controllers. Aren't you glad we made this change now instead of later when we've got more controllers?

Types of Includes

The `include` statement we've used so far is actually only one of four statements that can be used to include another PHP file in a currently running script:

- `include`
- `require`
- `include_once`
- `require_once`

The only difference between them is what happens when the specified file is unable to be included (that is, if it doesn't exist, or if the web server doesn't have permission to read it). With `include`, a warning is displayed and the

script continues to run. With `require`, an error is displayed and the script stops.

`include` **vs** `require`

In production environments, warnings and errors are usually disabled in `php.ini`. In such environments, a failed `include` has no visible effect (aside from the lack of content that would normally have been generated by the include file), while a failed `require` causes the page to stop at the point of failure. When a failed `require` occurs before any content is sent to the browser, the unlucky user will see nothing but a blank page!

In general, you should use `require` whenever your application simply wouldn't work without the required code being successfully loaded. I do recommend using `include` whenever possible, however. Even if the `DatabaseConnection.php` file for your site is unable to load, for example, you might still want to let the script for your front page continue to load. None of the content from the database will display, but the user might be able to use the **Contact Us** link at the bottom of the page to let you know about the problem!

`include_once` and `require_once` work just like `include` and `require`, respectively—but if the specified file has already been included at least once for the current page request (using *any* of the four statements described here), the statement will be ignored. This is handy for include files performing a task that only needs to be done once, like connecting to the database.

`include_once` and `require_once` are also useful for loading function libraries, as we'll see in the following section.

As you'll see later on in the book, as your code becomes more advanced, you'll start making use of *autoloaders*, and you'll rarely need to write `include` statements directly in the code, but I'm getting ahead of myself.

Custom Functions and Function Libraries

By this point, you're probably quite comfortable with the idea of functions. A function is PHP code that you can invoke at will, where you'd usually provide one or more arguments for it to use, and often receive a return value as a result. You can use PHP's vast library of functions to do just about anything a PHP script could ever be asked to do, from retrieving the current date (`date`) to generating graphics on the fly (using `imagecreatetruecolor`[3]).

But what you may be unaware of is that you can create functions of your own! **Custom functions**, once defined, work just like PHP's built-in functions, and they can do anything a normal PHP script can do.

Let's start with a really simple example. Say you had a PHP script that needed to calculate the area of a rectangle given its width (`3`) and height (`5`). Thinking back to your basic geometry classes in school, you should recall that the area of a rectangle is its width multiplied by its height:

```
$areaOfRectangle = 3 * 5;
```

But it would be nicer to have a function called `areaOfRectangle` that simply calculated the area of a rectangle given its dimensions:

```
$areaOfRectangle = areaOfRectangle(3, 5);
```

As it happens, PHP has no built-in `areaOfRectangle` function, but clever PHP programmers like you and me can just roll up our sleeves and write the function ourselves:

```
<?php
function areaOfRectangle($width, $height)
{
    return $width * $height;
}
```

---

3. http://www.php.net/imagecreatetruecolor

This include file defines a single custom function— `areaOfRectangle`. The `<?php` marker is probably the only line that looks familiar to you in this code. What we have here is a **function declaration**. Let me break it down for you one line at a time:

```
function areaOfRectangle($width, $height)
```

The keyword `function` tells PHP that we wish to declare a new function for use in the current script. Then, we supply the function with a name (in this case, `areaOfRectangle`). Function names operate under the same rules as variable names: they must start with a letter or an underscore (`_`), and may contain letters, numbers, and underscores—except, of course, that there's no dollar sign prefix. Instead, function names are always followed by a set of parentheses (`()`), which may or may not be empty.

The parentheses that follow a function name enclose the list of arguments that the function will accept. You should already be familiar with this from your experience with PHP's built-in functions. For example, when you use `rand` to generate a random number, you can provide it with a minimum and maximum number within the parentheses.

When declaring a custom function, instead of giving a list of values for the arguments, you give a list of variable names. In this example, we list two variables: `$width` and `$height`. When the function is called, it will therefore expect to be given two arguments. The value of the first argument will be assigned to `$width`, while the value of the second will be assigned to `$height`. Those variables can then be used to perform the calculation within the function.

Speaking of calculations, the rest of the function declaration is the code that performs the calculation, or does whatever else the function is supposed to do. That code must be enclosed in a set of braces (`{ … }`). You can write nearly any code you like between the opening and closing braces, but the code for the area function is as follows:

```
return $width * $height;
```

You can think of the code within those braces as a miniature PHP script. This function is a simple one, because it contains just a single statement—a `return` statement.

A `return` statement can be used in the code of a function to send data out of the function back into the place it was called. In this case, we're sending the result of the `$width * $height` calculation out of the function in the same way the `rand` function sends a random number from the function to the variable we're storing the number in.

In addition, the `return` statement jumps back into the main script immediately. When the PHP interpreter hits a `return` statement, it immediately stops running the code of this function and goes back to where the function was called. It's sort of an ejector seat for functions!

Writing a function on its own does nothing. No code inside the function is run until the function is called. Like the `rand` function we used at the start of this book, it just sits there waiting to be called.

You can think of writing a function like installing an app on your computer or phone. You need it there to use it, but once installed, it's dormant and available for use, but won't actually do anything until you run it.

Although it's possible to write the function in the same file that we want to run it in, that's a bad idea. We might want to run the function from different files, and like the database connection above, repeating code isn't good. Instead, we can put it in its own file (such as `areaOfRectangle.php`) in the `includes` directory, and then include it in the script we wish to use it in:

```
include_once  __DIR__ . '/../includes/areaOfRectangle.php';

$areaOfRectangle = areaOfRectangle(3, 5);
```

```
include 'output.html.php';
```

Avoid using *`include`* or *`require`* to load include files that contain functions. As explained in the "Types of Includes" section earlier in this chapter, that would risk defining the functions in the library more than once and covering the user's screen with PHP warnings.

It's standard practice (but not required) to include your function libraries at the top of the script, so that you can quickly see which include files containing functions are used by any particular script.

What we have here are the beginnings of a **function library**—an include file that contains declarations for a group of related functions. If you wanted to, you could rename the include file to *`geometry.php`*, add a whole bunch of functions to it, and perform various geometrical calculations.

Variable Scope

One big difference between custom functions and include files is the concept of **variable scope**. Any variable that exists in the main script will also be available and can be changed in the include file. While this is useful sometimes, more often it's a pain in the neck. Unintentionally overwriting one of the main script's variables in an include file is a common cause of errors—and ones that can take a long time to track down and fix! To avoid such problems, you need to remember the variable names in the script that you're working on, as well as any that exist in the include files your script uses.

Functions protect you from such problems. A function is like its own independent mini program with inputs (arguments) and an output (a return value). Anything the function does is effectively hidden from the script that calls it. The script calling the function sends it some values and then receives the return values. It doesn't matter what the function does in between receiving the arguments and returning the value.

Variables created inside a function (including any argument variables) exist

only within that function, and disappear when the function has run its course. In addition, variables created outside the function are completely inaccessible inside it. The only variables a function has access to are the ones provided to it as arguments.

In programmer-speak, the **scope** of these variables is the function; they're said to have **function scope**. In contrast, variables created in the main script, outside of any function, are unavailable inside functions. The scope of these variables is the main script, and they're said to have **global scope**.

Okay, but beyond the fancy names, what does this really *mean* for us? It means that you could have a variable called `$width` in your main script, and another variable called `$width` in your function, and PHP would treat these as two entirely separate variables! Perhaps more usefully, you can have two different functions each using the same variable names, and they'll have no effect on each other, because their variables are kept separate by their scope.

On some occasions, you may actually *want* to use a global-scope variable (**global variable** for short) inside one of your functions. For example, the `DatabaseConnection.php` file creates a database connection for use by your script and stores it in the global variable `$pdo`. You might then want to use this variable in a function that needed to access the database.

Let's create a function that queries the database and returns to us the number of jokes that are currently held in the `joke` table. Disregarding variable scope, here's how you might expect such a function to work:

```
include_once __DIR__ . '/../includes/DatabaseConnection.php';

function totalJokes() {
  $stmt = $pdo->prepare('SELECT COUNT(*) FROM `joke`');
  $stmt->execute();

  $row = $stmt->fetch();

  return $row[0];
```

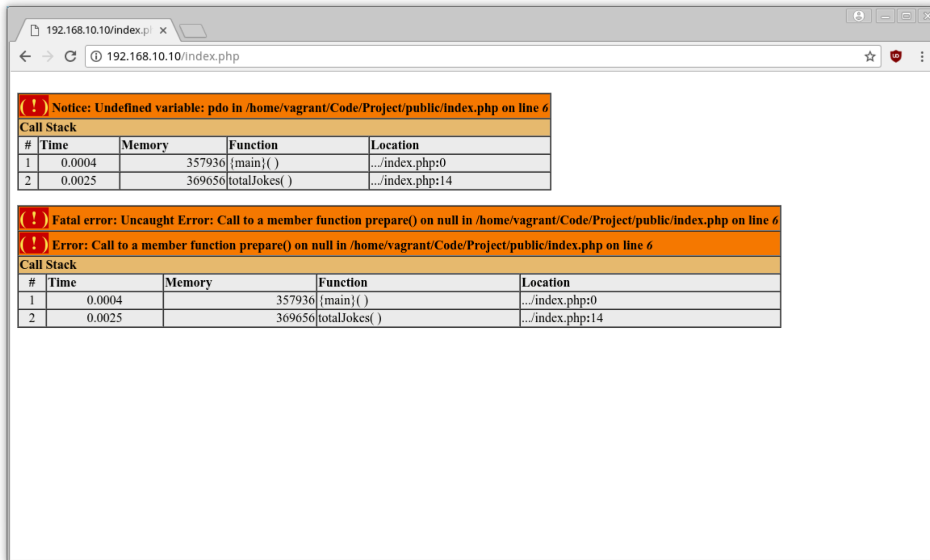
```
}

echo totalJokes();
```

Be Careful Where You Place Your Files

Note that the first line of this controller script uses a shared copy of the `DatabaseConnection.php` file in the `includes` directory. Make sure you've placed a copy of this file in the `includes` directory. Otherwise, PHP will complain that it's unable to find the `DatabaseConnection.php` file.

The file `DatabaseConnection.php` creates the variable `$pdo`, which we're referencing inside the function.

The `DatabaseConnection.php` include file creates the variable `$pdo` in the script. The problem here is that the `$pdo` variable is unavailable within the scope of the function. If you attempt to call this function as it is, you'll receive the errors shown in the following image.



6-2. The totaljokes function cannot access $pdo

The reason for this error is **scope**. The `$pdo` variable was created outside the function, so it's not available for use inside it.

Although there *is* a way[4] to make the same `$pdo` variable available in the function, it's a very bad idea. (Seriously, don't do this!) If, for example, the function changed the `$pdo` by accident to a string— `$pdo = 'select * from joke';` —the `$pdo` variable would now be a string everywhere else in the PHP script. Global variables are a *very* bad idea and lead to problems that are very difficult to track down and fix. You should avoid global variables at any cost.

4. http://php.net/manual/en/reserved.variables.globals.php

Why Global Variables Are Bad

For more information on why global variables are bad, see one of the following links:

- "Global Variables are Evil"[5]
- "Wow, They Were Right, Global Variables Are Bad"[6]
- "Global Variables Are Bad"[7]

To avoid this, you can use `$pdo` as an argument and pass in the required variables to your function:

```
function totalJokes($pdo) {
  $stmt = $pdo->prepare('SELECT COUNT(*) FROM `joke`');
  $stmt->execute();

  $row = $stmt->fetch();

  return $row[0];
}
```

Then, when the function is called, you pass in the `$pdo` object created in `DatabaseConnection.php`:

```
include_once __DIR__ . '/../includes/DatabaseConnection.php';
echo totalJokes($pdo);
```

This makes the `$pdo` variable available in the `totalJokes` function by passing it in as an argument. It's worth taking a moment to understand exactly what's happening here. The `$pdo` variable is created in `DatabaseConnection.php`, which makes it available in the block of code above. It's then passed in to the function `totalJokes`. The object stored in the global `$pdo` variable is then copied to the local variable called `$pdo` inside the function. The database

---

5. https://www.cs.usfca.edu/%7Ewolber/courses/110/lectures/globals.htm
6. https://blog.cranksoftware.com/wow-they-were-right-global-variables-are-bad/
7. http://wiki.c2.com/?GlobalVariablesAreBad

connection needs to be passed in as an argument, because functions only have access to data they're given; they can't access variables other than the ones they're given (or, as noted earlier, there are ways of doing this but they're almost always a bad idea).

By passing `$pdo` as an argument, if there's a mistake in the function and the `$pdo` variable is overwritten with a string inside the function, it will only be a string inside the function, not throughout the rest of the script.

What this means is that there are actually two different variables called `$pdo`. We could rename the `$pdo` variable inside the function to `$database` and the script would still work:

```
function totalJokes($database) {
  $stmt = $database->prepare('SELECT COUNT(*) FROM `joke`');
  $stmt->execute();

  $row = $stmt->fetch();

  return $row[0];
}
```

Here, the `$database` variable stores the same `PDO` connection that's stored in the `$pdo` variable in global scope. Even though the function would still be called using `totalJokes($pdo)`, the content of the `$pdo` variable outside the function contains the same database connection as the `$database` variable inside the function. Despite having different names, both are referencing the same connection.

If you want to get technical, this process is called **dependency injection**, but all you need to know is that, in practical terms, it's a method for making a single variable available in multiple locations.

It's a good idea to move functions into their own file. Place the `totalJokes` function inside a file called `DatabaseFunctions.php` in the `includes` directory, and you can then use the function as follows.

Example: Structure-TotalJokes[8]

```
// Include the file that creates the $pdo variable and connects to the database
include_once __DIR__ . '/../includes/DatabaseConnection.php';

// Include the file that provides the `totalJokes` function
include_once __DIR__ . '/../includes/DatabaseFunctions.php';

// Call the function
echo totalJokes($pdo);
```

Save this as `showtotaljokes.php` in the `public` directory and navigate to it in your browser. You should see a page that's mostly empty but displays the total number of jokes in your database.

You may have realized that, if the program is broken up into functions like this, you'd need to pass the `$pdo` variable into each function you wanted to use. As you've probably suspected, this isn't the most efficient way of achieving this goal, as we'll see later on when we get to objects and classes.

Let's use our new function inside the website. At the top of the list of jokes we can print "[number] jokes have been submitted to the Internet Joke Database".

Open up `jokes.php` and change it to the following:

```
<?php

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../includes/DatabaseFunctions.php';

    $sql = 'SELECT `joke`.`id`, `joketext`, `name`, `email`
    FROM `joke` INNER JOIN `author`
      ON `authorid` = `author`.`id`';

    $jokes = $pdo->query($sql);
```

8. https://github.com/spbooks/phpmysql7/tree/Structure-TotalJokes

```
    $title = 'Joke list';

    $totalJokes = totalJokes($pdo);

    ob_start();

    include  __DIR__ . '/../templates/jokes.html.php';

    $output = ob_get_clean();
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

This creates a `$totalJokes` variable that can be used in `jokes.html.php`.

Example: Structure-TotalJokesList[9]

```
<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>

<?php foreach ($jokes as $joke): ?>
<blockquote>
  <p>
  <?=htmlspecialchars($joke['joketext'], ENT_QUOTES, 'UTF-8')?>

  (by <a href="mailto:<?php
    echo htmlspecialchars(
    $joke['email'],
    ENT_QUOTES,
      'UTF-8'
); ?>"><?php
    echo htmlspecialchars(
          $joke['name'],
          ENT_QUOTES,
      'UTF-8'
      ); ?></a>)
```

9. https://github.com/spbooks/phpmysql7/tree/Structure-TotalJokesList

```
  <form action="deletejoke.php" method="post">
    <input type="hidden" name="id" value="<?=$joke['id']?>">
    <input type="submit" value="Delete">
  </form>
  </p>
</blockquote>
<?php endforeach; ?>
```

You may be wondering what happens if the query inside the `totalJokes` function contains an error. We could put a `try … catch` statement inside the function, but because the function has been called inside the existing `try … catch` statement, we don't need to.

Exceptions *bubble up*. In practical terms, this means that, even if an exception is thrown inside a function, it will still be caught by a surrounding `try … catch` statement.

It's better to avoid handling errors inside functions. Imagine we have a `try … catch` statement inside the function like so:

```
function totalJokes($database) {
  try {
    $stmt = $database->prepare('SELECT COUNT(*) FROM `joke`');
    $stmt->execute();

    $row = $stmt->fetch();

    return $row[0];
  }
  catch {
    $title = 'An error has occurred';

      $output = 'Database error: ' . $e->getMessage() . ' in ' .
      $e->getFile() . ':' . $e->getLine();

    include  __DIR__ . '/../templates/layout.html.php';
    die();
  }
```

```
}
```

Not only do we have repeated code, but now, regardless of where the `totalJokes` function is called from, the error will be handled in the same way. By keeping it out of the function, different error handling can be used in different places that the `totalJokes` function is used. For example, it's possible that we would want to use the website to generate a list of jokes as a PDF or an Excel file rather than as an HTML document. If this were the case, we certainly wouldn't want an error message displayed as HTML.

By just letting the error happen, we can handle the error in different ways under different circumstances.

Breaking Up Your Code into Reusable Functions

Now that you're familiar with declaring your own functions, you can start writing functions to perform each task. For example, instead of writing out a `SELECT` query each time you want to read a specific joke from the database, you could write a reusable function to do that for you:

```
function getJoke($pdo, $id) {
  $stmt = $pdo->prepare('SELECT * FROM `joke` WHERE `id` = :id');

  $values = [
      'id' => $id
  ];
  $stmt->execute($values);

  return $stmt->fetch();
}
```

This works in the same way as the `totalJokes` function from earlier. The only difference is that there's a second argument, `$id`, which is going to store the ID of the joke to be looked up.

This allows us to look up a joke very quickly by its ID:

```
include __DIR__ . '/../includes/DatabaseConnection.php';

//Equivalent of SELECT * FROM `joke` WHERE id = 1
$joke1 = getJoke($pdo, 1);

echo $joke1['joketext'];

//Equivalent of SELECT * FROM `joke` WHERE id = 2
$joke2 = getJoke($pdo, 2);

echo $joke2['joketext'];
```

You now have functions `getJoke` and `totalJokes`, which can be used anywhere you want to find a specific joke or count the number of jokes in the database. Rather than writing out a query, preparing it, creating a values array and executing the query, you can perform that entire process with a single short line of code, such as `getJoke($pdo, 1)`.

Using Functions to Replace Queries

The `totalJokes` function allows for easily querying the database for the number of jokes in the table, and the `getJoke` function lets us quickly fetch a joke from the database. Let's take the same approach and apply it to adding jokes to the database with an `INSERT` query:

```
function insertJoke($pdo, $joketext, $authorId) {
  $stmt = $pdo->prepare('INSERT INTO `joke` (`joketext`, `jokedate`, `authorId`)
      VALUES (:joketext, :jokedate, :authorId)');

  $values = [
    ':joketext' => $joketext,
    ':authorId' => $authorId,
    ':jokedate' => date('Y-m-d')
  ];

  $stmt->execute($values);
}
```

The `insertJoke` function lets us very quickly insert a record into the `joke`

table with a single line of code by providing it the database connection (`$pdo`), the text of the joke (`$joketext`), and the ID of the author (`$authorId`):

```
insertJoke($pdo, 'Why did the programmer quit his job? He didn\'t get arrays'
↪, 1);
```

Each of the columns in the database is an argument for the function, which can now be called repeatedly to quickly issue the relevant `INSERT` query with far less code than we'd have needed previously:

```
insertJoke($pdo, 'Why was the empty array stuck outside? It didn\'t have any
↪ keys', 1);

insertJoke($pdo, 'An SQL query goes into a bar, walks up to two tables and asks
↪ "Can I join you?"', 2);
```

This approach is considerably quicker and easier than having to write all the code for running the query each time you insert a joke: preparing the query, binding the parameters, and then finally executing the query.

Now, finally, you *could* use the following controller code for adding a joke:

```
if (isset($_POST['joketext'])) {
  try {
    include __DIR__ . '/../includes/DatabaseConnection.php';

    $sql = 'INSERT INTO `joke` (`joketext`, `jokedate`, `authorId`)
    VALUES (:joketext, :jokedate, :authorId)';

    $stmt = $pdo->prepare($sql);

    $values = [
      ':joketext' => $_POST['joketext'],
      ':authorId' => 1,
      ':jokedate' => date('Y-m-d')
    ];

    $stmt->execute($values);
```

```
    header('location: jokes.php');
  }
  catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
  }
}
```

But instead of this, you can put the `insertJoke` and `query` functions in the `DatabaseFunctions.php` file and then write the following much simpler version.

Example: Structure-AddJoke[10]

```
if (isset($_POST['joketext'])) {
  try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../includes/DatabaseFunctions.php';

    insertJoke($pdo, $_POST['joketext'], 1);

    header('location: jokes.php');
  }
  catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
  }
}
```

You can see that I've put the `getJoke`, `totalJokes` and `insertJoke` functions into a file called `DatabaseFunctions.php` and included it as required.

For now, all jokes will have `1` set as the `authorId`. Later on, I'll show you how to handle logins and associate the joke with the currently logged-in user.

---

10. https://github.com/spbooks/phpmysql7/tree/Structure-AddJoke

Updating Jokes

Along with inserting a joke, it's useful to be able to update a record. Perhaps there was a spelling error in one of the jokes and it needs to be changed.

An update function needs more information than the insert function. It needs to know the ID of the record being updated, along with the new values for each column:

```
function updateJoke($pdo, $jokeId, $joketext, $authorId) {
  $stmt = $pdo->prepare('UPDATE `joke` SET
                          `authorId` = :authorId,
                          `joketext` = :joketext
                        WHERE `id` = :id');

  $values = [
    ':joketext' => $joketext,
    ':authorId' => $authorId,
    ':id' => $jokeId
  ];

  $stmt->execute($values);

}
```

The `updateJoke` function can be called anywhere a joke needs to be updated:

```
updateJoke($pdo, 1, '!false - It\'s funny because it\'s true', 1);
```

The line above will update the joke whose ID is `1` with the supplied `authorId` and `jokeText` —the equivalent of running all the following code:

```
$stmt = $pdo->prepare('UPDATE `joke` SET
                        `authorId` = :authorId,
                        `joketext` = :joketext
                      WHERE `id` = :id');

$values = [
```

```
  ':joketext' => '!false - It\'s funny because it\'s true',
  ':authorId' => 1,
  ':id' => 1
];

$stmt->execute($values);
```

This is a significant advantage. Anywhere in the code that a joke needs to be updated, the single function call can do it, saving a lot of typing and duplication of effort.

Editing Jokes on the Website

Let's make use of our new functions `getJoke` and `updateJoke` by adding a page that allows for editing the existing jokes. In essence, it's the same as `addjoke.php`. It will display a form, and when the form is submitted it will send the data to the database.

However, there are two main differences:

- When the edit page loads, it needs to retrieve the current joke text from the database in order to pre-fill the `<textarea>` with the current joke. After all, if it's a simple typo, you don't want to make the user re-type the whole joke!

- When the form is submitted, it needs to run an `UPDATE` query rather than an `INSERT` query.

In the `public` directory, create the `editjoke.php`, which looks like this:

```
<?php
try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../includes/DatabaseFunctions.php';

    if (isset($_POST['joketext'])) {
        updateJoke($pdo, $_POST['jokeid'], $_POST['joketext'], 1);
```

```
        header('location: jokes.php');
    } else {
        $joke = getJoke($pdo, $_GET['id']);

        $title = 'Edit joke';

        ob_start();

        include  __DIR__ . '/../templates/editjoke.html.php';

        $output = ob_get_clean();
    }
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

There are a few main differences between this new `editjoke.php` page and the `addjoke.php` page.

Firstly, you may have noticed that the two include files have been included at the very top of the page. This is so the database functions are available whether or not the form has been submitted.

If the form hasn't been submitted, we need to query the database for the current joke text. If it has been submitted, we'll need to run the relevant update query.

Secondly, the `try` statement surrounds the `if` rather than having the `if` surround the `try`. The reason for this is simple: there may be an error in either the `if` or `else` block.

Speaking of the `else` block, that's the next change. Instead of just loading the form, the `else` block has this line of code:

```
$joke = getJoke($pdo, $_GET['id']);
```

This part of the code selects a joke from the database by its ID, using the `getJoke` function from earlier. The ID of the joke being edited must be supplied using a `GET` variable—so that visiting `editjoke.php?id=4`, for example, will execute the query `` SELECT * FROM `joke` WHERE `id` = 4 `` and store the resulting joke in the `$joke` array.

The `$joke` variable can now be used in the corresponding template file `editjoke.html.php`:

```
<form action="" method="post">
  <input type="hidden" name="jokeid" value="<?=$joke['id'];?>">
  <label for="joketext">Type your joke here:</label>
  <textarea id="joketext" name="joketext" rows="3" cols="40"><?=htmlspecialchars
  ➥($joke['joketext'], ENT_QUOTES, 'UTF-8')?></textarea>
  <input type="submit" value="Save">
</form>
```

This template is slightly different from the template for adding jokes. The most obvious change is that the current joke text, from the `$joke` variable, is loaded into the `<textarea>` when the page loads. However, there's another change: there's also a hidden input that sends the ID of the joke being edited back to the page when the submit button is pressed.

In `editjoke.php`, there are now two `$_POST` variables available when the form has been submitted—`$_POST['jokeid']`, representing the ID of the joke being edited, and `$_POST['joketext']`, containing the new text for the joke.

These are then passed into our `updateJoke` function:

```
updateJoke($pdo, $_POST['jokeid'], $_POST['joketext'], 1);
```

Before you can test this example for yourself, you'll need to add a link next to each joke in the joke list that gives us a link to the `editjoke.php` controller with the ID of the joke being edited. If you go directly to `editjoke.php`, you'll

see an error, as `editjoke.php` needs the ID of the joke being edited in order for the current value in the `joketext` column to be displayed in the `<textarea>`.

Open up `jokes.html.php` from the `templates` directory and add a link to `editjoke.php` for each joke:

```
<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>

<?php foreach ($jokes as $joke): ?>
<blockquote>
  <p>
  <?=htmlspecialchars($joke['joketext'], ENT_QUOTES, 'UTF-8')?>

  (by <a href="mailto:<?=htmlspecialchars($joke['email'], ENT_QUOTES,'UTF-8');
  ➥ ?>"><?=htmlspecialchars($joke['name'],ENT_QUOTES, 'UTF-8'); ?></a>)

  <a href="editjoke.php?id=<?=$joke['id']?>">Edit</a>

  <form action="deletejoke.php" method="post">
    <input type="hidden" name="id" value="<?=$joke['id']?>">
    <input type="submit" value="Delete">
  </form>
  </p>
</blockquote>
<?php endforeach; ?>
```

Each joke is linked to `editjoke.php?id=`, followed by the ID of the joke you wish to edit. When `editjoke.php` runs, the joke with the specified `id` is loaded from the database and used in the form.

Try this example (**Example: Structure-EditJoke**[11]) and load up the jokes list. Each joke will have an **Edit** link after the author's name, and by clicking the link, the joke text will be loaded into the form. When you press **Save**, it will trigger the update query by calling the `updateJoke` function.

11. https://github.com/spbooks/phpmysql7/tree/Structure-EditJoke

Delete Function

The same thing can now be done with *delete* to allow deletion of jokes in the same quick and easy way with a single line of code:

```
function deleteJoke($pdo, $id) {

  $stmt = $pdo->prepare('DELETE FROM `joke` WHERE `id` = :id');

  $values = [
    ':id' => $id
  ];

  $stmt->execute($values);
}
```

Then call the function to delete a joke with a specific ID from the database:

```
// Delete a joke with the ID of 2
deleteJoke($pdo, 2);
```

Add the function to *DatabaseFunctions.php* and amend *deletejoke.php* to use the new function:

```
<?php

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../includes/DatabaseFunctions.php';

    deleteJoke($pdo, $_POST['id']);

    header('location: jokes.php');
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Unable to connect to the database server: ' . $e->getMessage() .
    ↪ ' in ' .
    $e->getFile() . ':' . $e->getLine();
```

```
}

include  __DIR__ . '/../templates/layout.html.php';
```

Select Function

We can apply the same logic to fetching all the jokes from the `joke` table by adding an `allJokes` function to `DatabaseFunctions.php` and amending `jokes.php` accordingly:

```
function allJokes($pdo) {
  $stmt = $pdo->prepare('SELECT `joke`.`id`, `joketext`, `name`, `email`
    FROM `joke` INNER JOIN `author`
      ON `authorid` = `author`.`id`');

  $stmt->execute();

  return $stmt->fetchAll();
}
```

In this instance, the query is more complex than the one-liners used for `deleteJoke` and `updateJoke`. It's the same query we already had in `jokes.php` for selecting the jokes and the information about the author.

Here I've used PDO's `fetchAll` function. This returns an array of all records that were retrieved by the query. You can use the new `allJokes` function like this:

```
$jokes = allJokes($pdo);

echo '<ul>';
foreach ($jokes as $joke) {
  echo '<li>' . $joke . '</li>';
}
echo '</ul>';
```

This will print out all the jokes from the database in a list.

Update the `jokes.php` controller to use the new function.

Example: Structure-Delete-AllJokes[12]

```
try {
  include __DIR__ . '/../includes/DatabaseConnection.php';
  include __DIR__ . '/../includes/DatabaseFunctions.php';

  $jokes = allJokes($pdo);

  $title = 'Joke list';

  $totalJokes = totalJokes($pdo);

  ob_start();

  include  __DIR__ . '/../templates/jokes.html.php';

  $output = ob_get_clean();
}
catch (PDOException $e) {
  $title = 'An error has occurred';

  $output = 'Database error: ' . $e->getMessage() . ' in ' .
  $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

We've now got a set of reusable functions that can be used anywhere we need to interact with the database table. Whether it's finding all the jokes, a single joke, or issuing an `INSERT`, `UPDATE` or `DELETE` query, it's possible to run the query without actually typing out all the code each time. Anywhere on the website we need to interact with the database table `jokes`, it's quick and easy using our new set of functions.

The Best Way

In this chapter, I've helped you to rise above the basic questions of what PHP

12. https://github.com/spbooks/phpmysql7/tree/Structure-Delete-AllJokes

can do for you, and begin to look for the *best* way to code a solution. Sure, you can approach many simple scripts as lists of actions you want PHP to do for you, but when you tackle site-wide issues such as database connections, shared navigation elements, visitor statistics, and access control systems, it really pays to be able to structure your code carefully.

We've now explored a couple of simple but effective devices for writing structured PHP code. Include files let you reuse a single piece of code across multiple pages of your site, greatly reducing the burden when you need to make changes. Writing your own functions to put in these include files lets you build powerful libraries of functions that can perform tasks as needed and return values to the scripts that call them. These new techniques will pay off in a big way in the rest of this book.

In Chapter 7, you'll learn how to refine these functions further and make them even more reusable, using some new techniques that include writing your own objects and classes.

Chapter 7

Improving the Insert and Update Functions

In the last chapter, I showed you how to break code up into easily reusable functions. This has several advantages:

- the code where the function is called is easier to read
- you can reuse the same function from anywhere

In this chapter, I'll take this a step further and show you how to make a function that could be used for *any* database table, and then show you how object-oriented programming can simplify this task even further.

Improving the Update Function

```
updateJoke($pdo, 1, 'Why did the programmer quit his job? He didn\'t get arrays'
↪, 1);
```

To run this function, all of the arguments for the function must be provided:

- joke ID
- joke text
- author ID

What if you just wanted to update the joke text, rather than the joke text and author ID? Or just update the joke's author? With the `updateJoke` function above, all the information needs to be provided each time.

A better way of doing this is to amend the function to take the field values as an array, with the keys representing the field names and the corresponding values representing the data to be stored in the database. For example:

```
updateJoke($pdo, [
    'id' => 1,
    'joketext' => 'Why did the programmer quit his job? He didn\'t get arrays']
);
```

Another example:

```
updateJoke($pdo, [
    'id' => 1,
    'authorId' => 4
]);
```

This is a lot nicer, as only the information being updated (and the primary key) need to be sent to the function. It also has the advantage of being easier to read: you can read this code and see exactly what each field is being set to. With the earlier version, you have to know which argument represents which field. If you applied that to a table with 10 or 20 fields, you'd have to get the order of the arguments exactly right and remember which field was at each position.

The process of rewriting a function in order to support more use cases or to become more generic is called **refactoring**—something we'll be doing often in the next couple of chapters!

To make the function take an array, it needs to be updated. Currently it looks like this, and takes an argument for each field:

```
function updateJoke($pdo, $jokeId, $joketext, $authorId) {
    $stmt = $pdo->prepare('UPDATE `joke` SET
        `authorId` = :authorId,
        `joketext` = :joketext
        WHERE `id` = :id');

    $values = [
        ':joketext' => $joketext,
        ':authorId' => $authorId,
        ':id' => $jokeId
    ];

    $stmt->execute($values);

}
```

Changing the function so that it can take an array as a second argument and run the query is less than straightforward, as the query above is expecting parameters for `:authorid`, `:joketext` and `:primaryKey`. What will happen if

they're not all provided? If you try it, you'll get an error.

To work around this error, we need to dynamically generate the query so it only contains the relevant fields (only the ones we actually want to update).

The `foreach` loop (which we learned about in Chapter 2) can loop over an array. Take the following code:

```
$array = [
    'id' => 1,
    'joketext' => 'Why was the empty array stuck outside? It didn\'t have any
    ↪ keys'
];

foreach ($array as $key => $value) {
    echo $key . ': ' . $value . ',';
}
```

This will print the following:

```
id: 1, joketext: Why was the empty array stuck outside? It didn’t have any keys,
```

We can use a `foreach` along with a couple of other tools to generate the relevant parts of the `UPDATE` query.

It's possible to use a `foreach` loop to produce the `UPDATE` query:

```
$array = [
    'id' => 1,
    'joketext' => '!false - it\'s funny because it\'s true'
];

$query = ' UPDATE `joke` SET ';

$updateFields = [];
foreach ($array as $key => $value) {
    $updateFields[] = '`' . $key . '` = :' . $key;
}
```

```
$query .= implode(', ', $updateFields);

$query .= ' WHERE `id` = :primaryKey';

echo $query;
```

The `.=` Operator

Note the use of the `.=` operator above. It adds to the end of the existing string, rather than overwriting it!

There are a few steps here. Firstly, an empty array (`$updateFields`) is created. Then, for each field being updated, the SQL is generated. By the end of the loop, the `$updateFields` array will look like this:

```
$updateFields = [
    'id = :id',
    'joketext => :jokeText'
];
```

Finally, after the loop, these are joined together using PHP's `implode` function, which creates a single string from an array, separating each part with the text provided as the first argument—in this case, a comma followed by a space: `', '`.

By the end of the code above, the `$query` variable will contain a string for the entire `UPDATE` query:

```
UPDATE `joke` SET `id` = :id, `joketext` = :joketext WHERE `id` = :primaryKey
```

The query is generated dynamically based on the contents of `$array`, and only the field names from the array will appear in the query.

You'll notice that the `id` field is included in the statement. This isn't needed,

but having it here won't make any difference, as no update will be done if we're setting a value to what it is already.

The following code can be used to generate the query with just the `authorId`:

```
$array = [
    'id' => 1,
    'authorid' => 4
];

$query = 'UPDATE `joke` SET ';

$updateFields = [];
foreach ($array as $key => $value) {
    $updateFields[] = '`' . $key . '` = :' . $key;
}

$query .= implode(', ', $updateFields);

$query .= ' WHERE `id` = :primaryKey';

echo $query;
```

The code above will print the following query:

```
UPDATE `joke` SET `id` = :id, `authorId` = :authorId WHERE `id` = :id
```

By placing this code into the function, the improved version of `updateJoke` can now be used:

```
function updateJoke($pdo, $values) {

    $query = ' UPDATE `joke` SET ';

    $updateFields = [];
    foreach ($values as $key => $value) {
        $updateFields[] = '`' . $key . '` = :' . $key;
    }
```

```
    $query .= implode(', ', $updateFields);

    $query .= ' WHERE `id` = :primaryKey';

    // Set the :primaryKey variable
    $values['primaryKey'] = $values['id'];

    $stmt = $pdo->prepare($query);
    $stmt->execute($values);
}
```

You'll notice I set the `primaryKey` key manually with this line:

```
// Set the :primaryKey variable
$values['primaryKey'] = $values['id'];
```

This is so that the `WHERE` clause in the query is provided with the relevant ID to update. `:id` can't be used, because it has already been used in the query, and each parameter needs a unique name.

With this version of the `updateJoke` function, it's now possible to run it as we designed earlier:

```
updateJoke($pdo, [
        'id' => 1,
        'joketext' => '!false - it\'s funny because it\'s true'
    ]
);
```

Or we could do this:

```
updateJoke($pdo, [
        'id' => 1,
        'authorId' => 4
    ]
);
```

Writing Functions

When you write a function, it's usually easier to write some examples of how you think it should be called before writing the code inside the function itself. This gives you a target to work towards, and some code you can run to see whether it's working correctly or not.

Improving the Insert Function

Using this knowledge, we can do the same thing for the `insertJoke` function.

Like we did with the update function, let's start by thinking about how it will look when it's called. The function will take an array that represents the joke being added like this:

```
insertJoke($pdo, [
    'authorId' => 4,
    'joketext' => '!false - it\'s funny because it\'s true',
    'jokedate' = '2021-04-22'
    ]
);
```

To make the `insertJoke` function work in this way, it will need to take an array and generate the `INSERT` query based on the fields provided, in the same way we just did for the `updateJoke` function.

The `INSERT` query that we're going to generate looks like this:

```
$query = 'INSERT INTO `joke` (`joketext`, `jokedate`, `authorId`)
    VALUES (:joketext, :jokedate, :authorId)';
```

There are two parts to this—the field names and the values. Firstly, let's handle the column names. As we did with the `updateJoke` function, we can use a loop and `implode` to create the list of fields for the first line of the query:

```
function insertJoke($pdo, $values) {
    $query = 'INSERT INTO `joke` (';

    $insertFields = [];
    foreach ($array as $key => $value) {
        $updateFields[] = '`' . $key . '`';
    }

    $query .= implode(', ', $updateFields);

    $query .= ') VALUES (';
}
```

The `foreach` loops over each of the `$values` array and uses the key from the array (which contains the field name) to generate a list of comma-separated fields.

This will generate the first part of the query, and `$query` will store the following:

```
INSERT INTO `joke` (`authorId`, `:jokedate`, `joketext`) VALUES (
```

… when called with this array:

```
[
    'authorId' => 4,
    'joketext' => '!false - it\'s funny because it\'s true'
    'jokedate' = '2021-04-22'
]
```

The next part of the query should be the placeholders for the values:

```
VALUES (:authorId, :joketext)
```

These are the keys prefixed with a colon (`:`). Once again, we can use `foreach` to loop through the column names and add the placeholders before sending the query to the database:

```
function insertJoke($pdo, $values) {
    $query = 'INSERT INTO `joke` (';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '`,';
    }

    $query = rtrim($query, ',');

    $query .= ') VALUES (';

    foreach ($values as $key => $value) {
        $query .= ':' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ')';

    $stmt = $pdo->prepare($query);

    $stmt->execute($values);

}
```

The `insertJoke` function can now be used to insert a joke using an array specifying which value will be placed in which column:

```
insertJoke($pdo, [
    'authorId' => 4,
    'joketext' => '!false - it\'s funny because it\'s true',
    'jokedate' = '2021-04-22'
    ]
);
```

Of course, it will cause an error if the wrong column names are used, but it's clearer to anyone seeing this code which data is being placed into each of the columns. As we'll see later, it also makes it easier to use the function with forms.

For now, amend your website to use the two new functions by replacing the

existing ones in `DatabaseFunctions.php`. Then amend `editjoke.php` to use the new `updateJoke` function:

```
updateJoke($pdo, [
    'id' => $_POST['jokeid'],
    'joketext' => $_POST['joketext'],
    'authorId' => 1
]);
```

Finally, amend `addjoke.php` to use the new function.

Structure2-ArrayFunctions-Error[1]

```
insertJoke($pdo, [
                    'authorId' => 1,
                    'joketext' => $_POST['joketext'],
                    'jokedate' => date('Y-m-d')
            ]
 );
```

Handling Dates

You might be wondering how you could set a specific date, rather than the current date, when inserting a date into the database.

Previously, we used the `date()` function. Using the new `insertJoke` function, you may attempt to set the joke like so:

```
insertJoke($pdo, ['authorId' => 1,
    'jokeText' => $_POST['joketext'],
    'jokedate' =>  date('Y-m-d')
]);
```

The `date()` function is simple and easy to use when we want to use the system date, but if we want to easily set a specific date, we can use a different

---

1. https://github.com/spbooks/phpmysql7/tree/Structure2-ArrayFunctions-Error

tool. The PHP `DateTime` class can be used to represent dates in PHP and format them in any way. For example:

```
$date = new DateTime();

echo $date->format('d/m/Y H:i:s');
```

Like the simpler `date()` function, by default a new instance of `DateTime` (without a provided argument) will represent today's date, and the string in the `format` method is the same as the format used in the `date()` function you're already familiar with. If you were reading this on September 8, 2021 at around 7.00 p.m., this would print something like this:

```
08/09/2021 19:12:34
```

The biggest advantage of the `DateTime` class over the `date` function is that you can give it a string that represents any date and have it convert it to a different date format. For example:

```
$date = new DateTime('5th March 2021');

echo $date->format('d/m/Y');
```

The string `d/m/Y` here represents day/month/year, which would print `05/03/2021`. The PHP `DateTime` class is very powerful and incredibly useful for handling dates in PHP. A full list of the available methods of formatting a date is available in the PHP manual[2].

The `DateTime` class can be used to represent any date and time. Today's date could be induced in this format, like so:

```
// Don't give it a date so it uses the current date/time
$date = new DateTime();
```

2. http://php.net/manual/en/function.date.php

```
echo $date->format('Y-m-d H:i:s');
```

The above will print something like this:

```
2021-09-08 19:16:34
```

This is the format that's needed when you insert into a `DATETIME` field in MySQL.

The insert function can already handle this. We can just pass the current date as one of the array keys to the function:

```
$date = new DateTime();

insertJoke($pdo, [
    'authorId' => 4,
    'joketext' => 'Why did the chicken cross the road? To get to the other side',
    'jokedate' => $date->format('Y-m-d H:i:s')
    ]
);
```

This will work fine. However, this isn't really any different from just using the `Date` function, and every time we want to use the function, we need to remember the date format used by MySQL. This can be a bit of a pain, as it requires repeating the code for formatting the date (and remembering the string `Y-m-d H:i:s`) each time a date needs to be inserted.

Instead, a better approach would be to have the function format the date for us, saving some work each time it's called:

```
function insertJoke($pdo, $values) {
    $query = 'INSERT INTO `joke` (';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '`,';
    }
```

```
    $query = rtrim($query, ',');

    $query .= ') VALUES (';

    foreach ($values as $key => $value) {
        $query .= ':' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ')';

    foreach ($values as $key => $value) {
        if ($value instanceof DateTime) {
            $values[$key] = $value->format('Y-m-d');
        }
    }

        $stmt = $pdo->prepare($query);

        $stmt->execute($values);
}
```

Using the above version of the function, a date *object* can be passed into the function without formatting it:

```
insertJoke($pdo, [
    'authorId' => 4,
    'joketext' => '!false - it\'s funny because it\'s true',
    'jokedate' =>  new DateTime()
    ]
);
```

The function will look for any date objects it's been given and automatically format them to the format needed by MySQL. When the function is called, you don't need to remember the exact format needed by MySQL; it's done for you inside the function. The function will automatically convert any `DateTime` object it comes across into a string that MySQL can understand:

```
// Loop through the array of fields
```

```
foreach ($values as $key => $value) {
    // If any of the values are a DateTime object
    if ($value instanceof DateTime) {
        // Then replace the value in the array with the date in the format
        // Y-m-d H:i:s
        $values[$key] = $value->format('Y-m-d H:i:s');
    }
}
```

One operator you haven't come across yet is the `instanceof` operator. This is a comparison operator like `==` or `!=`. However, instead of checking to see if the two values are the *same*, it checks to see whether the variable on the left (`$value`) is the same kind of object as the one on the right (`DateTime`). One important distinction is that the *value* is not being checked, only the *type*. It's like saying "Is it a car?" rather than "Is it an E-Type Jaguar?"

Wherever a `DateTime` object is found in the array, it's replaced with the equivalent date string in the format MySQL needs. This allows us to completely forget about the format MySQL actually needs and just supply `DateTime` objects to the `insertJoke` function.

With this code in place, if we wanted to specify a date to be set in the database, we could do it like so:

```
insertJoke($pdo, [
        'authorId' => 4,
        'joketext' => '!false - it\'s funny because it\'s true',
        'jokedate' =>  new DateTime('5th September 2021')
        ]
);
```

The `insertJoke` function will see the `DateTime` instance and automatically convert it to the format MySQL uses.

Let's also add the automatic date formatting to the `updateJoke` function:

```
function updateJoke($pdo, $values) {
```

```
    $query = ' UPDATE joke SET ';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '` = :' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ' WHERE id = :primarykey';

    foreach ($values as $key => $value) {
        if ($value instanceof DateTime) {
            $values[$key] = $value->format('Y-m-d H:i:s');
        }
    }

    // Set the :primaryKey variable
    $values['primaryKey'] = $values['id'];

        $stmt = $pdo->prepare($query);
        $stmt->execute($values);
}
```

This will take the same approach we used in `insertJoke` and apply it to the update function. Any `DateTime` object that's passed to the function in the `$values` array will be converted to a string that MySQL understands.

As we've copied/pasted code, it's good practice to move the duplicated code into its own function to save repetition and follow the DRY principle:

```
function processDates($values) {
    foreach ($values as $key => $value) {
        if ($value instanceof DateTime) {
            $values[$key] = $value->format('Y-m-d H:i:s');
        }
    }

    return $values;
}
```

This function takes an array called `$values`, loops through them, and replaces any instance of `DateTime` with the formatted date. Importantly, it returns the array that's been amended with the updated values. The function is called like so:

```
$values = processDates($values);
```

When this happens, the `$values` array will be *copied* into the `processDates` function and a copy of the amended array will be *returned*. By using `$values =`, the original `$values` array is being replaced with the one that's had the `DateTime` instances replaced in the `processDates` function.

We could call the function like so:

```
processDates($values);
```

If we did, the `$values` array would remain the same as it was in the `insertJoke` function. The updated array only exists in the `processDates` function, so we have to copy the updated array out of the function using `return`.

Avoid Copy/pasting

If you ever find yourself reaching for `ctrl` + `c` and `ctrl` + `v`, you're probably better moving the code to a function with the relevant arguments as we just did here. That way, if there's a bug in the code or you have to amend it, you only have to make the change in one place!

Let's put the function in `DatabaseFunctions.php` and change the `updateJoke` and `insertJoke` functions to make use of it:

```
function insertJoke($pdo, $values) {
    $query = 'INSERT INTO `joke` (';
```

```
    foreach ($values as $key => $value) {
        $query .= '`' . $key . '`,';
    }

    $query = rtrim($query, ',');

    $query .= ') VALUES (';

    foreach ($values as $key => $value) {
        $query .= ':' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ')';

    $values = processDates($values);

        $stmt = $pdo->prepare($query);
        $stmt->execute($values);
}

function updateJoke($pdo, $values) {

    $query = ' UPDATE `joke` SET ';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '` = :' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ' WHERE `id` = :primaryKey';

    // Set the :primaryKey variable
    $values['primaryKey'] = $values['id'];

    $values = processDates($values);

        $stmt = $pdo->prepare($query);
        $stmt->execute($values);
}
```

Finally, amend `addjoke.php` to provide the `DateTime()` object for the value of the `jokedate` column.

Structure2-ArrayFunctions-Dates[3]

```
insertJoke($pdo, ['authorId' => 1,
    'jokeText' => $_POST['joketext'],
    'jokedate' => new DateTime()
]);
```

Displaying the Joke Date

While we're dealing with dates, let's display the date on which the joke was posted, alongside the author's name in the template. The page will now display something like "By Tom Butler on 2021-08-04".

Firstly, we'll need to amend the `allJokes` function to retrieve the date from the database:

```
function allJokes($pdo) {
    $stmt =  $pdo->prepare($pdo, 'SELECT `joke`.`id`, `joketext`, `jokedate`,
    ↪ `name`, `email`
        FROM `joke` INNER JOIN `author`
        ON `authorid` = `author`.`id`');

        $stmt->execute();

    return $stmt->fetchAll();
}
```

Now we can reference the date column in the `jokes.html.php` template file:

```
<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>

<?php foreach ($jokes as $joke): ?>
```

3. https://github.com/spbooks/phpmysql7/tree/Structure2-ArrayFunctions-Dates

```
<blockquote>
    <p>
    <?=htmlspecialchars($joke['joketext'], ENT_QUOTES, 'UTF-8')?>

    (by <a href="mailto:<?=htmlspecialchars(
        $joke['email'],
        ENT_QUOTES,
                'UTF-8'
); ?>">
    <?=htmlspecialchars(
                        $joke['name'],
                        ENT_QUOTES,
                'UTF-8'
                ); ?></a> on <?=$joke['jokedate']; ?>)
    <a href="editjoke.php?id=<?=$joke['id']?>">Edit</a>

    <form action="deletejoke.php" method="post">
        <input type="hidden" name="id" value="<?=$joke['id']?>">
        <input type="submit" value="Delete">
    </form>

    </p>
</blockquote>
<?php endforeach; ?>
```

If you run the code above, you'll see the date printed on the page. Unfortunately, it will be printed in the format that MySQL uses—that is, `2019-08-04`, rather than a nicer format that people who are viewing the website would prefer.

It's possible to use the `DateTime` class to do this for us, by amending the template file to create an instance of the `DateTime` class and formatting the date in a nicer way:

```
(by <a href="mailto:<?=htmlspecialchars($joke['email'],  ENT_QUOTES, 'UTF-8');
  ➥ ?>">
    <?=htmlspecialchars($joke['name'], ENT_QUOTES, 'UTF-8'); ?></a> on
<?php
$date = new DateTime($joke['jokedate']);
```

```
echo $date->format('jS F Y');
?>)
    <a href="editjoke.php?id=<?=$joke['id']?>">Edit</a>
```

The `DateTime` class can take an argument of a date and, luckily for us, it understands the format MySQL uses—allowing us to quickly format the data in any way we like.

The line `echo $date->format('jS F Y');` formats the date in a much nicer way, and will display something like `4th August 2019`.

This example can be found in **Structure2-ArrayFunctions-Dates2**.

Making Your Own Tools

> *Blacksmiths are unique in that they make their own tools. Carpenters don't make their saws and hammers, tailors don't make their scissors and needles, and plumbers don't make their wrenches, but blacksmiths can make their hammers, tongs, anvils, and chisels — Daniel C. Dennet,* ***Intuition Pumps and Other Tools for Thinking***

This is an interesting observation (and I'll explain why I've included it in a moment), but it implies a small paradox. If a blacksmith is needed to make a blacksmith's tools, where did the first blacksmith's tools come from? As it turns out, there's actually a simple explanation: the first tools were incredibly crude—just some rocks used to bash out some metal rods. But the blacksmith can fuse two of those rods together to make a basic hammer, and then use that hammer to make an even better hammer, gradually making newer and better equipment.

By improving the tools to be more precise and easier to use, the blacksmith can make products faster, create higher-quality items, produce a wider variety of products, make other specialist tools, and let lesser-skilled workers such as

apprentices produce products beyond their skill level.

It does take time to create a tool, but once the tool is created, the blacksmith can use it to make thousands of products. Over the long term, the time spent making the tool quickly pays off.

You're probably wondering what any of this has to do with PHP programming. I'm going to amend Dennet's quote slightly and say that blacksmiths are *almost* unique in being able to make their own tools—because programmers also possess this ability.

Every opportunity I listed for blacksmiths to create their own tools exists for programmers as well. In fact, everything you use on your computer is a tool written by another programmer. Even the PHP programming language you're using is a tool originally written by a developer called Rasmus Lerdorf.

Programming languages don't just suddenly exist. Computers don't even understand PHP at all! They only understand binary code.

Like blacksmiths, programmers started with very crude tools: punch cards that had a hole or no hole to represent a one or a zero. These were then manually fed into the computer to program it. Writing and understanding the code took an incredible amount of skill. As computers developed, so did the way we program them.

Instead of representing everything as ones and zeros with punch cards, programmers invented tools that take human-readable and easier-to-understand code and convert it (or "compile" it, if you want to get technical) into the binary code that the computer understands. It's easy to forget that programming languages exist for humans, not for computers.

When you sit down in front of your computer with your favorite code editor, you can be forgiven for thinking you're using the most refined hammer that can be made and that you have all the tools available to pick from.

A programming language is just a tool, and you can use it to make your own

tools. Every time you write a function, you're creating a new tool. You can either make tools that have many uses, which you can use over and over again, or tools with limited use that can only be used for one very specific job.

In your kitchen you have many tools. You probably need a knife to prepare nearly every meal, but how often do you use a garlic press?

A tool is more useful if it can be used for a variety of different tasks. In the kitchen, the more recipes the tool can help make, the more useful it is.

When writing a function in PHP (or any programming language), strive to write functions that are more like the knife than the garlic press. Write functions that can be used over and over again on any website, rather than functions that only apply to a very specific case on a single website.

So far in this chapter, we've taken functions that need an exact number of arguments in a very specific order and rewritten them as functions that allow arguments to be specified in (almost) any order, with values optionally omitted entirely if they don't need updating.

The problem with the function now is that it's more like the garlic press than the knife. Just as the garlic press is only useful for recipes that have garlic, the `updateJoke` function is only useful when we want to update a record in the `joke` table. We can't take this function and use it on another website, because the next website you build probably won't even have a table called `joke`.

The next step in refining our tools is to make our functions more like the knife, able to work with any database table. After all, a knife can be used to chop garlic anyway.

Generic Functions

Before we make any large-scale changes, let's expand the website and add a function for retrieving all the authors from the database in the same manner we used for the `allJokes` function:

```
function allAuthors($pdo) {
    $stmt =  $pdo->prepare('SELECT * FROM `author`');
    $stmt->execute();
    return $stmt->fetchAll();
}
```

Let's also add functions for inserting and deleting authors from the `author` table:

```
function deleteAuthor($pdo, $id) {
    $values = [':id' => $id];

    $stmt = $pdo->prepare('DELETE FROM `author` WHERE `id` = :id');

    $stmt->execute($values)
}
```

```
function insertAuthor($pdo, $values) {
    $query = 'INSERT INTO `author` (';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '`,';
    }

    $query = rtrim($query, ',');

    $query .= ') VALUES (';

    foreach ($values as $key => $value) {
        $query .= ':' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ')';

    $values = processDates($values);

    $stmt = $pdo->prepare($query);
    $stmt->execute($values);
}
```

You'll notice that these `deleteAuthor` and `insertAuthor` functions are almost identical to the corresponding joke functions, `deleteJoke` and `insertJoke`. It would be better to create *generic* functions that could be used with any database table. That way, we can write the function *once* and use it for any database table. If we continued down the route of having five different functions for each database table, we'd very quickly end up with a lot of very similar code.

The differences between the functions are just the names of the tables. By replacing the table name with a variable, the function can be used to retrieve all the records from any database table. That way, it's no longer necessary to write a different function for each table. Let's start with a very simple example—a function that can be used to find all the records from any table:

```
function findAll($pdo, $table) {
    $stmt = $pdo->prepare('SELECT * FROM `' . $table . '`');
        $stmt->execute();

    return $stmt->fetchAll();
}
```

This function differs in one major way from those we've created so far: the name of the table is an argument, which is then used in the query.

Once the function has been written, this new tool can be used to retrieve all the records from any database table:

```
// Select all the jokes from the database
$allJokes = findAll($pdo, 'joke');

// Select all the authors from the database
$allAuthors = findAll($pdo, 'author');
```

By replacing the table name with an argument, the same thing can be done to convert the `deleteJoke` function into a `delete` function that can be used with any table on any website:

```
function delete($pdo, $table, $id) {
    $values = [':id' => $id];

    $stmt = $pdo->prepare('DELETE FROM `' . $table . '` WHERE `id` = :id');

        $stmt->execute($values);
}
```

This allows for deleting a record from any table based on its ID:

```
// Delete author with the ID of 2
delete($pdo, 'author', 2);

// Delete joke with the id of 5
delete($pdo, 'joke', 5);
```

This function works, but it's still a little inflexible. It assumes that the primary key field in the table is called `id`. This function can only work with tables that have a field named `id` for their primary key, which isn't always the case. A table that stored information about books, for example, might have `isbn` as the primary key. In order for our function to work with any database table structure, the name of the column used in the `WHERE` clause of the query can also be replaced with a variable:

```
function delete($pdo, $table, $field, $value) {
        $values = [':value' => $value];

        $stmt = $pdo->prepare('DELETE FROM `' . $table . '` WHERE `' . $field . '` = :value');

        $stmt->execute($values);
}
```

Whenever the `delete` function is called, it's now supplied with four arguments:

- the `$pdo` database connection
- the name of the table to delete a record from
- the ID of the record to delete

- the name of the field to use

And it can be called like this:

```
// Delete author with the ID of 2
delete($pdo, 'author', 'id', 2);

// Delete joke with the id of 5
delete($pdo, 'joke', 'id', 5);

// Delete the book with the ISBN 9780994346988
delete($pdo, 'book', '9780994346988', 'isbn');
```

Like any deletion functionality, you'll need to be very careful with this function! The code `delete($pdo, 'author', 'name', 'John')` will execute the query `DELETE FROM author WHERE name = "John"`.

In addition to the `delete` and `findAll` functions, let's do the same thing with the `update` and `insert` functions, by replacing the table name with a function argument:

```
function insert($pdo, $table, $values) {
    $query = 'INSERT INTO `' . $table . '` (';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '`,';
    }

    $query = rtrim($query, ',');

    $query .= ') VALUES (';

    foreach ($values as $key => $value) {
        $query .= ':' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ')';
```

```
    $values = processDates($values);

        $stmt = $pdo->prepare($query);
        $stmt->execute($values);
}

function update($pdo, $table, $primaryKey, $values) {

    $query = ' UPDATE `' . $table .'` SET ';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '` = :' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ' WHERE `' . $primaryKey . '` = :primaryKey';

    // Set the :primaryKey variable
    $values['primaryKey'] = $values['id'];

    $values = processDates($values);

        $stmt = $pdo->prepare($query);
        $stmt->execute($values);
}
```

These functions are nearly identical to the `insertJoke` and `updateJoke` functions we created earlier. The only difference is that the table name has been replaced with a variable—so that they can be used to insert or update records in any table.

Notice that, in the `update` function, I've also created a variable called `primaryKey`, as I did with `delete`, because we can't assume that the primary key (used in the `WHERE` id `= :primaryKey` part of the query) will always be `id`.

In the last chapter, we created a function called `getJoke`, which found a specific joke by its ID. That function, too, can be made to work with any database table. This `findById` function can be used to find a single record from any table using the primary key:

```
function findById($pdo, $table, $primaryKey, $value) {
    $stmt = 'SELECT * FROM `' . $table . '` WHERE `' . $primaryKey . '` =
    ↪:value';

    $values = [
        'value' => $value
    ];

        $stmt = $pdo->prepare($query);
        $stmt->execute($values);
    return $stmt->fetch();
}
```

Before jumping ahead and using this function, it's worth considering whether this function can be improved. The code above allows finding any single record from the database using the field `$primaryKey`.

However, thinking ahead a little, that's quite limiting. What if we wanted to find all the jokes that were posted on a particular day, or all the jokes by the author with the name of "John"? For that, we'd need a query in the format `SELECT * FROM joke WHERE authorName = "John"`.

The function almost already supports this behavior, because although the variable is called `$primaryKey`, we could actually pass `authorName` as the argument. So, rather than having a function called `findById`, let's create an even more generic function called `find`. It looks like this:

```
function find($pdo, $table, $field, $value) {
    $query = 'SELECT * FROM `' . $table . '` WHERE `' . $field . '` = :value';

    $values = [
        'value' => $value
    ];

    $stmt = $pdo->prepare($query);
    $stmt->execute($values);
    return $stmt->fetchAll();
}
```

There are some very subtle differences here:

- The function has been given a more descriptive name—`find`—instead of `findById`, to make it clearer that it doesn't just find records by their ID.
- The `$primaryKey` variable has been renamed to `$field` for the same reason.
- The `return` statement now returns `$stmt->fetchAll()` instead of `$stmt->fetch()`. This is an important change, as it will find *all* the records that are returned by a query.

By having the `find` function return a list of all the records, it can be used when we want a list of records matching the search criteria—such as all the records by a particular author, or all the jokes posted on a particular date.

In addition, it can still be used when we want to find a record by its ID—with one small caveat. To find the joke with the ID of `123`, the function can be called like so:

```
$joke = find($pdo. 'joke', 'id', 123);

echo $joke['jokeText'];
```

However, this won't quite have the desired result. Because the `find` function returns a list (or more accurately, *an array*) of all the records that match the search criteria, the `$joke` variable here is an array of *all* the jokes with the ID of `123`.

In this case, it's an array containing a single joke. However, this has a practical implication. The line `echo $joke['jokeText'];` won't work as intended, because the `$joke` variable is a list of all the jokes, not a single joke.

As a demonstration, if we wanted to manually create the `$joke` variable using PHP code, it would look something like this:

```
$joke = [];

$joke[0] = [
    'id' => 123
    'jokeText' => 'Why did the programmer quit his job? He didn\'t get arrays',
    'jokeDate' => '2021-09-24',
];
```

That's an array inside an array (also known as a **multidimensional array**). Index zero contains an array representing the data about the first joke. Therefore, any time we want to use the `find` function to find a single joke, we need to extract the first joke from the array:

```
$jokes = find($pdo. 'joke', 'id', 123);

$joke = $joke[0];

echo $joke['jokeText'];
```

Alternatively, we can skip the middle line by extracting the first joke directly after the call to the function:

```
$joke = find($pdo. 'joke', 'id', 123)[0];

echo $joke['jokeText'];
```

This is equivalent to the previous code, except that the `[0]` is included directly after finding the list of jokes. I'll be using this shorter notation throughout the rest of the book wherever we want to retrieve one record using the `find` method.

With the `find` function completed, we now have a set of functions that can be used to interact quickly and easily with *any* database table and use PHP's `DateTime` class for dates:

```
// Add a new record to the database
$record = [
```

```
    'joketext' => '!false - it\'s funny because it\'s true',
    'authorId' => 2,
    'jokedate' => new DateTime()
];

insert($pdo, 'joke', $record);

// Delete from the author table where `id` is `2`
delete($pdo, 'author', 'id', 2);

$jokes = findAll($pdo, $joke);
```

Finally, for the sake of completeness, let's also make the same change to the *totalJokes* function, allowing us to get the number of records in any table:

```
function total($pdo, $table) {
    $stmt = $pdo->prepare('SELECT COUNT(*) FROM `' . $table . '`');
        $stmt->execute();
    $row = $stmt->fetch();
    return $row[0];
}
```

Using These Functions

Now that we've got the functions, let's put them into our controllers. Firstly, we'll change the *addjoke.php* controller to use the new generic *insert* function. It currently looks like this:

```
<?php
if (isset($_POST['joketext'])) {
    try {
        include __DIR__ . '/../includes/DatabaseConnection.php';
        include __DIR__ . '/../includes/DatabaseFunctions.php';

        insertJoke($pdo, [
                'authorId' => 1,
                'jokeText' => $_POST['joketext'],
                'jokedate' => new DateTime()
                ]
```

```
            );

        header('location: jokes.php');

        // …
```

To replace the `insertJoke` function with the generic `insert` function, we just need to change the name and supply the name of the table:

```
<?php
if (isset($_POST['joketext'])) {
    try {
        include __DIR__ . '/../includes/DatabaseConnection.php';
        include __DIR__ . '/../includes/DatabaseFunctions.php';

        insert($pdo, 'joke', [
                'authorId' => 1,
                'jokeText' => $_POST['joketext'],
                'jokedate' => new DateTime()
                ]
        );

        header('location: jokes.php');

        // …
```

Now amend the `editjoke.php` and `deletejoke.php` files in the same way.

Firstly, `editjoke.php`. Remember to replace the `updateJoke` function with the generic `updateFunction`, and also the `findJoke` function with the generic `find` function, as follows:

```
<?php
include __DIR__ . '/../includes/DatabaseConnection.php';
include __DIR__ . '/../includes/DatabaseFunctions.php';

try {
        if (isset($_POST['joketext'])) {
                update($pdo, 'joke', 'id', [
                        'id' => $_POST['jokeid'],
```

```
                'joketext' => $_POST['joketext'],
                'authorId' => 1
            ]
        );

        header('location: jokes.php');
    } else {
        $joke = find($pdo, 'joke', 'id', $_GET['id'])[0];

        $title = 'Edit joke';

        ob_start();

        include  __DIR__ . '/../templates/editjoke.html.php';

        $output = ob_get_clean();
    }
}
// …
```

And here's the updated `deletejoke.php`:

```
<?php
try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../includes/DatabaseFunctions.php';

    delete($pdo, 'joke', 'id', $_POST['id']);

    // …
```

The next part is the list of jokes. Currently, it uses the `allJokes` function, which also retrieves information about the author of each joke. There's no simple way to write a generic function that retrieves information from two tables. It would have to know which fields were used to join the tables and which tables were being joined. It would be possible to do this with a function with lots of arguments, but the function would be difficult to use and overly complex.

Instead, we can combine the use of the generic `findAll` and `find` functions

to achieve this:

```
$result = findAll($pdo, 'joke');

$jokes = [];
foreach ($result as $joke) {

    $author = find($pdo, 'author', 'id', $joke['authorId'])[0];

    $jokes[] = [
        'id' => $joke['id'],
        'joketext' => $joke['joketext'],
        'name' => $author['name'],
        'email' => $author['email']
    ];
}
```

The complete `jokes.php` is shown below.

Example: Structure2-GenericFunctions[4]

```
<?php

try {
        include __DIR__ . '/../includes/DatabaseConnection.php';
        include __DIR__ . '/../includes/DatabaseFunctions.php';

        $result = findAll($pdo, 'joke');

        $jokes = [];
        foreach ($result as $joke) {
                $author = find($pdo, 'author', 'id', $joke['authorId'])[0];

                $jokes[] = [
                        'id' => $joke['id'],
                        'joketext' => $joke['joketext'],
                        'jokedate' => $joke['jokedate'],
                        'name' => $author['name'],
                        'email' => $author['email']
```

4. https://github.com/spbooks/phpmysql7/tree/Structure2-GenericFunctions

```
                ];
        }

        $title = 'Joke list';

        $totalJokes = total($pdo, 'joke');

        ob_start();

        include  __DIR__ . '/../templates/jokes.html.php';

        $output = ob_get_clean();
} catch (PDOException $e) {
        $title = 'An error has occurred';

        $output = 'Database error: ' . $e->getMessage() . ' in ' .
        $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

This works by fetching the list of jokes (without the author information), then looping over each joke and finding the corresponding author by their `id`, then writing the complete joke with the information from both tables into the `$jokes` array. This is essentially what an `INNER JOIN` does in MySQL.

Each joke is now made up of values from the `$author` variable, which is a record from the `author` table and values from the `joke` table.

You may have realized that this method is going to be slower, because more queries are sent to the database. This is a common issue with these kinds of generic functions, and it's called the **N+1 problem**. There are several methods for reducing this performance issue, but for smaller sites, where we're dealing with hundreds or thousands of records rather than millions, it's unlikely to cause any real problems. The difference will likely be in the region of milliseconds.

The N+1 Problem

For more information on the N+1 problem, refer to the Microsoft article "Select N+1 Problem – How to Decrease Your ORM Performance"[5], which explains it in detail.

To see how the N+1 problem can be avoided entirely, refer to "The (Silver) Bullet for the N+1 Problem"[6].

Repeated Code Is the Enemy

Whenever you're writing software, you need to be vigilant about repeated code. Any time you find yourself with two or more copies of identical or very similar code, it's always worth taking a step back and looking to see if there's a way of merging the code into one reusable block.

By creating the generic functions `insert` , `update` , `delete` , `findAll` and `find` , it's now very quick and easy for us to create a website that deals with any kind of database operation. The functions can be used to interact very simply with any database table we might ever need.

But there's still room for improvement. The files `addjoke.php` and `editjoke.php` do very similar jobs: they display a form, and when the form is submitted, they send the submitted data off to the database.

Similarly, the template files `addjoke.html.php` and `editjoke.html.php` are almost identical. There's very little difference between them. The only real difference is that the edit page displays a pre-filled form, while the add page presents an empty form.

The problem with repeated code is that, if something has to change, you have to make the same change in multiple locations. What if we wanted to add a category to a joke, so that when a joke was added or edited the user could

5. https://dzone.com/articles/select-n1-problem-%E2%80%93-how
6. https://www.sitepoint.com/silver-bullet-n1-problem/"

choose *knock-knock jokes*, *programming jokes*, *puns*, *one-liners*, and so on?

We could achieve this by adding a `<select>` box to the add joke page, but we'd also need to make the same change to the edit joke page. Each time we made a change to `addjoke.php`, we'd need to make the corresponding change in `editjoke.php`.

If you ever find yourself in a situation like this, where you have to make similar changes in multiple files, it's a good sign that you should combine both sets of code into one. Of course, the new code needs to handle both cases.

There are a couple of differences between `addjoke.php` and `editjoke.php`:

- `addjoke.php` issues an `INSERT` query, while `editjoke.php` issues an `UPDATE` query.
- `editjoke.php`'s template file has a hidden input that stores the ID of the joke being edited.
- When the edit joke form is displayed, the information about the joke being edited is loaded into the form before it's displayed.

But everything else is almost the same.

Let's merge both pieces of code together, so that `editjoke.php` can handle both editing an existing joke and adding a new one. The script will be able to tell whether we're adding or editing based on whether or not an ID is supplied in the URI.

Visiting `editjoke.php?id=12` will load the joke with the ID 12 from the database and allow us to edit it. When the form is submitted, it will issue the relevant `UPDATE` query, while just visiting `editjoke.php`—without an ID specified—will display an empty form and, when submitted, perform an `INSERT` query.

Creating a Page for Adding and Editing

Let's handle the form first—which either loads the joke into the fields or displays a blank form. Currently, `editjoke.php` assumes there's the `id` URL variable set, and loads the joke accordingly before loading the template file:

```
else {
    $joke = find($pdo, 'joke', 'id', $_GET['id'])[0];

    $title = 'Edit joke';

    ob_start();

    include  __DIR__ . '/../templates/editjoke.html.php';

    $output = ob_get_clean();
}
```

This can be replaced with an `if` statement to only load the joke from the database if an `id` has actually been provided:

```
else {
    if (isset($_GET['id'])) {
        $joke = find($pdo, 'joke', 'id', $_GET['id']);
    }
    else {
      $joke = null;
    }


    $title = 'Edit joke';

    ob_start();

    include  __DIR__ . '/../templates/editjoke.html.php';

    $output = ob_get_clean();
}
```

If you try the above code and visit `editjoke.php` without supplying an ID as a

`GET` variable, it won't quite work as intended. You'll actually see some strange errors appearing inside the `<textarea>`. That's because, when `editjoke.html.php` is loaded, it references a variable called `$joke` which, because of the new `if` statement, is only created when the ID is supplied.

One solution would be to load the `addjoke.html.php` file if there's no ID set and `editjoke.html.php` if there is. Unfortunately, this doesn't solve the initial problem: to add a new field to the form using this approach, we'd still need to edit two files.

Instead, let's amend `editjoke.html.php` so it only tries to print the existing data into the textarea and hidden input if the joke variable is set:

```
<form action="" method="post">
    <input type="hidden" name="jokeid" value="<?php if (isset($joke)): ?>
                        <?=$joke['id']?>
                <?php endif; ?>
        ?>">
        <label for="joketext">Type your joke here:</label>
        <textarea id="joketext" name="joketext" rows="3" cols="40"><?php if
        ↳ (isset($joke)): ?>
                <?=$joke['joketext']?>
        <?php endif; ?></textarea>
        <input type="submit" name="submit" value="Save">
</form>
```

Now the `joketext` and `id` will only be written to the page if the `$joke` variable has been set, and it's only set when we're editing an existing record.

Before continuing, let's tidy up the code a little. Previously, to print the ID of the joke into the hidden input, it required a simpler piece of code:

```
<input type="hidden" name="jokeid" value="<?=$joke[id]?>">
```

Now that we're using the `if` statement, we need the full `<?php` opening and several lines of code.

A nice feature introduced in PHP 7 is the `??` operator, which acts as shorthand `if-isset` and can be used as shorthand for this code:

```
if (isset($something)) {
    echo $something;
}
else {
    echo 'variable not set';
}
```

This can be expressed using the `??` operator like so:

```
echo $something ?? 'variable not set';
```

On the left-hand side of the `??` operator is the variable being checked, and on the right is the output that's used if it's not set. In the case above, if the `$something` variable is set, it will print the contents of the variable. If the variable isn't set, it will print `variable not set`, and even better, it also works with arrays.

When teaching classes, I always try to get students to guess the name of this operator. Like my students, you'll be a long time guessing before you hit on its name. It's called the **Null coalescing operator**, which makes it sound a lot more complicated than it actually is.

Let's make use of it in our templates, rather than having to write out the entire `if` statement for each field:

```
<form action="" method="post">
    <input type="hidden" name="jokeid" value="<?=$joke['id'] ?? ''?>">
        <label for="joketext">Type your joke here:</label>
        <textarea id="joketext" name="joketext" rows="3" cols="40"><?=$joke
        ➥['joketext'] ?? ''?></textarea>
        <input type="submit" name="submit" value="Save">
</form>
```

In this instance, the right-hand part is an empty string. Either the text of the

loaded joke will be displayed in the text box, or if the `$joke` variable isn't set, it will display an empty string `''` in the box.

We'll need to use the `??` operator again to solve a different problem. In the code above, what happens if `$_GET['id']` is set to an ID that doesn't exist in the database? The following code will run and will attempt to read the first joke from the list of jokes with the specified ID. If there are no jokes in the list, it will print an error! To avoid that, we can once again use `?? null` so that `$joke` is set to `null` if there's no joke with the specified ID:

```
else {
    if (isset($_GET['id'])) {
        $joke = find($pdo, 'joke', 'id', $_GET['id'])[0] ?? null;
    }
    else {
      $joke = null;
    }

//...
```

To complete this page, we need to change what happens when the form is submitted. Either an `update` or `insert` query will need to be run.

The simplest way would be to look to see if the `ID` field has been supplied:

```
if (isset($_POST['id']) && $_POST['id'] != '') {
    update(...);
}
else ;
    insert(...)
}
```

Although this would work, once again there's an opportunity to make this more generic. This logic for *if the ID is set, update, otherwise insert* is going to be the same for any form.

If we had a form for *authors*, the same logic would be useful: *if there's no ID, perform an insert; if there is, perform an update*. Indeed, this would apply

anywhere we have a form that will be used for add or edit. The code above would need to be repeated for any form we implemented in this way.

Another problem with this approach is that it wouldn't work in cases where the primary key was a normal field on the form, such as ISBNs for books. Regardless of whether an *`UPDATE`* or *`INSERT`* query is required, the ISBN would be provided.

Instead, we can try to insert a record, and if it's unsuccessful, update instead using a *`try … catch`* statement:

```
try {
    insert(…);

}
catch(PDOException $e) {
    update(…);
}
```

Now an *`INSERT`* will be sent to the database, but it *may* cause an error—"Duplicate key"—when a record with the supplied ID already exists. If an error does occur, an *`UPDATE`* query is issued instead to update the existing record.

To stop us needing to repeat this logic for every form, we could make another function, called *`save`*, which performs an insert or an update using the *`try … catch`* above:

```
function save($pdo, $table, $primaryKey, $record) {
    try {
        if (empty($record[$primaryKey])) {
          unset($record[$primaryKey]);
        }
        insert($pdo, $table, $record);
    }
    catch (PDOException $e) {
        update($pdo, $table, $primaryKey, $record);
```

```
    }
}
```

This will work for any record in any table. If there's an error when trying to insert, it will issue the corresponding update query instead.

The `save` function here needs all the arguments required by both the `insert` and `update` functions in order to call them. There's no need to repeat any of the logic from those functions; they can just be called from within the relatively short `save` function.

Notice that I've unset the record's primary key if it's empty. This is because the `save` function may will take a full record from a form. It's possible that the primary key (usually `id`) has the value of an empty string (`''`). As we don't want to try to insert an empty string into the `id` column, it will need to be removed prior to inserting the record.

Performance Overheads

This approach relies on sending two queries to the database. This creates a small performance overhead when an update occurs. However, other approaches—such as using a `SELECT` query to determine if the record already exists—add a performance overhead 100% of the time.

Alternatively, MySQL/MariaDB supports `REPLACE INTO` and `INSERT INTO .. ON DUPLICATE KEY UPDATE`, which perform the same task (update or insert depending on whether a record with the specified ID exists). These can be slow in themselves, and are MySQL specific; they won't work if you use a different database engine in place of MySQL. The code above will work regardless of which database you're using.

Now `editjoke.php` can be modified to use the new `save` function:

```
try {
    if (isset($_POST['joketext'])) {

        save($pdo, 'joke', 'id', [
                    'id' => $_POST['jokeid'],
                    'joketext' => $_POST['joketext'],
                    'jokedate' => new DateTime(),
                    'authorId' => 1
                 ]
                );

        header('location: jokes.php');
    }
```

Finally, you can delete the controller `addjoke.php` and the template `addjoke.html.php`, as they're no longer needed. Both add and edit are now handled by `editjoke.php`. Change the link in the menu in `layout.html.php` to go to `editjoke.php` instead of `addjoke.php`.

This example can be found in **Example: Structure2-GenericFunction-Save**[7].

Further Polishing

The new `save` function can be used to add records to the database. It will automatically insert or update depending on whether the `id` field is set to an existing ID. For the add form, where the ID field isn't set, there's still an `id` key in the `$_POST` array, because the hidden input still exits on the form.

This allows us to treat adding and editing identically, saving us a lot of work. Without this generic `save` function, we'd need different HTML forms and different controller logic for processing the submissions.

Using this approach, there's less code to write initially, and the lack of HTML or PHP code repetition is an extra bonus. To add a new field, all we need to do is add the field to the database, amend the template and supply the new value to the `save` function.

---

7. https://github.com/spbooks/phpmysql7/tree/Structure2-GenericFunction-Save

Adding new fields is a very simple process. Because the `save` function handles both `INSERT` and `UPDATE` queries, and the `insert` and `update` functions both dynamically generate the queries, adding a field to the query is now as easy as adding an entry to an array!

This updated code is incredibly easy to manage compared to the code we started with at the beginning of the last chapter. However—and you're probably anticipating me saying this by now—it's always worth asking whether it can be simplified further. Once again, the answer is yes.

There's a little repetition here:

```
[
    'id' => $_POST['jokeid'],
    'authorId' => 1,
    'jokedate' => new DateTime(),
    'joketext' => $_POST['joketext']
];
```

Each field in the `$_POST` array is mapped to a key in the `$joke` array with the same name.

Take a look at the line `'joketext' => $_POST['joketext'],`. All we're really doing here is creating a key in the `$joke` array with the name `joketext` and the value from the `$_POST` array's `joketext` element.

Really, we're just copying data from the `$_POST` array exactly into the `$joke` array. If the form had more fields, we'd need to copy those too.

It would be possible to do something similar using this code:

```
    $joke = $_POST;
    $joke['authorId'] = 1;
    $joke['jokedate'] = new DateTime();

    save($pdo, 'joke', 'id', $joke);
```

This will automatically include all the fields from the form in the `$joke` array without needing to manually copy them. Once the form is submitted, `$_POST` is already an array with `joketext` and `id` keys with the corresponding values. By taking the existing `$_POST` array and adding the `jokedate` and `authorId` keys and corresponding values to it, we can avoid the unnecessary copying.

Now, if we added a new field to the form, there are only two steps: add the column in the database and add the form. As long as the column name in the database is the same as the field name on the form, it's possible to add new fields to the form without ever opening up the controller!

As neat as that sounds, if you do try the code above, you'll find you get an error. That's because the `$_POST` array also contains the submit button. When the `INSERT` query is generated, it would actually generate this query:

```
INSERT INTO `joke` (`joketext`, `jokedate`, `authorid`, `submit`)
```

This is obviously a problem: there's no submit column in the database table `joke`. A quick and dirty way of fixing this is to remove the submit button from the array using `unset`:

```
$joke = $_POST;

// Remove the submit element from the array
unset($joke['submit']);

$joke['authorId'] = 1;
$joke['jokedate'] = new DateTime();

save($pdo, 'joke', 'id', $joke);
```

Although this works, the problem with this approach is that you'd have to remove any form elements you don't want inserted into the database. For example, if you have a checkbox on the form that sends an email when checked, you'd also remove this checkbox from the `$_POST` array prior to calling the `save` function.

As usual, there's a better way. When using HTML forms, you can actually send an *array* as post data by changing the field names. Change `editjoke.html.php` to this:

```
<form action="" method="post">
    <input type="hidden" name="joke[id]" value="<?=$joke['id'] ?? ''?>">
        <label for="joketext">Type your joke here:</label>
        <textarea id="joketext" name="joke[joketext]" rows="3" cols="40">
        ↪<?=$joke['joketext'] ?? ''?></textarea>
        <input type="submit" name="submit" value="Save">
</form>
```

Each of the fields that represents some data for the joke has been changed slightly. The name attribute of each form field has been updated to represent an array: `jokeid` is now `joke[id]` and `joketext` is now `joke[joketext]`.

This tells PHP to treat these fields like an array when the form is submitted.

If you submit the form, the `$_POST` array will store two values: `submit` and `joke`. `$_POST['joke']` is itself an array from which you can read the `id` value using `$_POST['joke']['id']`.

For our purposes, we can read all of the information for the joke using `$joke = $_POST['joke']` and it won't contain the submit button entry, or any other array keys/values that we don't want to send to the `save` function:

```
$joke = $_POST['joke'];
$joke['authorId'] = 1;
$joke['jokedate'] = new DateTime();
```

The `$joke` array will contain all the values from `$_POST['joke']` and any we want to add that don't come from the form—in this case, `authorId` and `jokedate`. Following this approach, however, it's important that the field names on the form exactly match up to column names in the database, so I've used `joke[id]` instead of `joke[jokeid]`. The latter would try to write to a column in the database called `jokeid`, which doesn't exist.

Finally, you'll need to update the `if` statement that detects whether the form has been submitted to look for the new `joke` key, rather than `joketext`, which no longer exits:

```
if (isset($_POST['joke'])) {
```

The complete controller code is now as shown below.

Example: Structure2-GenericFunctions-SaveArray[8]

```
<?php
include __DIR__ . '/../includes/DatabaseConnection.php';
include __DIR__ . '/../includes/DatabaseFunctions.php';

try {
        if (isset($_POST['joke'])) {
                $joke = $_POST['joke'];
                $joke['jokedate'] = new DateTime();
                $joke['authorId'] = 1;

                save($pdo, 'joke', 'id', $joke);

                header('location: jokes.php');
        } else {
                if (isset($_GET['id'])) {
                    $joke = find($pdo, 'joke', 'id', $_GET['id'])[0] ?? null;
                }
                else {
                  $joke = null;
                }
                $title = 'Edit joke';

                ob_start();

                include  __DIR__ . '/../templates/editjoke.html.php';

                $output = ob_get_clean();
        }
```

8. https://github.com/spbooks/phpmysql7/tree/Structure2-GenericFunctions-SaveArray"

```
} catch (PDOException $e) {
        $title = 'An error has occurred';

        $output = 'Database error: ' . $e->getMessage() . ' in ' .
        $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

If we wanted add a field to the `joke` table and alter the form now, it would only require two changes: adding the field to the database and then editing the HTML form. A single update to `editjoke.html.php` will let us add a form field that works for both the edit and add pages.

Moving Forward

In this chapter, I've shown you how to reduce repeated code and write functions for use with any database table. We've moved from some very specific functions to functions that can be used in several different situations.

You now have a set of tools you can use to extend this website or even write a completely different one. None of the functions are tied to concepts like *jokes* or *authors*, which means we could use them on a website that dealt with entirely different concepts—for example, *books*, *products*, *blogs*, or anything you can think of. Now that you've written the tools, the hard part is done, and in your next project you can save a lot of time by reusing the tools you've created.

In the next chapter, we'll continue where we left off here, and I'll show you how we can use objects and classes to reduce some of the remaining repetition in our code.

Chapter

8

Objects and Classes

In the last chapter, I showed you how to write generic, reusable functions that could be used to manipulate any database table. In this chapter, we'll move those functions into a class, to avoid some of the repetition that's needed when they're used.

One of the biggest problems with functions is that all the information they need to execute has to be sent to them in arguments. In the case of the `delete` function we wrote, there are four pieces of information:

- the `$pdo` database instance
- the name of the table to delete from
- the name of the primary key field
- the value to delete

The same is true of all the functions— `find` , `findAll` , `update` , `insert` and `save` . Each of the functions we created needs to be passed at least the `$pdo` database instance and the name of the table. All of them except `findAll` and `insert` also need to know the name of the column that represents the primary key.

For example, the `save` function is used like this:

```
if (isset($_POST['joke'])) {

  $joke = $_POST['joke'];
  $joke['jokedate'] = new DateTime();
  $joke['authorId'] = 1;

  save($pdo, 'joke', 'id', $joke);
  // …
```

Each time one of the functions is called, it must be passed the `$pdo` instance. With up to four arguments for each function, it can be difficult to remember the order they need to be provided in.

A good method for avoiding this problem is putting the functions inside a container called a "class".

Time for Class

You can think of a **class** as a collection of functions and data (variables). Each class will contain a set of functions and some data that the functions can access.

As each class needs a name, and ours will deal with functions that have something to do with database tables, we'll call ours `DatabaseTable`.

Like variables, class names can contain any sequence of alphanumeric characters. However, special characters like `-`, `+`, `{` or a space aren't allowed.

By convention, classes in PHP use **CamelCase**, starting with an uppercase letter followed by lowercase letters until the start of the next word, which is signified by another uppercase letter. PHP allows for the class to be called `databasetable`, `DATABASETABLE`, or some other similar variation, but it's a good idea to follow the naming convention used by almost all PHP programmers.

Our `DatabaseTable` class needs to contain all the functions we created for interacting with the database, along with any functions that those functions need to call.

As a first step, move all the database functions into a class wrapper:

```
<?php
class DatabaseTable {

    public function total($pdo, $table) {
        $stmt = $pdo->prepare('SELECT COUNT(*) FROM `' . $table . '`');
        $stmt->execute();
        $row = $stmt->fetch();
        return $row[0];
    }

    public function find($pdo, $table, $field, $value) {
```

```
        $query = 'SELECT * FROM `' . $table . '` WHERE `' . $field . '` =
        ➥:value';

        $values = [
            'value' => $value
        ];

        $stmt = $pdo->prepare($query);
        $stmt->execute($values);

        return $stmt->fetchAll();
    }

    private function insert($pdo, $table, $values) {
        $query = 'INSERT INTO `' . $table . '` (';

        foreach ($values as $key => $value) {
            $query .= '`' . $key . '`,';
        }

        $query = rtrim($query, ',');

        $query .= ') VALUES (';

        foreach ($values as $key => $value) {
            $query .= ':' . $key . ',';
        }

        $query = rtrim($query, ',');

        $query .= ')';

        $values = $this->processDates($values);

        $stmt = $pdo->prepare($query);
        $stmt->execute($values);
    }

    private function update($pdo, $table, $primaryKey, $values) {
        $query = ' UPDATE `' . $table .'` SET ';

        foreach ($values as $key => $value) {
            $query .= '`' . $key . '` = :' . $key . ',';
```

```
    }

    $query = rtrim($query, ',');

    $query .= ' WHERE `' . $primaryKey . '` = :primaryKey';

    // Set the :primaryKey variable
    $values['primaryKey'] = $values['id'];

    $values = $this->processDates($values);

    $stmt = $pdo->prepare($query);
    $stmt->execute($values);
}

public function delete($pdo, $table, $field, $value) {
    $values = [':value' => $value];

    $stmt = $pdo->prepare('DELETE FROM `' . $table . '` WHERE `' .
    ↪$field . '` = :value');

    $stmt->execute($values);
}

public function findAll($pdo, $table) {
    $stmt = $pdo->prepare('SELECT * FROM `' . $table . '`');
    $stmt->execute();

    return $stmt->fetchAll();
}

private function processDates($values) {
    foreach ($values as $key => $value) {
        if ($value instanceof DateTime) {
            $values[$key] = $value->format('Y-m-d');
        }
    }

    return $values;
}

public function save($pdo, $table, $primaryKey, $record) {
    try {
```

```
            if (empty($record[$primaryKey])) {
              unset($record[$primaryKey]);
            }
            $this->insert($pdo, $table, $record);
        } catch (PDOException $e) {
            $this->update($pdo, $table, $primaryKey, $record);
        }
    }
}
```

Before we dig into the bits of the code that look unfamiliar here, save the file. Like templates and include files, it's good practice to store classes outside the `public` directory. Create a new directory called `classes` inside your `default` directory and save the code above as `DatabaseTable.php`.

Naming Your Class Files

It's good practice to name your class files exactly the same as your classes. So the class `DatabaseTable` should be placed in `DatabaseTable.php`, and a class called `User` should be stored in `User.php`, and so on. Although it doesn't matter at the moment, later on I'll introduce something called an *autoloader*, and it will be difficult to use without this convention.

If you examine the code above, you'll see that I've made two changes beyond just pasting the functions into the class. The first change I've made is that, when functions from within the class are called, they're prefixed with `$this->`. Instead of `update($pdo,…`, the updated code has `$this->update($pdo,…`.

Methods vs Functions

A function that exists inside a class is called a **method**. Although many developers—and the PHP language itself—use the word "function" to describe subroutines in classes, the correct term is *method*, which I'll be using as we continue. However, the difference between a *function* and a *method* is that a *method* is inside a class, while a *function* isn't.

Anything you can do with a *function* (arguments, return values, calling other functions) you can also do with a *method*. The inverse, however, isn't true: a method can access *class variables* through the `$this` keyword. If you try to use the `$this` keyword in a *function*, you'll get an error!

You can think of `$this` as "this class". We can't just use `processDates()` anymore, because, now that the `processDates()` function is inside the class, it can't be called like a *global* function, as it's outside of global scope. Any method inside a class can only be called on a *variable*. In the same way we use `$pdo->prepare()`, the `processDates()` method is now called on an object. From within the class, the current object is referenced as `$this`. The `$this` variable is created automatically inside any method and will always exist without being declared.

Public vs Private

The second change I've made when converting functions to methods is that each is prefixed with either `public` or `private`. The `private` and `public` modifiers determine the **visibility** of the method. That is, they allow the programmer to determine where the method can be called from.

Methods marked `private` can only be called from other methods inside the class, whereas methods marked `public` can be called from both inside and outside the class. You've already seen some public methods on the PDO instance we've been using throughout this book. When you call `$pdo-`

`>prepare()`, you're calling a *public* method called `prepare`. If the method were marked private, this wouldn't be possible.

So, what's the point of private methods? Take a look at the methods I've marked as `private` in the `DatabaseTable` class: `insert`, `update` and `processDates`. The reason these are `private` is that, on their own, they aren't very useful. Nobody using the `DatabaseTable` class should ever need to call the `processDates` method directly. The `processDates` method is only there to provide some shared functionality for the other methods in the class—`save`, `find`, and so on.

I've also set the `insert` and `update` methods to `private`. This is because any script using the code will just call the `save` method, which will then determine whether it's a new or existing record.

At first glance, it seems a little pointless. However, it's actually a very useful tool. Once a method such as `update` is private, you can completely rewrite the way it works, and you can guarantee the only place it's being called from is another method within the same class.

When you're working in large teams or sharing your code online, knowing exactly where a method is called from is useful. You can release a new version of the class without the `insert` method, and you can be guaranteed it's not being used anywhere else. Someone else can use the new version and their code won't be broken.

If it were public and you changed the code, you'd have no idea if the `insert` function was being called directly from anywhere else, and you'd need to be wary of changing the way it worked in case it broke someone else's code.

Protected

You may occasionally also come across the `protected` visibility modifier. This is a kind of middle ground between `private` and `public`. If you're used to Java, PHP works differently. In PHP, anything marked as `protected` can be

accessed by subclasses and the class it was declared in.

Because creating subclasses through inheritance is almost always a poor design choice, you won't find any examples of it in this book.

Creating Subclasses through Inheritance

To learn more about why creating subclasses through inheritance is a poor design choice, here are some resources you can read:

- "One pattern to rule them all"[1]
- "Inheritance vs Composition: Composition wins by knockout"[2]
- "Inheritance Is a Procedural Technique for Code Reuse"[3]
- "Why extends is evil"[4]

Professional developers follow a programming technique called "the principle of least knowledge", which dictates that methods and variables should only be exposed to exactly what needs them. In essence, this means that you shouldn't make something `public` if you can make it `private`. As the author of the class, once you make something `public` or `protected` you no longer know where the variable or method is being used, so you can't remove it when you want to redesign the class.

It's generally a bad idea to have any variables that are `public` or `protected` (there are some exceptions!), but you'll quite often need `public` *methods*.

Objects

You can think of a class as a recipe. On its own, it's just a series of instructions. To make something from it that's actually useful—something you can eat—you need to follow the instructions.

---

1. https://blog.twitter.com/engineering/en_us/topics/insights/2019/onepattern
2. https://r.je/you-do-not-need-inheritance-oop
3. https://www.yegor256.com/2016/09/13/inheritance-is-procedural.html
4. https://www.infoworld.com/article/2073649/why-extends-is-evil.html

A class on its own isn't very useful; it's just a series of instructions. There's no way to call a method from within the class without creating an **object**. An object is an *instance* of a class.

Creating an instance of the `DatabaseTable` class is done the same way as the `pdo` instance we've been using throughout this book, only the name of our class, `DatabaseTable`, is used:

```
$databaseTable = new DatabaseTable();
```

The `new` keyword creates an *object* from the defined class, which can then be used. Without this step, none of the functions defined in the class can be used.

At the moment, this feels like an extra step that doesn't achieve anything special. But as we'll see later on, this is a very powerful tool for programmers, allowing us to create different instances that represent different database tables.

Once the object is created, the methods can be called on the variable in the same way we call `$pdo->prepare()` on the `$pdo` object:

```
$databaseTable = new DatabaseTable();
$jokes = $databaseTable->findAll($pdo, 'joke');
```

Any of the `public` methods can be called in this way. However, if you try to call one of the `private` methods, you'll get an error.

Class Variables

At the start of this chapter, I mentioned that the goal of using objects and classes was to reduce repeated code. However, so far we've actually made the code longer. With the `DatabaseTable` class, each time we want to use one of the methods, we need to call it on an object:

```
$databaseTable = new DatabaseTable();

$jokes = $databaseTable->findAll($pdo, 'joke');

$databaseTable->save($pdo, 'joke', 'id', $_POST['joke']);
```

In this case, we've increased rather than reduced the amount of code that's required.

Each time one of the methods in the class is called, it needs the same information—at a minimum, the database connection and the name of the table that's being interacted with.

Rather than supply these values every time a method is called, it's possible to supply them once, to the class, and have the values used within the methods.

Every class can have variables that are available to be used within any method. To declare a variable that will be used inside the class, you need to declare the variable *within* the class. It's convention to define variables at the top of the class, before any methods.

To declare a variable, you must make it visible and give it a name. For example:

```
class Cake {
  public $flavor;
}
```

Once the variable has been declared, you can use it when you create an instance of the class. Like methods, variables can be written to, and read from, using the arrow (`->`) operator:

```
$myInstance = new Cake();
$myInstance->flavor = 'Chocolate';

echo $myInstance->flavor; // prints "Chocolate"
```

An important distinction between **class variables** and normal variables is that

they're bound to a specific *instance*. In practice, all this means is that each instance can have a different value for the same variable. For example:

```
$cake1 = new Cake();
$cake1->flavor = 'Chocolate';

$cake2 = new Cake();
$cake2->flavor = 'Fruit';

echo $cake1->flavor;
echo $cake2->flavor;
```

This will print "Chocolate" and then "Fruit", because each *instance* of the class has its own value for the `flavor` variable. Later on, this will be very useful for our `DatabaseTable` class, but for now, let's just add the variables for the `$pdo` instance, the table name and the primary key to the class:

```
class DatabaseTable {
  public $pdo;
  public $table;
  public $primaryKey;

  // …
}
```

A class is more than just a collection of functions. You can think of it as a *blueprint* that can be used to create objects. Each object or *instance* of the class can store its own values for these variables.

For example, when you create the `$pdo` variable for your database connection, the `$pdo` variable stores the connection information—the database server address, username, password, and so on. You don't need to send this information every time you call `prepare` or `execute`; the information is stored inside the `$pdo` object.

The same can be done with the `DatabaseTable` class. Once the variables have been declared, they can be written to on each instance:

```
$databaseTable = new DatabaseTable();
$databaseTable->pdo = $pdo;
$databaseTable->table = 'joke';
$databaseTable->prmaryKey = 'id';
```

Now that the variables are set, they can be used instead of the arguments inside any of the methods in the class. For example, the `findAll` method can be rewritten to use the class variables, instead of having the database connection and table name passed in explicitly:

```
public function findAll() {
    $stmt = $this->pdo->prepare('SELECT * FROM `' . $this->table . '`');
    $stmt->execute();

    return $result->fetchAll();
}
```

The variables in the class are accessed the same way as the functions using the `$this` variable. Now, when the `findAll()` function is called, it doesn't need any arguments, because the `$pdo` connection and the name of the table are read from the class variables:

```
$jokesTable = new DatabaseTable();
$jokesTable->pdo = $pdo;
$jokesTable->table = 'joke';

$jokes = $databaseTable->findAll();
```

Let's go ahead and make this change to all the methods. Anywhere `$pdo`, `$table` or `$primaryKey` was used as an argument, the argument can be removed and replaced with a reference to the class variable.

Here's what the `total` method looks like now:

```
public function total() {
    $stmt = $this->pdo->prepare('SELECT COUNT(*) FROM `' . $this->table . '`');
    $stmt->execute();
```

```
    $row = $stmt->fetch();
    return $row[0];
}
```

And the `find` method:

```
public function find($field, $value) {
    $query = 'SELECT * FROM `' . $this->table . '` WHERE `' . $field . '` =
    ↪:value';

    $values = [
        'value' => $value
    ];

    $stmt = $this->pdo->prepare($query);
    $stmt->execute($values);

    return $stmt->fetchAll();
}
```

Notice that, in this function, there's still an argument for `$value`. The `$pdo`, `$table` and `$primaryKey` variables will be the same every time this method is called, but the `$value` will be different and therefore stays as an argument.

Once this is implemented, the class can be used like this:

```
$jokesTable = new DatabaseTable();
$jokesTable->pdo = $pdo;
$jokesTable->table = 'joke';
$jokesTable->primaryKey = 'id';

$joke1 = $jokesTable->find('id', 1)[0];

$joke2 = $jokesTable->find('id', 2)[0];
```

By using the class variables, the database connection (`$pdo`) and table name (`$table`) don't need to be provided every time the `find` method is called. They can be set *once* as class variables, and each time the `find` method is called, it will use them.

Here's the `save` method:

```
public function save($record) {
    try {
       if (empty($record[$this->primaryKey]) {
         unset($record[$this->primaryKey]);
       }
       $this->insert($record);
    } catch (PDOException $e) {
       $this->update($record);
    }
}
```

Here's the `update` method:

```
private function update($values) {
    $query = ' UPDATE `' . $this->table .'` SET ';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '` = :' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ' WHERE `' . $this->primaryKey . '` = :primaryKey';

    // Set the :primaryKey variable
    $values['primaryKey'] = $values['id'];

    $values = $this->processDates($values);

    $stmt = $this->pdo->prepare($query);
    $stmt->execute($values);
}
```

Like the `find` method, the `$values` that will be updated in the database will be different each time the method is called, but the table name, primary key and database connection will be the same, and will use the class variables. The `insert` and `delete` methods can be implemented in the same way:

```
private function insert($values) {
    $query = 'INSERT INTO `' . $this->table . '` (';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '`,';
    }

    $query = rtrim($query, ',');

    $query .= ') VALUES (';

    foreach ($values as $key => $value) {
        $query .= ':' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ')';

    $values = $this->processDates($values);

    $stmt = $this->pdo->prepare($query);
    $stmt->execute($values);
}

public function delete($field, $value) {
    $values = [':value' => $value];

    $stmt = $this->pdo->prepare('DELETE FROM `' . $this->table . '` WHERE `' .
    ↪$field . '` = :value');

    $stmt->execute($values);
}
```

The `processDates` method remains unchanged, as it doesn't require any of the class variables.

Now, to interact with the database, the common variables only need to be set once:

```
$jokesTable = new DatabaseTable();
```

```
$jokesTable->pdo = $pdo;
$jokesTable->table = 'joke';
$jokesTable->primaryKey = 'id';
```

And then the methods can be used without repeating all the arguments:

```
// Find the joke with the ID `123`
$joke123 = $jokesTable->find('id', 123)[0];

// Find All the jokes
$jokes = $jokesTable->findAll();

foreach ($jokes as $joke) {
  // …
}

// Delete the joke with the ID `33`
$jokesTable->delete('id', 33);

$newJoke = [
  'authorId' => 1,
  'jokedate' => new DateTime(),
  'joketext' => 'A man threw some cheese and milk at me. How dairy!'
];

$jokesTable->save($newJoke);
```

This reduces the number of arguments needed by each method, and makes it easier for someone using the methods to follow. They don't have to remember the order of all the arguments—for example, whether the table name is the first or second argument.

You might be wondering what would happen if we wanted to query the authors table. Each table can be represented by its own instance, and either table could be queried by calling the method on the relevant instance:

```
$jokesTable = new DatabaseTable();
$jokesTable->pdo = $pdo;
$jokesTable->table = 'joke';
```

```
$jokesTable->primaryKey = 'id';

$authorsTable = new DatabaseTable();
$authorsTable->pdo = $pdo;
$authorsTable->table = 'author';
$authorsTable->primaryKey = 'id';

// Find the joke with the ID `123`
$joke123 = $jokesTable->find('id', 123)[0];

// Find the author with the ID `333`
$joke123 = $authorsTable->find('id', 333)[0];
```

Each table can be represented as its own instance, and more instances can be added as new tables are added. We'll come back to this concept a little later, but for now it's good to keep in mind that class variables should contain things that are going to stay the same each time a method on a particular instance is run (such as a database connection), and arguments are used for values that will differ each time the method is called (like the ID of the record being queried, as shown above).

Moving the repeated variables into class variables is a huge improvement, but there are some potential problems. What happens if the variables aren't set before the `findAll()` or another method is called? What happens if the `$pdo` variable is set to a string rather than an object?

```
$jokesTable = new DatabaseTable();
$jokes = $databaseTable->findAll();
```

You'll get an error if you run this, because the `findAll` method is expecting the `$pdo` and `table` variables to be set to valid values. Luckily, there's a way to prevent this from happening.

Constructors

As the author of a class, you get to tell anyone who uses it how it works. (If you want to get technical, this is called the **application programming interface**, or **API**). You can make sure that any required variables are set correctly before

any functions are run.

There's a special method you can add to the class, called a **constructor**. This is a method that's automatically run whenever an instance of the class is created. To add a constructor to a class, you simply add a method called `__construct()`.

Magic Methods

That's *two* underscores in front of the word `construct`. If you use just one, it won't work!

In PHP, any method prefixed by two underscores is a **magic method**. These are generally called automatically in different cases. As the language evolves, more of these magic methods may be added, so it's a good idea to avoid giving your own methods names beginning with two underscores.

A complete list of the available magic methods can be found in the PHP manual[5].

The `__construct` method is declared like any other, but it's called automatically. For example:

```
class MyClass {
  public function __construct() {
    echo 'construct called';
  }
}

$myclass1 = new MyClass();
$myclass2 = new MyClass();
```

Once a method with the name `__construct()` is created, the method is called each time you create a new instance of the class. The code above will output the following:

---

5. http://php.net/manual/en/language.oop5.magic.php

```
construct called
construct called
```

Even though we've never directly called the method using `$myclass1->__construct()`, you can see it has been called because the string `construct called` has been printed.

You may also have noticed that it's been called *twice*. This is because, each time an *instance* of the class is created, the constructor is called automatically by PHP.

Like any other function, the constructor can also take *arguments*. For example:

```
class MyClass {
  public function __construct($argument1) {
    echo $argument1;
  }
}
```

When you create an instance of the class, the arguments can be provided:

```
$myclass1 = new MyClass('one');
$myclass2 = new MyClass('two');
```

Given the class `MyClass` and the code instantiating it as above, this will print `one`, then `two`, as the constructor will be called twice, each time with a different value for `$argument1`.

If you try to create an instance of the class that needs a constructor argument and doesn't have a default defined, but you fail to pass in an argument, you'll see an error. For example, `$myclass3 = new MyClass()` will cause an error, because no value has been provided for `$argument`.

Let's add a constructor to the `DatabaseTable` class that sets the `$pdo`, `$table` and `$primaryKey` variables:

```
class DatabaseTable {
  public $pdo;
  public $table;
  public $primaryKey;

  public function __construct($pdo, $table, $primaryKey) {
    $this->pdo = $pdo;
    $this->table = $table;
    $this->primaryKey = $primaryKey;
  }

  // …
}
```

You'll come across constructors like this frequently if you start using objects and classes regularly, so it's important to understand what's happening here.

Placing Your Constructor Methods

It's common practice to put the constructor at the top of the class, after variables but before any other methods.

There are two different variables with the same names, which can be confusing at first. The first version is the *argument*, which is defined in the line `public function __construct($pdo, $table, $primaryKEy) {`. When you create a function argument, the variable is only available inside that specific function and isn't available to be used in other functions in the class.

When the constructor is called, the `$pdo` instance is sent to it, but we want to make the `$pdo` instance available to *every* function in the class. The only way to make a variable available to every function inside the class is to make it a class variable like we did above. Instead of setting the class variable from outside the class—for example, `$jokesTable->pdo = $pdo;`—we want to set it from within the constructor inside the class.

Like before, the `$this` variable represents the current *instance*, and `$this->pdo = $pdo;` is doing the same thing as `$jokesTable->pdo = $pdo;`, only from

inside the class. Both `$jokesTable` and `$this` reference the same *object*, and making changes to one will be reflected in the other.

You can think of this like the English language. Although you'll always refer to yourself as "I", your friends will use your name. Regardless of whether you're referring to yourself, or someone is referring to you by your name, it's always the same person—you—who's being referred to.

The same thing happens here. `$this` references the current instance from inside the class, like "I" in English. However, `$jokesTable` refers to the same instance using its name from outside the class.

Using either `$jokesTable->pdo = $pdo` from outside the class, or `$this->pdo = $pdo;` from inside the class, the `$pdo` class variable will be set and then be available inside any methods when they're called on that instance.

By using a constructor, when the instance is created, two variables *must* be supplied:

```
$jokesTable = new DatabaseTable($pdo, 'jokes', 'id');
```

If you tried to create an instance of the `DatabaseTable` class without passing it two arguments, it would give you an error, because the two arguments are required for the code to work.

This kind of check ensures the code is robust. It also helps anyone who uses the class, because they'll see an error as soon as they do something wrong.

Type Hinting

If we're trying to make the class foolproof, there's still a problem. What happens if the person using your `DatabaseTable` class gets the order of the arguments wrong? Consider these two examples:

```
$jokesTable = new DatabaseTable('jokes', $pdo, 'id');
```

```
$jokesTable = new DatabaseTable($pdo, 'jokes', 'id');
```

The first could easily be written instead of the second. This is an innocent mistake, and an easy one to make by accident. An error won't be seen until one of the functions in the class is called. For example:

```
$jokesTable = new DatabaseTable('jokes', $pdo, 'id');
$jokes = $jokes->findAll();
```

The code above will result in the error "Call to function prepare on non-object", because the `findAll` function contains the line `$result = $this->pdo->prepare('SELECT * FROM ' . $this->table);` .

Because the order of the constructor arguments is wrong, the `$pdo` variable will actually be set to the string `joke` . The string `joke` doesn't have a function called `prepare` , which is what causes the error.

The error "Call to function prepare on non-object" doesn't make it clear what went wrong, and it would be difficult for the person who made the mistake to figure it out without looking in depth at your class and examining it line by line.

To help them out, it's better to ensure that the arguments are the correct **type**. PHP is **loosely typed**, meaning that a variable can be any type—such as a string, a number, an array, or an object. Even so, you can enforce types when you create a function. This is particularly useful for constructors, where getting the arguments in the wrong order appears to work. For example, take the following code:

```
$jokesTable = new DatabaseTable('jokes', $pdo, 'id');
```

This won't actually cause any errors. The person running this line of code won't know that it's wrong. It's possible to use `if` statements to check the type of each argument, but PHP also provides a nice feature called "type

hinting" to address this issue with less code.

Type hinting allows you to specify the type of an argument. The type can be a class name, or one of the basic types, such as string, array or integer.

To provide a type hint for an argument, prefix the variable name with the type that the variable should be. For our database class this will be:

```
public function __construct(PDO $pdo, string $table, string $primaryKey) {
```

This tells PHP to check the types of each argument when they're provided. If the object is constructed with the wrong types now—as in `$jokesTable = new DatabaseTable('jokes', $pdo, 'id');`, for example—PHP will check to see whether the *type* of each argument matches the *hint*. If it doesn't, it will produce an error. The error that's printed in this case is this:

```
Uncaught TypeError: Argument 1 passed to DatabaseTable::__construct() must be an
  ↪ instance of PDO, string given
```

This error explains much more clearly what the problem is than "Call to function prepare on non-object", and it prevents the rest of the script from even running. As soon as a mistake is detected, the script is halted so you can fix it. This is a lot better than only getting a vague error message at the point you're trying to call one of the methods on the object!

By using type hinting on constructors like this, you can ensure that the class variables are set to the types you're expecting. This way, when the code `$this->pdo->prepare` is run inside one of the methods in the class, `$this->pdo` *must* be set to an instance of `$pdo` and have a `prepare` method. There's no way for the `$this->pdo` variable to be set to a string, a number, or even not set to anything.

This is known as **defensive programming**, and it's a very useful way of preventing bugs. By stopping variables being set to the wrong type, you can rule out the possibility of many potential bugs.

Private Variables

The class variables and constructor in the `DatabaseTable` class now look like this:

```
class DatabaseTable {
  public $pdo;
  public $table;
  public $primaryKey;

  public function __construct(PDO $pdo, string $table, string $primaryKey) {
    $this->pdo = $pdo;
    $this->table = $table;
    $this->primaryKey = $primaryKey;
  }

  // …
}
```

When an instance of the class is created, it *must* be passed three arguments, and those three arguments must be of specific types (a `$pdo` instance, a string, and a string).

It's now impossible to construct the class without providing the correct parameters:

```
$jokesTable = new DatabaseTable($pdo, 'jokes', 'id');
```

Any other combination—such as `new DatabaseTable($pdo, 'jokes');`, or `new DatabaseTable('jokes', $pdo, 'id');`, or `new DatabaseTable();`—will display an error. Once one of the methods is called (for example, `$jokesTable->findAll();`), all of the class variables must have been set to the correct type, which should stop the `$pdo` variable being set to anything but a real database connection, a PDO instance.

However, the code still has a weak point. There's still a way of making it so the `$pdo` variable in the class isn't a PDO instance.

That's because the variable `$pdo` is *public*. Like public functions, this means that the variables are accessible from outside the class, and it means the following code is possible:

```
// Correctly create the instance with a database connection
$jokesTable = new DatabaseTable($pdo, 'jokes', 'id');

$jokesTable->pdo = 'a string';

$jokes = $jokesTable->findAll();
```

Although the constructor is ensuring that a valid database connection is being set when the object is created, the code above has overwritten the `$pdo` class variable between the constructor being executed and the `findAll` method being called. The `$pdo` variable in the `$jokesTable` object has been set to "a string". When the `findAll()` method runs, `$this->pdo->prepare()` will throw an error, because `$this->pdo` is a string, not an object with a `prepare` method.

Public class variables like these cause problems, because they allow the variable to be overwritten from anywhere. Instead, it's good practice to make class variables *private* to prevent these issues:

```
class DatabaseTable {
  private $pdo;
  private $table;
  private $primaryKey;

  public function __construct(PDO $pdo, string $table, string $primaryKey) {
    $this->pdo = $pdo;
    $this->table = $table;
    $this->primaryKey = $primaryKey;
  }

  // …
}
```

When the variables are private, like private functions, they can't be accessed from outside the class (for either reading or writing).

By combining type hints, constructors and private properties, several conditions have been imposed on the class:

- it's impossible to create an instance of the `DatabaseTable` class without passing it a `$pdo` instance
- the first argument must be a valid PDO instance
- there's no way to change the `$pdo` variable from outside the class after it's been set

As a result of these conditions, when any of the functions are called (such as `findAll()` or `save()`), the `$pdo`, `$table` and `$primaryKey` variables must be set, and must be of the correct type. When `$this->pdo->prepare()` is called, it won't cause an error, because there's no way that `findAll()` can be called unless the variables are correctly set.

This type of defensive programming can take a little more thinking about (for example, what needs to be public and what needs to be private?), but it's worth it in all but the most simple projects! By eliminating the conditions for a bug to exist, you can save yourself a lot of bug-tracking time later on.

Constructor Property Promotion

Take a look at your code with the `private` properties:

```
class DatabaseTable {
    private $pdo;
    private $table;
    private $primaryKey;

    public function __construct(PDO $pdo, string $table, string $primaryKey) {
        $this->pdo = $pdo;
        $this->table = $table;
        $this->primaryKey = $primaryKey;
    }
```

There's a lot of repetition in this small block of code. Each variable name is repeated four times!

Until PHP 8, this approach was the only way to achieve the benefits listed above. However, PHP now supports a new feature aimed at removing all this repetition: **constructor property promotion**.

Rather than declaring the class variables, then declaring the arguments, then setting the class variables to the arguments, it's now possible to do all of this at once.

The code below is equivalent to the code above:

```
class DatabaseTable {
    public function __construct(private PDO $pdo, private string $table, private
    ➥ string $primaryKey) {
    }
```

A lot of the code has been removed. The variables aren't declared in the class, and all the lines beginning with `$this->` have also disappeared.

When using this new PHP feature, you declare the class variable as an argument by prefixing it with a visibility. The argument defined as `private PDO $pdo` tells PHP to create a class variable from the argument with this name and private visibility.

Functionally, it's identical to manually assigning class variables to the corresponding arguments, but this new feature saves a lot of typing!

You might be wondering how to create a constructor argument that isn't also a class variable. To do that, you simply create the argument without the `private` or `public` visibility prefix and the argument is only available in the constructor, not promoted to a class variable.

Using the `DatabaseTable` Class

The final version of the `DatabaseTable` class looks like this:

```
<?php
class DatabaseTable {
    public function __construct(private PDO $pdo, private string $table, private
    ↪ string $primaryKey) {
    }

    public function find($field, $value) {
        $query = 'SELECT * FROM `' . $this->table . '` WHERE `' . $field . '`
        ↪= :value';

        $values = [
            'value' => $value
        ];

        $stmt = $this->pdo->prepare($query);
        $stmt->execute($values);

        return $stmt->fetchAll();
    }

    public function findAll() {
        $stmt = $this->pdo->prepare('SELECT * FROM `' . $this->table . '`');
        $stmt->execute();

        return $result->fetchAll();
    }

    public function total() {
        $stmt = $this->pdo->prepare('SELECT COUNT(*) FROM `' .
        ↪$this->table . '`');
        $stmt->execute();
        $row = $stmt->fetch();
        return $row[0];
    }

    public function save($record) {
        try {
           if (empty($record[$this->primaryKey]) {
             unset($record[$this->primaryKey]);
           }
           $this->insert($record);
        } catch (PDOException $e) {
           $this->update($record);
```

```
        }
    }

    private function update($values) {
        $query = ' UPDATE `' . $this->table .'` SET ';

        foreach ($values as $key => $value) {
            $query .= '`' . $key . '` = :' . $key . ',';
        }

        $query = rtrim($query, ',');

        $query .= ' WHERE `' . $this->primaryKey . '` = :primaryKey';

        // Set the :primaryKey variable
        $values['primaryKey'] = $values['id'];

        $values = $this->processDates($values);

        $stmt = $this->pdo->prepare($query);
        $stmt->execute($values);
    }

    private function insert($values) {
        $query = 'INSERT INTO `' . $this->table . '` (';

        foreach ($values as $key => $value) {
            $query .= '`' . $key . '`,';
        }

        $query = rtrim($query, ',');

        $query .= ') VALUES (';

        foreach ($values as $key => $value) {
            $query .= ':' . $key . ',';
        }

        $query = rtrim($query, ',');

        $query .= ')';

        $values = $this->processDates($values);
```

```
        $stmt = $this->pdo->prepare($query);
        $stmt->execute($values);
    }

    public function delete($field, $value) {
        $values = [':value' => $value];

        $stmt = $this->pdo->prepare('DELETE FROM `' . $this->table . '` WHERE `'
        ➥. $field . '` = :value');

        $stmt->execute($values);
    }

    private function processDates($values) {
        foreach ($values as $key => $value) {
            if ($value instanceof DateTime) {
                $values[$key] = $value->format('Y-m-d');
            }
        }

        return $values;
    }

}
```

Let's save this in its own file, `DatabaseTable.php`. Remember to put the `<?php` tag at the top of the file.

Omitting the Closing PHP Tag from Your Files

Whenever you create a PHP file, you need to remember to put the PHP code inside PHP tags. However, the closing tag is optional, and it's actually better to omit it if the file only contains PHP code.

This is because, if there are any whitespace characters (blank lines, tabs or spaces) at the end of the file after the closing PHP tag `?>`, they'll be sent to the browser, which isn't what you want to happen. Instead, it's better to prevent this from happening by omitting the `?>` tag entirely. By leaving out the closing PHP tag, the whitespace will be interpreted on the server by PHP, and ignored, rather than being sent as part of the HTML code to the browser.

As briefly noted earlier, one of the most useful features of using classes is that, once a class has been written, it can be used as many times as you like. And each time you use it, by creating an *instance*, that instance can store different values of the class variables. For example, it's possible to use the `DatabaseTable` class to interact with the `joke` table and the `author` table.

Because each *instance* has its own version of the variables, we can have one version of the class where `$table` is set to `joke` and one version where `$table` is set to `author`:

```
$jokesTable = new DatabaseTable($pdo, 'joke', 'id');
$authorsTable = new DatabaseTable($pdo, 'author', 'id');

// Find the joke with the ID 123
$joke = $jokesTable->find('id', 123)[0];

// Find the author with the ID 34
$author = $authorsTable->find('id', 34)[0];
```

Because the class variable `$table` is different, a different table will be used for each of the instances.

When `$authorsTable->find('id', 34)[0]` is called, `$this->table` is equal to

`author`, so the query that runs will be `SELECT * FROM author …`, whereas when `$joke = $jokesTable->find('id', 123)[0];` is called, `$this->table` is set to `joke`, so the query that runs is `SELET * FROM joke …`.

This means the `DatabaseTable` class can now be used to insert, update or find records from any table in the database by constructing an instance with the table name in the constructor!

Updating the Controller to Use the Class

Now that we have the complete `DatabaseTable` class, let's use it in the controllers.

Firstly, delete `includes/DatabaseFunctions.php`. All our functions are now stored inside the class in `classes/DatabaseTable.php`.

Secondly, let's update `public/jokes.php` to use the new class:

```
<?php

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');

    $result = $jokesTable->findAll();

    $jokes = [];
    foreach ($result as $joke) {
        $author = $authorsTable->find('id', $joke['authorId'])[0];

        $jokes[] = [
            'id' => $joke['id'],
            'joketext' => $joke['joketext'],
            'jokedate' => $joke['jokedate'],
            'name' => $author['name'],
```

```
            'email' => $author['email']
        ];
    }

    $title = 'Joke list';

    $totalJokes = $jokesTable->total();

    ob_start();

    include  __DIR__ . '/../templates/jokes.html.php';

    $output = ob_get_clean();
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

This controller is better: it's using the `DatabaseTable` class and no longer using the `INNER JOIN` query. For now, each joke and its author is placed into an array called `$jokes`. We'll tidy this up in Chapter 13, but before we get to that, let's examine what we've achieved here.

The advantage is that we no longer have to provide the table name and `$pdo` instance to each of the functions—`total`, `find` and `findAll`. Notice that two instances of the `DatabaseTable` class are being created—one for the `joke` table and one for the `author` table. The functions can then each be called on either the `$jokesTable` variable or the `$authorsTable` variable to run the relevant query on either table.

Let's do the same thing with our other controllers.

Here's the updated `deletejoke.php`:

```
<?php
try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');

    $jokesTable->delete('id', $_POST['id']);

    header('location: jokes.php');
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Unable to connect to the database server: ' . $e->getMessage() .
    ↪ ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

And `editjoke.php`:

```
<?php
try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');

    if (isset($_POST['joke'])) {
        $joke = $_POST['joke'];
        $joke['jokedate'] = new DateTime();
        $joke['authorId'] = 1;

        $jokesTable->save($joke);

        header('location: jokes.php');
    } else {
        if (isset($_GET['id'])) {
            $joke = find($pdo, 'joke', 'id', $_GET['id'])[0] ?? null;
        }
        else {
```

```
          $joke = null;
        }
        $title = 'Edit joke';

        ob_start();

        include  __DIR__ . '/../templates/editjoke.html.php';

        $output = ob_get_clean();
    }
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

The example above can be found in **Example: OOP-DatabaseTable**[6].

Now that you're familiar with objects and classes, and you know that repeated code is a very bad thing for a programmer, you've probably already noticed that there's a lot of repetition in these files. It's time to start tidying up these controller scripts.

While making the last few changes, you'll have found yourself making similar changes in multiple locations. As I mentioned earlier in this book, the DRY (don't repeat yourself) principle states that it's bad practice to have repeated code.

DRY

Carefully examine the different controllers. What is actually different about them?

Each of the controllers follows this basic pattern:

---

6. https://github.com/spbooks/phpmysql7/tree/OOP-DatabaseTable

```
<?php
try {
    /*
        - include some required files
    */
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabasetabaseTable.php';

    /*

         - create one or more database table instances
    */
    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');

    /*
        - Do something that's unique to this particular page
            and create the $title and $output variables
    */
} catch (PDOException $e) {

    /*

        - Handle errors if they occur
    */
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}
    /*

        - Load the template file
    */
include  __DIR__ . '/../templates/layout.html.php';
```

Using this approach, if you wanted to rename the `DatabaseConnection.php` file, you'd have to go through each controller to use the new name. Similarly, if you wanted to change the layout file, you'd need to edit each controller separately.

All that really changes for each controller is the middle section that creates the `$output` and `$title` variables for the layout to use.

Rather than having different files for each controller, it's possible to write a single controller as a class that handles each *action* as a method. That way, we can have one file that handles all the parts that are common to each page, and methods in a class that handle the individual parts.

Creating a Controller Class

The first thing we could do is move the code for each controller into a method in a class.

We'll firstly create a class called `JokeController`. In this class, each method will represent a page of the website. There will be a method for the joke list page, a method for the edit joke page, a method for the delete joke page, and so on. As this is a special type of class, we won't store it in the `classes` directory. Instead, we'll create a new directory called `controllers` and save this as `JokeController.php`:

```
<?php
class JokeController {

}
```

When creating a controller class like this, a method representing a single page on the site is often referred to as an **action**. Before moving the relevant code into methods, let's consider what variables this class needs. Any variables required by the various actions will need to be class variables so that they can be defined once and used in any of the methods.

In this case, there are only two variables that are common to the controllers: `$authorsTable` and `$jokesTable`. We could add these two variables to the class:

```
class JokeController {
    private $authorsTable;
    private $jokesTable;
}
```

Like the `DatabaseTable` class, it's good practice to make these variables private so they can only be changed from within the class.

Also like the `DatabaseTable` class, rather than defining them in the constructor, you can use constructor property promotion to save a lot of work. Instead of:

```
class JokeController {
    private $authorsTable;
    private $jokesTable;

    public function __construct(DatabaseTable $jokesTable, DatabaseTable
    ↪$authorsTable) {
        $this->jokesTable = $jokesTable;
        $this->authorsTable = $authorsTable;
    }
}
```

… you can express this as follows, as long as you're using PHP 8.0 or higher:

```
class JokeController {
    public function __construct(private DatabaseTable $jokesTable, private
    ↪DatabaseTable $authorsTable) {

    }
}
```

Let's start adding the actions—the methods that each represent one page of the website. Add the `list` method first for the joke list page. Copy/paste the relevant section from `jokes.php`, but remember that only the code that's unique for this particular page should be placed in the method. Any code that exists on every page—such as the require statements at the top, or the code that loads the website layout—should be left out. Rather than repeating that code for every page of the website, we'll put that in a single location shortly.

Also, remember to use the class variables `$jokesTable` and `$authorsTable`, rather than including the code that creates them. We'll create the instances once and pass them into the controller:

```
public function list() {
    $result = $this->jokesTable->findAll();

    $jokes = [];
    foreach ($result as $joke) {
        $author = $this->authorsTable->find('id', $joke['authorId'])[0];

        $jokes[] = [
            'id' => $joke['id'],
            'joketext' => $joke['joketext'],
            'jokedate' => $joke['jokedate'],
            'name' => $author['name'],
            'email' => $author['email']
        ];

    }

    $title = 'Joke list';

    $totalJokes = $this->jokesTable->total();

    ob_start();

    include  __DIR__ . '/../templates/jokes.html.php';

    $output = ob_get_clean();

}
```

Before getting this working, let's add the other methods for the corresponding `editjoke` and `deletejoke` pages, along with the home page from `index.php`:

```
public function home() {
    $title = 'Internet Joke Database';

    ob_start();

    include  __DIR__ . '/../templates/home.html.php';

    $output = ob_get_clean();
}
```

```
public function delete() {
    $this->jokesTable->delete('id', $_POST['id']);

    header('location: jokes.php');
}

public function edit() {
    if (isset($_POST['joke'])) {

        $joke = $_POST['joke'];
        $joke['jokedate'] = new DateTime();
        $joke['authorId'] = 1;

        $this->jokesTable->save($joke);

        header('location: jokes.php');

    }
    else {

        if (isset($_GET['id'])) {
            $joke = find($pdo, 'joke', 'id', $_GET['id'])[0] ?? null;
        }
        else {
          $joke = null;
        }

        $title = 'Edit joke';

        ob_start();

        include  __DIR__ . '/../templates/editjoke.html.php';

        $output = ob_get_clean();
    }
}
```

If you examine the controller code closely, you might notice that, regardless of how we eventually use this class, it's not going to be very useful. That's because the `$title` and `$output` variables can never be used in `layout.html.php`. Once either the `home`, `edit` or `list` methods are run, the `$title` and `$output` variables, along with their contents, are lost.

To make those variables available to the code that calls the methods, we'll use the `return` keyword. We already used `return` in the `DatabaseTable` class. Each time a method was run, it was able to send some data back to the place it was called from. The `findAll` method returns an array of all the records in the table.

It would be possible to return the `$output` variable using `return $output`, but when `layout.html.php` is loaded, it will need both the `$output` and the `$title` variables.

Like the `findAll` method from the `DatabaseTable` class, the individual controller methods can return arrays:

```
public function home() {
    $title = 'Internet Joke Database';

    ob_start();

    include  __DIR__ . '/../templates/home.html.php';

    $output = ob_get_clean();

    return ['output' => $output, 'title' => $title];
}

public function list() {
    $result = $this->jokesTable->findAll();

    $jokes = [];
    foreach ($result as $joke) {
        $author = $this->authorsTable->find('id', $joke['authorId'])[0];

        $jokes[] = [
            'id' => $joke['id'],
            'joketext' => $joke['joketext'],
            'jokedate' => $joke['jokedate'],
            'name' => $author['name'],
            'email' => $author['email']
        ];
```

```
    }

    $title = 'Joke list';

    $totalJokes = $this->jokesTable->total();

    ob_start();

    include  __DIR__ . '/../templates/jokes.html.php';

    $output = ob_get_clean();

    return ['output' => $output, 'title' => $title];

}

public function edit() {
    if (isset($_POST['joke'])) {

        $joke = $_POST['joke'];
        $joke['jokedate'] = new DateTime();
        $joke['authorId'] = 1;

        $this->jokesTable->save($joke);

        header('location: jokes.php');

    }
    else {

        if (isset($_GET['id'])) {
            $joke = find($pdo, 'joke', 'id', $_GET['id'])[0] ?? null;
        }
        else {
          $joke = null;
        }

        $title = 'Edit joke';

        ob_start();

        include  __DIR__ . '/../templates/editjoke.html.php';
```

```
        $output = ob_get_clean();

        return ['output' => $output, 'title' => $title];
    }
}
```

Not a lot has changed here, but to make the `$output` and `$title` variables available outside the methods after they're called, a `return` statement has been added.

The `return` value of each of the functions is an array that contains the `output` and `title` variables. Now, when one of the methods is called, it will return the output and title strings, which can then be used.

If the `home` method were run, the returned array would look like this:

```
[
    'title' => 'Internet Joke Database',
    'output' => '<h2>Internet Joke Database</h2>
                <p>Welcome to the Internet Joke Database</p>'
]
```

Each method will return an array that contains the title for the page and the HTML that's going to be placed in the overall website layout.

Importantly, because each method returns data in the same format (an array with `output` and `title` keys), no matter which of the methods is called, we'll have an array with two variables. A method in the class will always either redirect to another page or provide an array in this format.

Until now, we've had each different page using its own file: `index.php`, `jokes.php`, `editjoke.php`, and `deletejoke.php`.

Single Entry Point

With the controller complete, we can now write a single file to handle any page. Importantly, this single file can contain all the code that was previously

repeated in each of the files.

The obvious question here is, if there are no longer individual files for each page, how does the code know whether to display the add joke page, the home page, or the list page?

As a starting point, we'll use `$_GET` variables. Going to `index.php?edit` will display the edit joke page, `index.php?list` will display the list page, and so on. We won't keep this approach for long, as it's not very efficient, but it will give you a crude example of how `index.php` can be used to display all the pages of the site. The code for this is shown below.

Example: OOP-EntryPoint[7]

```
<?php
try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/JokeController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');

    $jokeController = new JokeController($jokesTable, $authorsTable);

    if (isset($_GET['edit'])) {
        $page = $jokeController->edit();
    } else if (isset($_GET['delete'])) {
        $page = $jokeController->delete();
    } else if (isset($_GET['list'])) {
        $page = $jokeController->list();
    } else {
        $page = $jokeController->home();
    }

    $title = $page['title'];
    $output = $page['output'];
} catch (PDOException $e) {
```

7. https://github.com/spbooks/phpmysql7/tree/OOP-EntryPoint"

```
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

Save this over the top of `index.php` in the `public` directory and visit the home page at `https://v.je`. If everything is correct, the page will display as expected.

While you have the `public` directory open, delete `jokes.php`, `editjoke.php` and `deletejoke.php`. We've moved the relevant code from these files into `JokeController`, so they're no longer needed.

The new `index.php` page follows the same structure as each of our controllers. A lot of this code looks familiar, but let's go through the new lines individually.

```
$jokeController = new JokeController($jokesTable, $authorsTable);
```

This creates an instance of the `JokeController` class that we just wrote. When the constructor is called, it's passed the two instances of `DatabaseTable`, called `$jokesTable` and `$authorsTable`.

```
if (isset($_GET['edit'])) {
    $page = $jokeController->edit();
} else if (isset($_GET['delete'])) {
    $page = $jokeController->delete();
} else if (isset($_GET['list'])) {
     $page = $jokeController->list();
} else {
    $page = $jokeController->home();
}
```

This `if … else if` block is the clever part. These `if` statements examine the `$_GET` variables to determine which of the methods in the

`JokeController` class is called. Because of the `else` clause, at least one of these blocks is guaranteed to get executed.

Regardless of how this page is accessed, one of the methods in the `$jokeController` object will be run and the `$page` variable will be created, and it will contain two values—the page title, in the `title` key, and the page contents, stored under the `output` key. The contents of the array will be different for each page, but its structure will always be the same.

The final part creates the `$title` and `$output` variables for use in the template by reading them out of the newly created `$page` array:

```
$title = $page['title'];
$output = $page['output'];
```

To check all this works, open up the home page in your browser at `https://v.je/`.

Unfortunately, if you click any of the links—for example, the link to the jokes list—you'll see an error.

That's because there are no longer individual pages that represent each *page* of the website. Now, everything is diverted through `index.php`. To access any of the pages on the website, you'll have to type `index.php`, followed by a relevant URL variable.

To access the joke list page, you'll have to visit `https://v.je/index.php?list`.

This is called a **single entry point** or **front controller**.

We'll have to go through and change all links to the old pages to go via `index.php`, but before we do that, let's do some tidying up.

I've already described the new `index.php` as "crude", because it's not very efficient. Every time you want to add a page to the website, you'll need to do

two things:

- add the method in `JokeController`
- add the relevant `else if` block in `index.php`

You've probably already noticed that the `GET` variable name maps exactly to the name of the function:

- when `$_GET['edit']` is set, the `edit` function is called
- when `$_GET['list']` is set, the `list` function is called

This seems a bit redundant. PHP allows for some cool stuff. For example, you can do this:

```
$function = 'edit';
$jokeController->$function();
```

This will evaluate `$function` to `edit` and actually call `$jokeController->edit()`. We can utilize this feature to read the `GET` variable and call the method with that name.

We could use the `GET` variable `action` to call the relevant function on the controller. `index.php?action=edit` would call the edit function, `index.php?action=delete` would call delete, and so on. The code for this is remarkably simple:

```
<?php
try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/JokeController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');

    $jokeController = new JokeController($jokesTable, $authorsTable);
```

```
    $action = $_GET['action'] ?? 'home';

    $page = $jokeController->$action();

    $title = $page['title'];
    $output = $page['output'];
} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

The whole `if … else` block that selects the relevant action has been replaced with two lines:

```
$action = $_GET['action'] ?? 'home';
$page = $jokeController->$action();
```

The first line utilizes the confusingly named "null coalescing operator" I introduced in Chapter 7. This reads the `GET` variable called `action`. If it's set, `action` is read from the `GET` variable, and if it's not, `$action` is set to "home".

The second line calls the relevant method on the `$jokeController` object.

If you open your browser and visit `index.php?action=list`, you'll see the list of jokes. If you visit `index.php` without an `action` set, you'll see the home page.

The advantage of this approach is that, to add a new page to the website, all we need to do is add a method to the `JokeController` class and link to `index.php`, supplying the relevant action variable.

Now that the URL structure of the website has changed completely, we'll need

to go through each page and update any links, form actions, or redirects.

Firstly, `layout.html.php`:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="jokes.css">
    <title><?=$title?></title>
  </head>
  <body>
  <nav>
    <header>
      <h1>Internet Joke Database</h1>
    </header>
    <ul>
      <li><a href="index.php">Home</a></li>
      <li><a href="index.php?action=list">Jokes List</a></li>
      <li><a href="index.php?action=edit">Add a new Joke</a></li>
    </ul>
  </nav>

  <main>
  <?=$output?>
  </main>

  <footer>
  &copy; IJDB 2017
  </footer>
  </body>
</html>
```

Now open `jokes.html.php`, and change the “Edit” link and form action for deleting:

```
<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>

<?php foreach ($jokes as $joke): ?>
<blockquote>
  <p>
```

```
  <?=htmlspecialchars($joke['joketext'], ENT_QUOTES, 'UTF-8')?>

  (by <a href="mailto:<?=htmlspecialchars($joke['email'], ENT_QUOTES, 'UTF-8');
  ↪ ?>">
  <?=htmlspecialchars($joke['name'], ENT_QUOTES, 'UTF-8'); ?></a> on
<?php
$date = new DateTime($joke['jokedate']);

echo $date->format('jS F Y');
?>)
  <a href="index.php?action=edit&id=<?=$joke['id']?>">Edit</a>
  <form action="index.php?action=delete" method="post">
    <input type="hidden" name="id" value="<?=$joke['id']?>">
    <input type="submit" value="Delete">
  </form>
  </p>
</blockquote>
<?php endforeach; ?>
```

Finally, change the two redirects in `JokeController` from `header('location: jokes.php');` to `header('location: index.php?action=list');`.

The example above can be found in **Example: OOP-EntryPoint2**[8].

Keeping it DRY

A large proportion of your PHP code is now neatly organized into methods inside classes, and you can quickly add new pages to the website by simply creating a new method inside `JokeController`. Before we continue, let's quickly remove some of the remaining repeated code.

If you examine `JokeController`, most of the methods perform the same set of steps. The `edit` method contains this code:

```
ob_start();

include  __DIR__ . '/../templates/editjoke.html.php';
```

8. https://github.com/spbooks/phpmysql7/tree/OOP-EntryPoint2

```
$output = ob_get_clean();

return ['output' => $output, 'title' => $title];
```

The `home` method contains this code:

```
ob_start();

include  __DIR__ . '/../templates/home.html.php';

$output = ob_get_clean();

return ['output' => $output, 'title' => $title];
```

The `list` method contains this code:

```
ob_start();

include  __DIR__ . '/../templates/jokes.html.php';

$output = ob_get_clean();

return ['output' => $output, 'title' => $title];
```

These blocks of code are all very similar. Some of the lines are identical. As always, whenever you see repeated code like this, it's worth considering how it can be removed.

Rather than having each action include this block of code, it would be simpler to have the action provide a file name—such as `home.html.php`—and then have it loaded from within `index.php`.

To make that change, firstly open up `index.php` and change it to this:

```
<?php
try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
```

```
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/JokeController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');

    $jokeController = new JokeController($jokesTable, $authorsTable);

    $action = $_GET['action'] ?? 'home';

    $page = $jokeController->$action();

    $title = $page['title'];

    ob_start();

    include  __DIR__ . '/../templates/' . $page['template'];

    $output = ob_get_clean();

} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

I've moved the three repeated lines from the individual methods into `index.php`. `index.php` now expects the `$page` variable to provide a `template` key in place of the `output` key. Rather than the generated HTML code, the name of the template file is provided. Let's amend each of the controller actions to provide it. The controller actions will no longer provide the `$output` variable, but instead just a filename for `index.php` to include.

The `home` method:

```
public function home() {
    $title = 'Internet Joke Database';
```

```
    return ['template' => 'home.html.php', 'title' => $title];
}
```

The `list` method:

```
public function list() {
    $result = $this->jokesTable->findAll();

    $jokes = [];
    foreach ($result as $joke) {
        $author = $this->authorsTable->find('id', $joke['authorId'])[0];

        $jokes[] = [
            'id' => $joke['id'],
            'joketext' => $joke['joketext'],
            'jokedate' => $joke['jokedate'],
            'name' => $author['name'],
            'email' => $author['email']
        ];
    }

    $title = 'Joke list';

    $totalJokes = $this->jokesTable->total();

    return ['template' => 'jokes.html.php', 'title' => $title];
}
```

The `edit` method:

```
public function edit() {
    if (isset($_POST['joke'])) {

        $joke = $_POST['joke'];
        $joke['jokedate'] = new DateTime();
        $joke['authorId'] = 1;

        $this->jokesTable->save($joke);

        header('location: index.php?action=list');
```

```
    }
    else {

        if (isset($_GET['id'])) {
            $joke = find($pdo, 'joke', 'id', $_GET['id'])[0] ?? null;
        }
        else {
          $joke = null;
        }

        $title = 'Edit joke';

        return ['template' => 'editjoke.html.php', 'title' => $title];
    }
}
```

Each action now provides the name of a template that gets loaded in `index.php`. We've saved ourselves from needing to repeat the output buffer and `include` lines.

However, if you try the code above, only the home page will work. If you try viewing the list of jokes, you'll get an error:

```
Notice: Undefined variable: totalJokes in /websites/default/templates/
↪jokes.html.php on line 2
```

The reason for this error is that `jokes.html.php` is now being included from `index.php`, and `index.php` doesn't have the variable `$totalJokes` set.

We need a way to get the `$totalJokes` and `$jokes` variables, which are created in the `list` method, into `index.php`.

At first glance, you might think to do it in the `return` statement—the same way we did with the `title`, `output`, and later `template` variables:

```
return ['template' => 'jokes.html.php', 'title' => $title, 'totalJokes' =>
↪ $totalJokes, 'jokes' => $jokes];
```

And then we could recreate the variables in `index.php`:

```
$action = $_GET['action'] ?? 'home';

$page = $jokeController->$action();

$title = $page['title'];

$totalJokes = $page['totalJokes'];

$jokes = $page['jokes'];
ob_start();

include  __DIR__ . '/../templates/' . $page['template'];

$output = ob_get_clean();
```

If you try this, the jokes list page will work as expected. However, as soon as you navigate to another page, you'll get errors. For example, the edit joke page requires a variable called `joke`, and it doesn't provide the variables for `totalJokes` or `jokes`.

A very messy solution would be to have each method in the controller return every single variable that's needed, but leave the array values blank when they're not needed. The edit `return` statement would then end up looking like this:

```
return ['template' => 'jokes.html.php', 'title' => $title, 'totalJokes' => '',
↪ 'jokes' => '', 'joke' => $joke];
```

This is obviously not a viable solution. Each time we add a template that requires a variable with a new name, we'd need to amend every single controller method to provide an empty string for that variable and then amend `index.php` to set it!

Template Variables

Instead, we'll solve the problem in the same way we did for the `return`

statement. Each method will supply an *array* of variables.

The `list` return statement will now look like this:

```
return ['template' => 'jokes.html.php',
    'title' => $title,
    'variables' => [
        'totalJokes' => $totalJokes,
        'jokes' => $jokes
    ]
];
```

This is called a **multi-dimensional array**: there's an array inside an array. In this case, the `variables` key maps to a second array that contains the keys `totalJokes` and `jokes`.

Although this looks like a lot of pointless copying, the reason for this is to make the data in the variables available outside this function as part of the `$page` variable in `index.php`.

The code is slightly more difficult to read, but the advantage of this approach is that each controller method can provide a different array in the `variables` key. Each method will provide the name of the template to load and a list of variables that are required for that template.

The `editJoke` page can use this return statement:

```
return ['template' => 'editjoke.html.php',
    'title' => $title,
    'variables' => [
        'joke' => $joke
    ]
];
```

In the code above, the `joke` array key is mapped to `$joke ?? null`. You were probably expecting to see `'joke' => $joke`.

Both the `list` and `edit` controller actions now consistently return an array with the keys `template`, `title` and `variables`.

We can now use the `variables` array in `index.php`. The simplest way to achieve this would be to create a variable called `$variables` inside `index.php`, in the same way we did with `$title`:

```
$title = $page['title'];

$variables = $page['variables'];

ob_start();

include  __DIR__ . '/../templates/' . $page['template'];

$output = ob_get_clean();
```

Each template (such as `jokes.html.php`) now has access to the `$variables` array, and could read values from it—for example, by replacing this:

```
<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>
```

… with this:

```
<p><?=$variables['totalJokes']?> jokes have been submitted to the Internet Joke
↪Database.</p>
```

This solution works, but it means opening up and changing every template file. A simpler alternative is to create the variables that are required.

Luckily, PHP provides a method of doing exactly that. The `extract` function can be used to create variables from an array:

```
$array = ['hello' => 'world'];

extract($array);
```

```
echo $hello; // prints "world"
```

Only Use `extract` on Data You Control

The `extract` function creates variables from the given array. As such, it should only ever be used on data you directly control. In the example above, it's being used on values defined in the PHP script. This is fine, but it should *never* be used with `$_GET` or `$_POST`, as any existing variables will be overwritten by those provided by a user, potentially giving someone control over your PHP code!

A variable is created for any key in the array, and its value is set to the corresponding value from the array. We can use `extract` to create the relevant template variables in `index.php`.

Example: OOP-EntryPoint3[9]

```
$action = $_GET['action'] ?? 'home';

$page = $jokeController->$action();

$title = $page['title'];

if (isset($page['variables'])) {
    extract($page['variables']);
}

ob_start();

include  __DIR__ . '/../templates/' . $page['template'];

$output = ob_get_clean();
```

If `$page['variables']` is an array that's come from the `list` method, variables called `totalJokes` and `jokes` will be created. If it was created by

9. https://github.com/spbooks/phpmysql7/tree/OOP-EntryPoint3

the `edit` method, a single variable named `joke` will be created.

I've surrounded the `extract` call with `if (isset($page['variables']))`, because some methods, like the `home` method, may not need to provide a `variables` key, as the template doesn't have any variables.

Be Careful with `extract`

Everything is working perfectly, and we've managed to remove the repeated code from the controller's methods. Unfortunately, we're not quite done yet.

One of the biggest problems with `extract` is that it creates variables in the current scope. Take another look at this block of code:

```
$action = $_GET['action'] ?? 'home';

$page = $jokeController->$action();

$title = $page['title'];

if (isset($page['variables'])) {
    extract($page['variables']);
}

ob_start();

include  __DIR__ . '/../templates/' . $page['template'];
```

What would happen if the array `$page['variables']` contained the keys `page` and `title`? The `$title` and `$page` variables would be overwritten! It's likely the overwritten `$page` variable would not be an array with a key called `template` that contains the name of a template file.

If the return statement from a controller action happened to include a key called `page` in the `variables` array, it would prevent that controller action from loading a template.

It *is* possible to tell the extract function not to overwrite variables, but then if

the template is expecting a variable called `$page`, it's going to be given the wrong information.

A very simple solution to this is to move the code that loads the template into its own function. Amend `index.php` to the code below.

Example: OOP-EntryPoint-Template[10]

```
<?php
function loadTemplate($templateFileName, $variables) {
    extract($variables);

    ob_start();
    include  __DIR__ . '/../templates/' . $templateFileName;

    return ob_get_clean();
}

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/JokeController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');

    $jokeController = new JokeController($jokesTable, $authorsTable);

    $action = $_GET['action'] ?? 'home';

    $page = $jokeController->$action();

    $title = $page['title'];

    $variables = $page['variables'] ?? [];
    $output = loadTemplate($page['template'], $variables);

} catch (PDOException $e) {
    $title = 'An error has occurred';
```

10. https://github.com/spbooks/phpmysql7/tree/OOP-EntryPoint-Template

```
    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

I've created the `loadTemplate` function to load template files and create variables supplied in the `$variables` argument.

When the function is called, it's given the name of the template specified by the controller action in the `$page`, along with the variables specified in `$page['variables']`:

```
$variables = $page['variables'] ?? [];
$output = loadTemplate($page['template'], $variables);
```

Because the `'variables'` key may not exist, I've used the now familiar `??` operator to set the `$variables` variable to either the variables specified by the return value of the controller action or an empty array.

By moving the code that loads the template into its own function, if the `variables` array does contain keys called `page` or `title`, the existing variables won't be overwritten, because they don't exist inside the function's scope.

In this chapter, I've shown you how to use object-oriented programming to break up the code further and reduce repetition. We've also begun to add a clearer structure to the controller code.

With this new structure, adding a new page to the website is now a very simple task:

1. Create a function in `jokeController` that returns an array containing the template file name, page title and variables used by the template.

2. Access the page by specifying the name of the function in the URL: `index.php?action=functionName`.

You may have realized that this isn't very scalable. A website with 100 pages would require a controller with 100 functions. That would be very difficult to manage.

In the next chapter, I'll show you how to make `index.php` usable with other controllers and start thinking about code that we can use on different websites.

Summary

In this chapter, you've learned the basics of object-oriented programming and how to break code up into classes. With these changes in place, adding new pages to the website has become much simpler and requires a lot less copying and pasting (remember the DRY principle!).

Each page is now a function call accessed via `index.php` instead of having one file per page. As a result, all the setup code that happens on every page for connecting to the database and loading class files can be done in a single place.

Chapter 9

Creating an Extensible Framework

Now that you're able to write a controller with methods, and call those methods from `index.php`, the next step is to add the rest of the pages for managing the website. Currently, we can add jokes to the database through `index.php` by specifying an `action` URL parameter. However, a real website will need to do considerably more than handle basic database operations for a single table.

The next extension of the website will be allowing users to register as authors and post their own jokes. However, before we do that, I'll show you how to write a modern, flexible framework to build upon. By the end of this chapter, you'll have the foundation for building any website, and you'll have a good understanding of the techniques and concepts used by professional PHP developers.

We're not going to add any new features in this chapter. Instead, I'm going to show you how the code can be organized so that it can be reused on each website you build.

In the last chapter, we ended with this code in `index.php`:

```
<?php
function loadTemplate($templateFileName, $variables) {
    extract($variables);

    ob_start();
    include  __DIR__ . '/../templates/' . $templateFileName;

    return ob_get_clean();
}

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/JokeController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');
```

```
    $jokeController = new JokeController($jokesTable, $authorsTable);

    $action = $_GET['action'] ?? 'home';

    $page = $jokeController->$action();

    $title = $page['title'];

    $variables = $page['variables'] ?? [];
    $output = loadTemplate($page['template'], $variables);

} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

This lets us call any of the functions in the `JokeController` class by specifying the `action` URL parameter—for example, by linking to or visiting `index.php?action=list`.

Search Engine Optimization

Before we make any structural changes to the code, we need to do a little housekeeping. In PHP, functions aren't case-sensitive. `list` is treated exactly the same way as `LIST`. Due to case insensitivity, visiting `index.php?action=list` will display the page, but so will `index.php?action=LIST` or `index.php?action=List`. This may seem like a good thing, as people will be able to mistype the URL and still see the correct page. However, this feature can also cause problems for search engines.

Some search engines will see these two URLs as two entirely different pages, even if they display the exact same content. Both `index.php?action=LIST` and `index.php?action=list` will appear in search results. You might think "Great, more of my pages will appear in search results!" But search engines generally dislike "duplicate content", either ranking it lower or ignoring it altogether.

Would you prefer one result on the first page or two results potentially much further down in the search pages?

There are several ways to fix this. You can tell search engines to ignore certain pages. Or you can tell them which is the "canonical" (primary) version. But this can be difficult to manage on larger sites, and it's usually simpler to enforce strict URLs. A common way to do this is to force all URLs to be lowercase.

Forcing URLs to be lowercase is possible using a simple piece of PHP code, which detects whether or not the user entered a lowercase URL:

```
$action = $_GET['action'];

if ($action == strtolower($action)) {
    $jokeController->$action();
} else {
    echo 'Sorry that page does not exist.';
}
```

This code compares `$action` to a lowercase version of `$action`. The `strtolower` function converts any string to lowercase: `LISTJOKES`, `listJokes` or `listjokes` all become `listjokes`. By comparing the original `$action` to the lowercase version, it's possible to work out whether or not someone came to the page using a lowercase value for `$action`. If they didn't, an error is displayed. Visitors and search engines will only see the content on the lowercase version of the URL.

Configuration: PHP vs Server

You can do a lot of the work that processes the URL (such as forwarding any request to the lowercase version) in the server configuration instead of in the PHP script.

It's generally a good idea to avoid putting things in the server config file unless it's unavoidable, as the website logic is then spread among the PHP scripts and the server configuration. With logic in the server configuration, if you move the website to a different server (for example, to an Apache server) you have to recreate the same logic there, if it's even supported.

While doing this will prevent duplicate content and protect your search engine ranking, it's not very helpful for users who accidentally mistype the URL. Luckily, we can get the best of both worlds by redirecting non-lowercase pages to their lowercase equivalents.

We've already used the `header` function to redirect people to different pages. We can also use the `header` function to send all uppercase URLs to their lowercase equivalents:

```
<?php

$action = $_GET['action'] ?? 'home';

if ($action == strtolower($action)) {
    $jokeController->$action();
} else {
    header('location: index.php?action=' . strtolower($action));
}
```

Now anyone who visits `index.php?action=LISTJOKES` or `index.php?action=listJokes` will be redirected to `index.php?action=listjokes`. However, there's one more thing we need to do. There are two types of redirection, **temporary** and **permanent**. To tell search engines not to list the page, you need to tell them the redirection is

permanent.

This is done with an “HTTP response code”. You’ve probably come across at least one of these while browsing the Web. The code `404` means “Not found”. Each time a page is sent to the browser, a response code is sent along with it to tell the browser and search engines how to treat the page. To tell the browser that a redirect is permanent, you need to send the code `301`. PHP has a function called `http_response_code` that can be used to send the `301` response code along with the redirect:

```
<?php
$action = $_GET['action'] ?? 'home';

if ($action == strtolower($action)) {
    $jokeController->$action();
} else {
    http_response_code(301);
    header('location: index.php?action=' . strtolower($action));
    exit;
}
```

It’s also possible to use the second argument of the `header` function to set the response code. However, depending on which response you’re sending, you have to remember to specify the correct header string. The `http_response_code` function sets this string for you.

I’ve also added `exit;` after setting the header, as it terminates the script. Without this, PHP will continue processing the rest of the script, generating the HTML for a page that the user will never see.

HTTP Response Codes

There are many different HTTP response codes you can use. `404` is particularly useful when you display an error message, as this will stop the page appearing in search results and prevent the page from going into the browser's history. You don't generally want "Sorry, the product you requested is no longer available" appearing in search engines. You can even change the HTTP response code to `404` to get search engines to unlist pages that are no longer relevant.

A complete list of response codes and their meaning can be found on the W3.org website[1].

The complete `index.php`, including the redirect we just added, is now as shown below.

Example: CMS-Redirect[2]

```
<?php
function loadTemplate($templateFileName, $variables) {
    extract($variables);

    ob_start();
    include  __DIR__ . '/../templates/' . $templateFileName;

    return ob_get_clean();
}

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/JokeController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
```

1. https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
2. https://github.com/spbooks/phpmysql7/tree/CMS-Redirect

```
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');

    $jokeController = new JokeController($jokesTable, $authorsTable);

    $action = $_GET['action'] ?? 'home';

    if ($action == strtolower($action)) {
        $jokeController->$action();
    } else {
        http_response_code(301);
        header('location: index.php?action=' . strtolower($action));
        exit;
    }

    $page = $jokeController->$action();

    $title = $page['title'];

    $variables = $page['variables'] ?? [];
    $output = loadTemplate($page['template'], $variables);

} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

The `index.php` code we have allows any function to be called in the `JokeController` class by specifying the `action` URL parameter.

Using a single file to handle any controller action is an improvement over having a unique file for each action, because it avoids us repeating code. But what happens when the website grows? If there are 100 pages, this will require 100 methods in the `JokeController` class. Finding the right piece of code will require a lot of scrolling down, and large files are difficult to manage.

Thinking Ahead: User Registration

The next step for us will be to allow someone to register as a user so they can post a joke. To do this, we'll need a new page on the website. Although it would be possible to keep adding methods to the `JokeController` class, to avoid that class becoming excessively large, we'll put the code for registering—and later logging in—in its own controller.

A programming practice developers follow is the "single responsibility principle", which states that a class should only do one thing. While our controllers don't quite follow it to the letter, separating out the actions relating to jokes to the actions related to users is a step towards that. If the joke actions and registration actions were in the same file, the class wouldn't be useful on another website, but the code for user registrations alone may be!

Let's create a new controller called `AuthorController` with some methods to handle user registration. This helps keep the code manageable, by keeping anything to do with jokes in `JokeController` and any page related to user registration in `AuthorController`.

With the `index.php` above, we'd need to write a new PHP script to utilize `AuthorController`, such as `author.php`, that looked like this:

```
<?php
try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/AuthorController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');

    $authorController = new AuthorController($authorsTable);

    $action = $_GET['action'] ?? 'home';

    if ($action == strtolower($action)) {
```

```
        $page = $authorController->$action();
    } else {
        http_response_code(301);
        header('location: index.php?action=' . strtolower($action));
    }

    $title = $page['title'];

    $variables = $page['variables'] ?? [];
    $output = loadTemplate($page['template'], $variables);

} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

Most of this code is identical to `index.php`. To make this support `AuthorController.php` instead of `JokeController`, the following changes were required:

- `include 'JokeController.php';` becomes `include 'AuthorController.php';`
- `$jokeController = new JokeController($jokesTable, $authorsTable);` becomes `$authorController = new AuthorController($authorsTable);`
- `$jokeController->$action();` becomes `$authorController->$action();`

Other than these minor changes, the code is identical to the previous version. While we could potentially have a different version of `index.php` for every controller, following the DRY principle means it would be better if a single `index.php` could work with any controller. In the same way, it currently works with any action, avoiding the need for different PHP files for loading each controller. To achieve that, we'd need to amend the code above so that it can work with any controller.

We'll rewrite the three changes listed above in a generic way.

1. Include the Relevant Controller

Having the include statement load the correct controller class is simple to implement. As we saw when loading templates, the `include` statement can be used to include files using a string stored in a variable.

As you already know, variables can be built up from other variables, including `$_GET` variables. Using the same process we used to define `$action`, it's also possible to specify a URL parameter for `controller`, like so: `index.php?controller=joke&action=listJokes`. We could use this to load `JokesController` and call the action `listJokes` or use the URL `index.php?controller=register&action=registrationForm` to call the action `registrationForm` in `AuthorController`.

Using the `GET` variable `controller`, it would be possible to `include` the relevant controller class by using the name of the controller as part of the filename:

```
$controllerName = ucfirst($_GET['controller']) . 'Controller';

include __DIR__ . '/../controllers/' . $controllerName . '.php';
```

However, this is incredibly insecure. A user could amend the URL to load any PHP script on the website they like! Dynamic includes like this should be avoided whenever possible. Instead, although inefficient (and we'll deal with that a little later), just include both controllers regardless of which is used:

```
include __DIR__ . '/../controllers/JokeController.php';
include __DIR__ . '/../controllers/AuthorController.php';
```

2. Create an Instance of the Controller

Step two is having a line like below, which can work with any controller:

```
$jokeController = new JokeController($jokesTable, $authorsTable);
```

You might consider substituting the class name in a similar way to create an instance of the relevant controller:

```
$controller = new $controllerName($jokesTable, $authorsTable);
```

You might be surprised that this is valid code. `$controllerName` will be evaluated and an instance of the class whose name is stored in `$controllerName` will be created.

With this code, by visiting `index.php?controller=joke&action=listJokes` the `$controller` variable will contain an instance of `JokeController`, and visiting `index.php?controller=register&action=...` will create an instance of `AuthorController`.

However, you may have spotted a potential problem. When the controller is created, its constructor is called and passed the `$jokesTable` and `$authorsTable` objects.

Dependencies

Different controllers will inevitably require different dependencies. The `JokeController` we built in the last chapter requires the `$jokesTable` and `$authorsTable` objects, but not all controllers will require those same objects.

An object that's required by another object is called a **dependency**. For example, `JokeController` is *dependent* on the `$jokesTable` instance, as without it, it won't work correctly.

To identify a dependency in a piece of code, look for a function call on another object. For example, the `delete` method in the controller depends on the `jokesTable` variable, and that variable must contain a `DatabaseTable` instance. Without a `DatabaseTable` instance, the `delete` method below can't work. It's *dependent* on functionality from another class:

```
public function delete() {
    $this->jokesTable->delete($_POST['id']);
    header('Location: .');
    exit();
}
```

The method being called `jokesTable->delete()` is in another class. If the `DatabaseTable` class didn't exist, this delete function would fail. We can say that the `JokeController` class is *dependent* on the `DatabaseTable` class. Likewise, we can say that the `DatabaseTable` class has a dependency on the `PDO` class, because it can't function without it.

We're going to add a second controller, `AuthorController`, that deals with allowing new authors to register so they can post jokes.

To begin with, we'll create a single form for registration. When submitted, information will need to be inserted into the `author` table. There's no reason this controller will ever need to interact with the `$jokesTable` object, so a `AuthorController` object would be instantiated like this:

```
$controller = new AuthorController($authorsTable);
```

Other controllers might need other database tables—for example, a `categories` table for categorizing the jokes or objects that don't even deal with database access, to validate data that's been entered.

We face a problem here. We can use a variable in place of the class name like so:

```
$controllerName = $_GET['controller'];
$controller = new $controllerName($authorsTable);
```

But there's no easy way to determine what dependencies the required controller needs.

I'll warn you now, this is the most complicated topic in this book, and

something even very experienced developers struggle with! Different people have come up with some potential solutions, and there are many approaches you can take. However, many are considered “bad practice” and should be avoided. I could write a book on this subject alone (it’s a large section of my PhD thesis!) so instead of telling you what not to do, I’m going to stick with best practices and show you a few options for handling it in the preferred way.

What Not to Do

If you want to learn more about what *not* to do, here are some useful references.

Creating the objects in the constructor of the controller:

- “How to Think About the “new” Operator with Respect to Unit Testing”[3]
- “Dependency Injection for Loose Coupling”[4]
- “Flaw: Constructor does Real Work”[5]

On singletons:

- “Singletons are Pathological Liars”[6]
- “What’s so bad about the Singleton?”[7]

On service locators:

- “Managing Class Dependencies: An Introduction to Dependency Injection, Service Locators, and Factories, Part 2”[8]
- “Flaw: Digging into Collaborators”[9]
- “Service Locator is an Anti-Pattern”[10]

---

3. http://misko.hevery.com/2008/07/08/how-to-think-about-the-new-operator/
4. http://www.codeproject.com/Articles/13831/Dependency-Injection-for-Loose-Coupling
5. http://misko.hevery.com/code-reviewers-guide/flaw-constructor-does-real-work/
6. http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/
7. http://www.sitepoint.com/whats-so-bad-about-the-singleton/
8. https://www.sitepoint.com/managing-class-dependencies-2/
9. http://misko.hevery.com/code-reviewers-guide/flaw-digging-into-collaborators/

It's a very good idea to pass dependencies into the constructor of classes that need them. As I mentioned in the last chapter, this stops an object being able to exist without having the dependencies set. The problem we have is that different controllers will require different dependencies. The `JokesController` class constructor looks like this:

```
public function __construct(private DatabaseTable $jokesTable, private
↪DatabaseTable $authorsTable) {
}
```

And when we write the code for the `AuthorController` class, the constructor will look like this:

```
public function __construct(private DatabaseTable $authorsTable) {

}
```

The `JokesController` class has dependencies on two `DatabaseTable` objects—one for authors and one for jokes. The `AuthorController` class only has a dependency on one, `$authorsTable`. If we're trying to automate creation of the controllers, it presents a problem: if the constructors are different, how can the objects be automatically created?

One method of fixing this would be to ensure that all controllers have the same constructor. They all require access to all the possible `DatabaseTable` objects. This works, but it's messy. It results in controllers with dependencies on everything that any controller may ever need. One major downside to this approach is that, when a new database table is added, all the controllers' constructors must be changed. We could overcome this by passing an array of all the possible dependencies and picking out the ones we need. This is essentially something known as a "service locator", and it's a common approach, although it's been widely considered bad practice over the last few years. (See the "What Not to Do" note box above for more information on service locators.)

---

10. http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/

The technical term for what we're doing is **dependency injection**. It sounds complicated, but it's just a fancy term for passing dependencies into constructors. You've been doing it all along without even knowing!

The simplest way of solving the problem of different constructors needing different arguments is a series of if statements. This way, each controller can be created with the correct dependencies:

```
$action = $_GET['action'] ?? 'home';
$controllerName = $_GET['controller'] ?? 'joke';

if ($controllerName === 'joke') {
    $controller = new JokeController($jokesTable, $authorsTable);
}
else if ($controllerName === 'author') {
    $controller = new AuthorController($authorsTable);
}

$page = $controller->$action();
```

Now the controller is selected by the variable supplied in `$_GET['controller']`, and the method defined in `$_GET['action']` is called on the controller object. If `$_GET['controller']` isn't set, it defaults to `home`, and if `$_GET['action']` isn't set, it defaults to `home`.

This approach is very flexible. It allows us to call any method in any controller by specifying the class name in the `controller` URL variable and method name in the `action` URL variable.

3. Call the Action on the Correct Controller

The third change is very easy to make. We need to call the correct method on the correct controller. After making the previous change, only one controller is instantiated and stored in the `$controller`. It doesn't matter which controller it is. You can just use this code:

```
$controller->$action();
```

If `$controller` stores a `JokeController` instance, it will call a method in the `JokeController` class. If it stores a `AuthorController` instance, a method in the `AuthorController` class will be called.

Done

The complete `index.php` is shown below.

Example: CMS-Controller[11]

```
<?php
function loadTemplate($templateFileName, $variables = []) {
    extract($variables);

    ob_start();
    include  __DIR__ . '/../templates/' . $templateFileName;

    return ob_get_clean();
}

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/JokeController.php';
    include __DIR__ . '/../controllers/AuthorController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');

    $action = $_GET['action'] ?? 'home';
    $controllerName = $_GET['controller'] ?? 'joke';

    if ($controllerName === 'joke') {
        $controller = new JokeController($jokesTable, $authorsTable);
    }
    else if ($controllerName === 'author') {
```

11. https://github.com/spbooks/phpmysql7/tree/CMS-Controller

```
        $controller = new AuthorController($authorsTable);
    }

    if ($action == strtolower($action) && $controllerName ==
    ➥strtolower($controllerName)) {
        $page = $controller->$action();
    } else {
        http_response_code(301);
        header('location: index.php?controller=' . strtolower($controllerName)
        ➥ .'&action=' . strtolower($action));
    }

    $title = $page['title'];

    $variables = $page['variables'] ?? [];
    $output = loadTemplate($page['template'], $variables);

} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

Note that I've also updated the `if` statement that checks the URL is the correct case. It now also checks that the controller name is lowercase, and the redirect to send users to the lowercase controller and action if they aren't already.

Ideally, we were looking to be able to use *any* controller without editing `index.php`. But for simplicity's sake, we'll stick with this approach.

Now that they've changed, we'll also need to amend `layout.html.php` to use the new routes in the menu:

```
<li><a href="index.php?controller=joke&amp;action=list">Jokes List</a></li>
<li><a href="index.php?controller=joke&amp;action=edit">Add a new Joke</a></li>
```

Before we go ahead and change all the links throughout the website, these URLs are getting long and cumbersome, and to avoid that, I want to introduce an approach called "URL Rewriting".

URL Rewriting

A lot of websites are written in PHP, including Facebook and Wikipedia. If you visit one of these sites, you'll see that the URLs don't look like the ones we've been using on the joke website.

The URL for SitePoint's Wikipedia page is `https://en.wikipedia.org/wiki/SitePoint`, and its Facebook page URL is `https://www.facebook.com/sitepoint/`.

Using the structure we've looked at so far, you'd probably expect to see something like `https://www.facebook.com/index.php?controller=page&id=sitepoint` or `https://en.wikipedia.org/index.php?controller=wiki&action=sitepoint`.

Most PHP websites don't actually show you the PHP filename in the URL. Many years ago, search engines penalized sites that made heavy use of URL variables, and the trend was to use more friendly URLs to rank higher in search engines. These days, search engines don't care about URL structure[12], and friendly URLs are used more for aesthetic reasons. When you paste a link on social media, for example, it's preferable to be able to offer a clean, concise address.

As most websites use friendlier URLs, it's useful to know how to do it.

The default behavior of a web server is to map a URL directly to a filename. If you visit `website.com/logo.png`, it will look for `logo.png` in its `public` directory. Visiting `about-the-company.php` will look for the PHP script based on the URL.

---

12. https://www.sitepoint.com/friendly-urls/"

This mapping of a URL to a file on the sever is a sensible convention, but it's only the default behavior and can be overridden. If you wanted to, you could instruct the web server to load `home.php` from its `public` directory when someone visits `homepage.php`.

A URL and a file are only linked by this default behavior, but they're actually entirely independent of each other.

URL rewriting is a tool for changing this convention. You can configure your web server so that, when someone visits `/jokes/list`, it actually runs `index.php?controlelr=joke&action=list`, or even when someone visits `contact.php` it instead it runs `index.php?action=contact`.

Importantly, the URL in the address bar can be completely different from the PHP script that's actually being run on the server, and the person viewing the page can just see the more friendly URL.

URL rewriting is a long and complex topic. You can set up all kinds of wonderful and impressive rules. However, almost all modern PHP websites use the same easy-to-implement rule: if a file requested doesn't exist, load `index.php`.

In fact, the Docker environment we're using is already set up to do this. Visit `https://v.je/I-dont-exist`, or any whimsical filename you can think of, and you'll see the website home page rather than an error page.

If you create the file `I-dont-exist` and visit the page in the browser, it will be loaded. Otherwise, all requests are sent to `index.php`.

Configuring NGINX and Apache

If you need to configure an NGINX server for URL rewriting, the guide on the NGINX website[13] is the first place to look for examples.

However, for most setups you'll just need the configuration directive:

```
location / {
   try_files $uri $uri/ /index.php?$args;
}
```

For Apache servers, the same can be achieved by creating a file called `.htaccess` in the `public` (or, more likely for Apache, `public_html` or `httpdocs`) directory with the following contents:

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^.*$ /index.php [NC,L,QSA]
```

Make sure that your server is configured to allow `.htaccess` files (`AllowOverride all`) and that the `mod_rewrite` Apache module is loaded.

How to configure a server is beyond the scope of this book, but the `.htaccess` file above will have the same effect as the pre-configured Docker environment. If a file doesn't exist, it will load `index.php` rather than display an error. More information on configuring URL rewriting when using Apache can be found in the SitePoint article "Learn Apache mod_rewrite: 13 Real-world Examples"[14].

You know just enough about URL rewriting to make use of it on the site.

13. https://www.nginx.com/blog/creating-nginx-rewrite-rules/
14. https://www.sitepoint.com/apache-mod_rewrite-examples/

Rather than using a `$_GET` variable to determine the route, you can use the URL that the person used to connect to the website. PHP supplies this information in the variable `$_SERVER['REQUEST_URI']`.

We're going to use a URL in the format `/controller/action`. For example, `/joke/list` will be equivalent to `index.php?controller=joke&action=list`. A URL that maps to a controller action is commonly referred to a **route**, as it determines the route the script takes through the application.

Open up `index.php` and remove this:

```
$action = $_GET['action'] ?? 'home';
$controllerName = $_GET['controller'] ?? 'joke';
```

Replace it with this:

```
$uri = strtok(ltrim($_SERVER['REQUEST_URI'], '/'), '?');

if ($uri == '') {
    $uri = 'joke/home';
}

$route = explode('/', $uri);

$controllerName = array_shift($route);
$action = array_shift($route);
```

There's a lot going on here. Before I explain this code step by step, I'll show you the result of what it does.

The route is read from `$_SERVER['REQUEST_URI']` and converted into an array in the `$route` variable. If you visit the URL `/joke/list`, the `$route` variable will contain this array:

```
[
    'joke',
    'list'
```

```
]
```

The entries in this array are then used to set the `$controller` and `$action` variables.

Conceptually, this is fairly simple, and the code needed to get there seems excessive. There are three different functions being called: `explode`, `strtok` and `ltrim`. Let's take a look at how this actually works.

I'll break it apart inside out, starting with `$_SERVER['REQUEST_URI']`. This reads the URL that the user is seeing. If the user navigates to `https://v.je/joke/list`, the variable `$_SERVER['REQUEST_URI']` will store `/joke/list`.

As the next step, `ltrim($_SERVER['REQUEST_URI'], '/')` removes the leading `/`. The `ltrim` function (short for "left trim") is used to remove characters from the start of the string. In this case, it removes any `/` from the beginning so `/joke/list` becomes `joke/list`.

Next, `strtok` is used:

```
$uri = strtok(ltrim($_SERVER['REQUEST_URI'], '/'), '?');
```

This removes the query string. If you were to visit `/joke/edit?id=4`, the `$_SERVER['REQUEST_URI']` would include the query string (the question mark and everything after it). For our purposes, we don't want the query string. We're only interested in everything before it.

The `strtok` function breaks a string apart on a separator and returns everything before it. This takes `joke/edit?id=4` and converts it to just `joke/edit` and stores that in the `$uri` variable. I've placed this in its own variable, as we'll need it again later.

Finally, the `explode` function takes two arguments: a string, and a separator. The string, in this instance, is the URI after the query string has been removed (for example, `joke/edit`) and the separator is `/`. It then splits the string on

the `/` character and produces the array `['joke', 'edit']`:

```
$route = explode('/', $uri);
```

While it would be possible to use the `strtok` function again here, we might want to have URLs with more than two levels, and using `explode` enables us to keep everything after `joke/edit`.

Once the string has been exploded, the `$rotue` variable contains an array with two entries.

You can read the first entry from the array and remove it using the `array_shift` function. Unlike reading the value using `$route[0]`, `array_shift($route)` will read whatever is in index 0, then remove it from the array, leaving only the rest of the array.

Let's say we run `array_shift($route)` twice, like so:

```
$controllerName = array_shift($route);
$action = array_shift($route);
```

If the `$route` array only had two items to begin with, it would be empty after this code has run.

The only remaining issue is this: what if someone just visited the home page `https://v.je`? There would be no URL to split on `/` and specify the controller and action. To solve this, I've explicitly set the `$uri` variable if it's empty:

```
if ($uri == '') {
    $uri = 'joke/home';
}
```

If someone visits `https://v.je`, the `$uri` variable is empty, and it's overwritten with `joke/home`. From there, the script runs as if someone had

visited `https://v.je/joke/home`.

Finally, we'll update the case sensitivity check. As we no longer have different `$_GET` variables, we can do one check on the URI up to the `?`, again using `strtok`:

```
if ($uri == strtolower($uri)) {
    $page = $controller->$action();
} else {
    http_response_code(301);
    header('location: /' . strtolower($uri));
}
```

Plug this into your `index.php` and it will work much like before. The only difference now is that the URLs are much more user-friendly. Instead of visiting `https://v.je/index.php?controller=joke&action=edit`, you can visit the much nicer `https://v.je/joke/edit` to view the page.

The complete `index.php` looks like this:

```
<?php
function loadTemplate($templateFileName, $variables = []) {
    extract($variables);

    ob_start();
    include  __DIR__ . '/../templates/' . $templateFileName;

    return ob_get_clean();
}

try {
    include __DIR__ . '/../includes/DatabaseConnection.php';
    include __DIR__ . '/../classes/DatabaseTable.php';
    include __DIR__ . '/../controllers/JokeController.php';
    include __DIR__ . '/../controllers/AuthorController.php';

    $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
    $authorsTable = new DatabaseTable($pdo, 'author', 'id');
```

```
    $uri = strtok(ltrim($_SERVER['REQUEST_URI'], '/'), '?');

    if ($uri == '') {
        $uri = 'joke/home';
    }

    $route = explode('/', $uri);

    $controllerName = array_shift($route);
    $action = array_shift($route);

    if ($controllerName === 'joke') {
        $controller = new JokeController($jokesTable, $authorsTable);
    }
    else if ($controllerName === 'author') {
        $controller = new AuthorController($authorsTable);
    }

    if ($uri == strtolower($uri)) {
        $page = $controller->$action();
    } else {
        http_response_code(301);
        header('location: /' . strtolower($uri));
    }



    $title = $page['title'];

    $variables = $page['variables'] ?? [];
    $output = loadTemplate($page['template'], $variables);

} catch (PDOException $e) {
    $title = 'An error has occurred';

    $output = 'Database error: ' . $e->getMessage() . ' in ' .
    $e->getFile() . ':' . $e->getLine();
}

include  __DIR__ . '/../templates/layout.html.php';
```

Try visiting `https://v.je/joke/list`. The page will load, but it will look strange. It will be black and white, and missing all the styling. That's because of

this line in *layout.html.php*:

```
<link rel="stylesheet" href="jokes.css">
```

When the browser sees the line above, it will look for *jokes.css* at *https://v.je/joke/jokes.css*. When a browser encounters */* in a URL, it treats it as a directory. If you view the page *https://v.je/joke/list*, the browser thinks you're viewing a file called *list* in a directory called *joke*.

As all links are relative to the directory the file being viewed is in, the browser looks for *jokes.css* in the *joke* directory. As that file doesn't exist, the stylesheet isn't applied.

There are two possible fixes:

- The HTML *<base>* tag. Although it's a viable solution, it introduces a few issues that aren't worth covering here.
- Make all URLs relative to the domain. To do this, just prefix all links with a */*.

The second option is more work, but it causes fewer issues. Let's open up *layout.html* and find this line:

```
<link rel="stylesheet" href="jokes.css">
```

We'll amend it to this:

```
<link rel="stylesheet" href="/jokes.css">
```

By prefixing a link in an HTML document with a forward slash (*/*), it tells the browser to look for the file from the top level of the website. If we refresh the page, we'll see the styles now display correctly.

Let's amend each link on the website to use the new prettier format.

Here's `layout.html.php`:

```
<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/joke/list">Jokes List</a></li>
  <li><a href="/joke/edit">Add a new Joke</a></li>
</ul>
```

We'll also need to update the two redirects in `JokeController.php`:

```
header('location: /joke/list');
```

There are two more URLs that need to be updated—the edit link and the delete form action in `jokes.html.php`. Here's the `action` attribute for the delete button form:

```
<form action="/joke/delete" method="post">
  <input type="hidden" name="id" value="<?=$joke['id']?>">
  <input type="submit" value="Delete">
</form>
```

Next, we'll update the edit link for each joke. It's possible to amend the URL and keep the `$_GET` variable, like so:

```
<a href="/joke/edit?id=<?=$joke['id']?>">Edit</a>
```

But to keep the URL as clean and tidy as possible, we can make one more amendment to the way URLs are handled in `index.php`.

Rather than using the URL `/joke/edit?id=1` to edit a joke, we can use the cleaner `joke/edit/1` with a couple of small tweaks to `index.php` and `JokeController.php`.

Currently, the `edit` method in the `JokeController` class uses `$_GET['id']` to determine which joke is being edited:

```
public function edit() {
    if (isset($_GET['id'])) {
        $joke = $this->jokesTable->find('id', $_GET['id'])[0] ?? null;
    }
    //...
```

Rather than using a `$_GET` variable, this method could be updated to take an *argument*, like so:

```
public function edit($id = null) {
    if (isset($id)) {
        $joke = $this->jokesTable->find('id', $id)[0] ?? null;
    }
    //...
```

I've added an argument `$id` with a default value of `null`. This means that, if the method is called without an argument, `$id` will be set to `null`, but if an argument *is* provided, it will use the `id` given.

If we can get `index.php` to take the `1` from the URL `/joke/edit/1` and pass it to the `edit` method as an argument, we can use this tidier URL.

Earlier, I used `array_shift` to remove the controller name and action from the URL.

Given the URL `joke/edit` after this code has run:

```
$controllerName = array_shift($route);
$action = array_shift($route);
```

... the `$route` variable will be empty.

However, if there were more levels to the URL—for example, `/joke/edit/1`—there would still be a `1` in the array after removing the controller name and action. The `$route` array would look like this:

```
[
    1
]
```

In `index.php`, the method in the controller is run using this line:

```
$page = $controller->$action();
```

For the edit page, we could use the following code to send the ID of the joke being editing into the method as an argument:

```
$page = $controller->$action($route[0]);
```

Conceptually this works: when the URL `joke/edit/1` is visited, `joke` specifies the controller (the class), `edit` is the action (method to call), and then `1` is sent to the method as an argument. We can think of our URL as being `/controller/action/argument`.

This particular code will cause two problems. Most notably, there will be an error if you visit `/joke/list`, because there's no third value set, and `$route[0]` here won't exist once the controller and action have been removed.

But what happens if we wanted a controller action with more than one `$_GET` variable? Or, using our new approach, more than one argument?

We could format our URL as `/controller/action/argument1/argument2/argument3`, and then call the method with this code:

```
$page = $controller->$action($route[0], $route[1], $route[2]);
```

This works, and is roughly what we want to do, but it's not very flexible. What if we wanted two arguments, or four?

A fairly recent addition to PHP is the **argument unpacking operator** (also

sometimes referred to as the **splat operator**), `...`. This operator enables you to use an array as the arguments for a function:

```
$fruits = [
    'apple',
    'banana'
];

eat(...$fruits);
```

The code above is equivalent to this:

```
eat('apple', 'banana');
```

We can utilize this operator when calling the action:

```
$page = $controller->$action(...$route);
```

By using argument unpacking here, any values left in the `$route` array will be sent to the method as arguments. This code will work whether there are zero arguments or a hundred. Each argument can then be specified in the URL by separating them with a `/`.

You can now amend the edit joke link as shown below.

Example: CMS-Controller-Rewrite[15]

```
<a href="/joke/edit/<?=$joke['id']?>">Edit</a>
```

Now you know why we're making these structural changes before our website has too many pages!

With this change in place, all the pages are now accessible at friendly URLs such as `/joke/list`. Besides aesthetics, there are some further benefits:

15. https://github.com/spbooks/phpmysql7/tree/CMS-Controller-Rewrite

- In future, you could swap out your PHP website for an ASP.NET, Node.js or other web technology and all the URLs could still work.
- Attackers can't get as much information about how the website works, because they can't see that it uses PHP or what the `GET` variables are. Still, don't put *too* much faith in this as a security measure, as disguising the URL won't deter anyone but the laziest hacker.

Tidying Up

You've probably noticed that `index.php` is getting a bit long and unwieldy. Before we get into creating a new controller to handle user registration, let's tidy up `index.php` a little.

Make it OOP

One of the primary causes of overly complex code is nested `if` statements. It's possible to break up any long piece of code into a single class with a set of methods.

This can be done by identifying unique tasks within the code. Looking at the code, we can see the following distinct tasks:

- instantiating the controller and calling the relevant action based on `$route`
- the `loadTemplate` function
- redirecting to a lowercase version of the URL if required
- loading the relevant template file and setting its variables

Let's take each of these and make it a function inside a class called `EntryPoint`, inside the `classes` directory.

As a starting point, let's break up `index.php` a little to make it easier to manage. Firstly, we already have a function called `loadTemplate`. This function can just be made into a method in the class. I've made it private, as we're only going to call it from another method in the same class:

```
private function loadTemplate($templateFileName, $variables = []) {
    extract($variables);

    ob_start();
    include  __DIR__ . '/../templates/' . $templateFileName;

    return ob_get_clean();
}
```

We can also separate out the logic that ensures the URL is lowercase. Again, it's only going to be used in this class, so I've made it private. It will take one argument—the URI of the page—and check to see if it's lowercase. If not, it redirects to the lowercase version of the page:

```
private function checkUri($uri) {
    if ($uri != strtolower($uri)) {
        http_response_code(301);
        header('location: ' . strtolower($uri));
    }
}
```

I've amended the logic very slightly. Instead of running the action in the controller if the page is lowercase and redirecting if it's not, this function redirects to the lowercase version of the page if the URI is *not* lowercase.

For now, let's paste in the rest of the code from `index.php` into a method called `run`. The `run` method will be public, as we're going to be calling it from outside the class, and it will take a single argument—the URI of the page without the query string:

```
public function run($uri) {
    try {
        include __DIR__ . '/../includes/DatabaseConnection.php';
        include __DIR__ . '/../classes/DatabaseTable.php';
        include __DIR__ . '/../controllers/JokeController.php';
        include __DIR__ . '/../controllers/AuthorController.php';

        $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
```

```
        $authorsTable = new DatabaseTable($pdo, 'author', 'id');

        $this->checkUri($uri);

        if ($uri == '') {
            $uri = 'joke/home';
        }

        $route = explode('/', $uri);

        $controllerName = array_shift($route);
        $action = array_shift($route);


        if ($controllerName === 'joke') {
            $controller = new JokeController($jokesTable, $authorsTable);
        }
        else if ($controllerName === 'author') {
            $controller = new AuthorController($authorsTable);
        }

        $page = $controller->$action(...$route);

        $title = $page['title'];

        $variables = $page['variables'] ?? [];
        $output = $this->loadTemplate($page['template'], $variables);

    } catch (PDOException $e) {
        $title = 'An error has occurred';

        $output = 'Database error: ' . $e->getMessage() . ' in ' .
        $e->getFile() . ':' . $e->getLine();
    }

    include  __DIR__ . '/../templates/layout.html.php';
}
```

After pasting, I made a couple of changes. The `$uri` variable will be created before being sent to this function, so the line that creates it has been removed. In addition, the code for checking the URI and calling to the `loadTemplate` function now runs the functions we've created inside the class.

We're going to break this up further, but now's a good time to test that everything's working. We're going to use this class from `index.php`, amending the contents to this:

```
include_once '../classes/EntryPoint.php';

$uri = strtok(ltrim($_SERVER['REQUEST_URI'], '/'), '?');

$entryPoint = new EntryPoint();
$entryPoint->run($uri);
```

We just made some very substantial changes. Although none of the code is new, and it doesn't produce any different output, the structure is completely different. Take a look through the completed `EntryPoint.php` to see how it works. Each task is now in its own method, rather than being nested inside a series of if statements.

Our `index.php` and `EntryPoint.php` files can now be used to load any controller class, and to call any method on it, by specifying the appropriate route.

A controller can easily be added by creating a class in the `controllers` directory and adding the logic for creating the controller and calling the relevant action in the `run` method.

As every single page on the website is now going to be loaded using the `EntryPoint` class, it's worth taking some time to make sure it's correct. Before we add another controller, let's consider how code can be reused on a much larger scale than we've looked at until now.

Reusing Code on Different Websites

Now that we've tidied up `index.php`, it's worth considering what we've achieved by doing so. We've broken up the code into more easily manageable chunks, and the code is easier to read. If there's a problem with the redirect check, you know to look in the `checkUri` method. If a template isn't loading

correctly, you know to look in the `loadTemplate` method. And if a URL isn't displaying the page it should, you know to look in the `run` method.

Even though we haven't finished our Internet Joke Database website yet, it's worth thinking about your *next* website. You probably didn't buy this book so you could make a website for people to post jokes. You likely have a real project in mind that you're planning to build using the knowledge you learn from this book.

How much of the code we've written so far can be used *without modification* in your next website?

We specifically built the `DatabaseTable` class so that it can work with *any* database table. Not only can it work with tables that exist for the joke website— `joke` and `author` —but it could also work with tables called `customer`, `product` and `order` for a shopping website, or `account` and `message` on a social media website, or indeed any database table on any website.

Generic or Project-specific?

Besides the `DatabaseTable` class, how much of the code we've written so far would be useful on another website? The templates probably wouldn't. Another website would likely have completely different HTML, its forms would have completely different fields, and the website would deal with a different topic.

The controllers would be different. The code in `JokeController` is very specific to the joke website we're building. It's unlikely the code in the controller will be useful without changing it.

However, the code we just wrote in the `EntryPoint` class—which loads controllers and template files—would be useful on another website. The templates and controllers loaded would be different, but the code to load those files would be the same.

There are two types of code files in any given website:

- project-specific files containing code that's only relevant to that particular website
- generic, reusable files containing code that can be used to build any website

The more code we can make *generic*, the bigger the foundation we have to work from when we start a new website. If we can use a lot of code from our previous website on our next website, we'll save ourselves a lot of time.

Rather than having to write code that's similar to the `EntryPoint` and `DatabaseTable` classes for our next website, we could save a lot of time by using the code we already have.

This foundation is called a **framework**—a body of generic code (usually classes) that can be quickly built upon to create any website. It doesn't contain any code that's specific to one particular project.

It's important to make a distinction between *framework code* and *project-specific code*. If we can successfully separate them, we can reuse our framework code in every website we build, saving significant upfront development time. If we have framework code mixed with project-specific code, we'll find ourselves writing very similar code for every website we build.

When you first start a project, it can be difficult to recognize which parts of the code belong specifically to that project and which can be used across different projects.

As a rule of thumb, processes are generic, but data is specific. For example, *adding a joke to the database* is specific to the joke site, but *adding to the database* is a generic process that's needed on most websites.

In Chapter 8, I showed how to separate out the generic *add to database* process from the project-specific process of *adding a joke*. Anything related to *jokes* is in `JokeController.php`, but all the code related to *adding to the*

database is stored in `DatabaseTable.php`.

By identifying the process and separating it from the project-specific data being worked with, we can repeat the same process of *adding to the database* with any data on any website by reusing the `DatabaseTable` class.

The first step to making something *generic* is usually placing it inside a *class*. This helps us break up the problem into smaller parts. Once we've broken the problem up into individual methods, we can then see which are *generic* and which are *project-specific*.

A dead giveaway that something is *project-specific* is a hardcoded value or variable name that alludes to something for that project only.

Let's apply that to the new `EntryPoint` class. The methods `loadTemplate`, `checkUri` and `run` don't contain any references to *jokes*, *authors*, or anything that's specific to the joke website. Indeed, we'll need a way of loading controllers and templates on future websites; they just won't be dealing with jokes and authors.

However, the `run` method contains several references to *jokes* and *authors*. If we wanted to reuse this class on a different website—for example, an online shop—we'd need to rewrite the entire method. It won't have controllers or database tables dealing with `jokes` authors; it will have controllers and database tables for `products`, `customers`, and `orders`.

Let's imagine we do have two websites—the joke website and an online shop. We've copy/pasted the file `EntryPoint.php` and changed `run` to suit each website. If there's a bug in the `checkUri` or `run` methods, we'll need to fix the bug in two places, while being careful not to change any code specific to that website.

Instead, if the file contained only the *generic* framework code, we could overwrite the old `EntryPoint.php` with the new one everywhere it's being used, and fix the bug without worrying about undoing changes that were

made specifically for one project.

Making `EntryPoint` Generic

The answer, then, is to remove all references to project-specific concepts from otherwise generic classes.

This process is more of an art than a science, and even experienced developers can struggle to work out where to draw the line between generic and project-specific code. However, I'm going to show you a fairly simple, step-by-step process that can be used to remove a project-specific method from a class, turning it into a framework class.

Firstly, identify the method you want to remove. In this case, the `run` method contains a lot of code that's specific to the joke website:

```
public function run($uri) {
    try {
        include __DIR__ . '/../includes/DatabaseConnection.php';
        include __DIR__ . '/../classes/DatabaseTable.php';
        include __DIR__ . '/../controllers/JokeController.php';
        include __DIR__ . '/../controllers/AuthorController.php';

        $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
        $authorsTable = new DatabaseTable($pdo, 'author', 'id');

        $this->checkUri($uri);

        if ($uri == '') {
            $uri = 'joke/home';
        }

        $route = explode('/', $uri);

        $controllerName = array_shift($route);
        $action = array_shift($route);


        if ($controllerName === 'joke') {
```

```
            $controller = new JokeController($jokesTable, $authorsTable);
        }
        else if ($controllerName === 'author') {
            $controller = new AuthorController($authorsTable);
        }

        $page = $controller->$action(...$route);

        $title = $page['title'];

        $variables = $page['variables'] ?? [];
        $output = $this->loadTemplate($page['template'], $variables);

    } catch (PDOException $e) {
        $title = 'An error has occurred';

        $output = 'Database error: ' . $e->getMessage() . ' in ' .
        $e->getFile() . ':' . $e->getLine();
    }

    include  __DIR__ . '/../templates/layout.html.php';
}
```

Let's go through the code here and determine which parts are specific to the joke website and which are generic. Where possible, we'll identify a generic process that we could separate out from specific steps needed only for the joke website.

Conceptually, what we're going to do is fairly simple. We'll take all the code that's specific to the joke website and move it into another class, while leaving all the code that can be used on any website in the `EntryPoint` class we already have.

The tricky part is deciding which parts of the code need to stay and which need to be moved. I'll start with a fairly simple part first, so you can get the hang of how this problem is solved.

Look at this block of code:

```
if ($uri == '') {
    $uri = 'joke/home';
}
```

This sets up the script so that, later on, it runs the `home` method from the `joke` controller. This is the logic here: if the URI is blank, run a specific action that will remain the same on every website, but not every website will have a `home` method in a controller we're calling `joke`.

To begin with, we'll create a class to contain all the code that's specific to the joke website. In this instance, I'll move the code `'joke/home'` into a method in a class. Save this as `JokeWebsite.php` in the `classes` directory:

```
class JokeWebsite {
    public function getDefaultRoute() {
        return 'joke/home';
    }
}
```

This isn't very complicated code. It has a single method that returns the default route for the joke website, in this case `joke/home`.

To use this class, the `EntryPoint` class needs to have it as a *dependency*:

```
class EntryPoint {
    public function __construct(private $website) {
    }

    public function run($uri) {
        try {
            include __DIR__ . '/../includes/DatabaseConnection.php';
            include __DIR__ . '/../classes/DatabaseTable.php';
            include __DIR__ . '/../controllers/JokeController.php';
            include __DIR__ . '/../controllers/AuthorController.php';

            $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
            $authorsTable = new DatabaseTable($pdo, 'author', 'id');
```

```
            $this->checkUri($uri);

            if ($uri == '') {
                $uri = $this->website->getDefaultRoute();
            }
            //...
```

I've used the shorthand property promotion again here to create the `$this->website` class variable, but notice what's happening: when the `EntryPoint` class is constructed, it will need to be given an instance of `JokeWebsite`:

```
$website = new JokeWebsite();
$entryPoint = new EntryPoint($website);
```

Then, when it comes to setting the `$uri` variable for the home page, it reads it from the `$this->website` instance, which will call the `getDefaultRoute()` method in the `JokeWebsite` class:

```
if ($uri == '') {
    $uri = $this->website->getDefaultRoute();
}
```

We've added a new class with new method, a constructor and class variable in the existing `EntryPoint` class, just to provide a simple string: `joke/home`. This seems like a lot of work for something so trivial.

However, consider what's happening here. If we developed a new website—such as an online shop—we could provide a version of the `JokeWebsite` class for that particular website:

```
class OnlineShopWebsite {
    public function getDefaultRoute() {
        return 'product/onsale';
    }
}

// and in index.php
```

```
$website = new OnlineShopWebsite();
$entryPoint = new EntryPoint($website);
```

By passing the `OnlineShopWebsite` instance into the `EntryPoint` class, when the `getDefaultRoute` method is run, it will return `product/onsale` and load the sale page as the home page!

This method is relatively simple, but it illustrates the technique of moving site-specific code out of one class and into another. By doing so, we can create another class, with the same set of methods, that exists for a completely different website.

We'll build on this idea and move all of the code that's specific to the joke website into the `JokeWebsite` class, leaving just the code that's useful on any website in the `EntryPoint` class.

The `EntryPoint` class currently contains a lot of code for creating objects needed for the joke website. Firstly, there are the various `DatabaseTable` instances, and slightly further down there's the code that creates the controllers:

```
include __DIR__ . '/../includes/DatabaseConnection.php';
include __DIR__ . '/../classes/DatabaseTable.php';
include __DIR__ . '/../controllers/JokeController.php';
include __DIR__ . '/../controllers/AuthorController.php';

$jokesTable = new DatabaseTable($pdo, 'joke', 'id');
$authorsTable = new DatabaseTable($pdo, 'author', 'id');

//...
if ($controllerName === 'joke') {
    $controller = new JokeController($jokesTable, $authorsTable);
}
else if ($controllerName === 'author') {
    $controller = new AuthorController($authorsTable);
}
```

The first half of this code is creating `DatabaseTable` instances so they can

later be used when creating the controller classes. Look through the code: neither the `$jokesTable` nor `$authorsTable` variables are used anywhere other than when the controllers are created.

Let's take all this code and move it into a function in the `JokeWebsite` class:

```
class JokeWebsite {
    public function getDefaultRoute() {
        return 'joke/home';
    }

    public function getController(string $controllerName) {
        include __DIR__ . '/../includes/DatabaseConnection.php';
        include __DIR__ . '/../classes/DatabaseTable.php';
        include __DIR__ . '/../controllers/JokeController.php';
        include __DIR__ . '/../controllers/AuthorController.php';

        $jokesTable = new DatabaseTable($pdo, 'joke', 'id');
        $authorsTable = new DatabaseTable($pdo, 'author', 'id');

        if ($controllerName === 'joke') {
            $controller = new JokeController($jokesTable, $authorsTable);
        }
        else if ($controllerName === 'author') {
            $controller = new AuthorController($authorsTable);
        }

        return $controller;
    }
}
```

I've added one line of code here at the bottom—`return $controller`—because we need to use this variable in the `EntryPoint` class. This method also needs some information that comes from the `EntryPoint` class, so I've added `$controllerName` as an argument, with a type hint that specifies that it must be a string.

Notice that I haven't actually changed any of the code here. I've just moved it into its own method and provided it with the variable it uses as an argument.

Conceptually, this method is actually fairly simple. It's passed the name of the controller as an argument (for example, `joke`) and then returns the corresponding fully constructed object (for example, an instance of `JokeController`).

Using it on its own, it would look like this:

```
$website = new JokeWebsite();
$jokeController = $website->getController('joke');
$authorController = $website->getController('author');
```

Now that the code is in the `JokeWebsite` class, the `EntryPoint` class can be used to call the new method rather than including all this code. Here's the complete `EntryPoint` class with the `getController` call to the `JokeWebsite` class:

```
class EntryPoint {
    public function __construct(private $website) {
    }

    public function run($uri) {
        try {
            $this->checkUri($uri);

            if ($uri == '') {
                $uri = $this->website->getDefaultRoute();
            }

            $route = explode('/', $uri);

            $controllerName = array_shift($route);
            $action = array_shift($route);

            $controller = $this->website->getController($controllerName);

            $page = $controller->$action(...$route);

            $title = $page['title'];

            $variables = $page['variables'] ?? [];
```

```
            $output = $this->loadTemplate($page['template'], $variables);

        } catch (PDOException $e) {
            $title = 'An error has occurred';

            $output = 'Database error: ' . $e->getMessage() . ' in ' .
            $e->getFile() . ':' . $e->getLine();
        }

        include  __DIR__ . '/../templates/layout.html.php';
    }

    private function loadTemplate($templateFileName, $variables) {
        extract($variables);

        ob_start();
        include  __DIR__ . '/../templates/' . $templateFileName;

        return ob_get_clean();
    }

    private function checkUri($uri) {
        if ($uri != strtolower($uri)) {
            http_response_code(301);
            header('location: ' . strtolower($uri));
        }
    }
```

For completeness, the whole of `index.php` that makes use of this class is shown below.

Example: CMS-EntryPoint-Framework[16]

```
include_once '../classes/EntryPoint.php';
include_once '../classes/JokeWebsite.php';

$uri = strtok(ltrim($_SERVER['REQUEST_URI'], '/'), '?');

$jokeWebsite = new JokeWebsite();
$entryPoint = new EntryPoint($jokeWebsite);
```

16. https://github.com/spbooks/phpmysql7/tree/CMS-EntryPoint-Framework

```
$entryPoint->run($uri);
```

There's now *nothing* in the `EntryPoint` class that's specific to the joke website! We have a generic `EntryPoint.php`. There are no longer any references to *jokes*, *authors*, or anything specific to one particular website.

If we wanted to use this code on another website, we'd just need to create a class that had the `getController` and `getDefaultRoute` methods and then adjust `index.php` to create the alternate object. For example, for an online shop you could create this class:

```
class OnlineShopWebsite {
    public function getDefaultRoute() {
        return 'product/onsale';
    }

    public function getController(string $controllerName) {
        include __DIR__ . '/../includes/DatabaseConnection.php';
        include __DIR__ . '/../classes/DatabaseTable.php';
        include __DIR__ . '/../controllers/ProductController.php';
        include __DIR__ . '/../controllers/SearchController.php';

        $productsTable = new DatabaseTable($pdo, 'joke', 'id');

        if ($controllerName === 'product') {
            $controller = new ProductController($productsTable);
        }
        else if ($controllerName === 'search') {
            $controller = new ProductController($productsTable);
        }

        return $controller;
    }
}
```

A new website can be added by making any class that follows this structure. As long as it includes the correct methods, you can use the existing `EntryPoint` class on any website you build in the future, saving a significant amount of time.

The `index.php` for the shop website would contain this code:

```
$onlineShopWebsite = new OnlineShopWebsite();
$entryPoint = new EntryPoint($onlineShopWebsite);
$entryPoint->run($uri);
```

Autoloading

We're often repeating the `include` line of code to include a relevant class each time a class is required.

Any time we use one of the classes we've created, it must be referenced with an `include` statement. This can get tricky to manage, as you need to ensure the class file has been included before you use the class. On top of that, if you accidentally issue the `include` statement twice for the same class, you'll see an error.

Our `JokeWebsite` class has to include the `DatabaseTable` and controller classes, and `index.php` has to include the files that contain the `EntryPoint` and `JokeWebsite` classes.

Some pages may require some classes to be loaded, whereas others may require other classes to be loaded.

A very inefficient but easy-to-organize method of managing loading classes would be to include every single class at the top of the `index.php` file, so that any class that might be needed has already been loaded. Using this method, we'll never have to write an `include` statement for a class outside `index.php`.

A major drawback of this approach is that, each time you add a new class to the project, you'll have to open up `index.php` and add the relevant `include` statement. This is time-consuming, and will use an unnecessary amount of memory on the server, as all classes would be loaded whether they're needed or not.

I've advised placing classes in their own files throughout this book, as well as naming files to match identically the name of the classes they contain. The class `DatabaseTable` is inside the file `DatabaseTable.php`, `JokeController` is stored in `JokeController.php`, and `EntryPoint` is stored in a file called `EntryPoint.php`, and so on.

One of the reasons I've advised structuring files this way is that it's considered good practice. It helps someone reading the code to find the classes that are referenced. If they want to look at the code for `JokeController`, they know to look in `JokeController.php`.

One other advantage of a standardized file structure is that PHP contains a feature called "autoloading". **Autoloading** is used to *automatically load* PHP files that store classes. As long as your filenames are consistent with the class names, it's easy to write an autoloader to load the relevant PHP file.

Once we've written an autoloader, we'll never need to write an `include` line for a class anywhere in the project.

When we use the statement `new ClassName()`, if the class `ClassName` doesn't exist (because it hasn't been included), PHP can trigger an *autoloader* that can then load the file `ClassName.php`, and the rest of the script will continue as normal without us ever needing to manually write the line `include 'ClassName.php';`.

An **autoloader** is a function that takes a class name as an argument and then includes the file that contains the corresponding class. The function can be as simple as this:

```
function autoloader($className) {
  $file = __DIR__ . '/../classes/' . $className . '.php';
  include $file;
}
```

It would be possible to use this function manually to save a little time:

```
autoloader('DatabaseTable');
autoloader('EntryPoint');
```

This would include both `DatabaseTable.php` and `EntryPoint.php`. However, it's possible to instruct PHP to call this function automatically whenever it can't find a class that's referenced:

```
spl_autoload_register('autoloader');
```

The function `spl_autoload_register` is built into PHP and allows us to tell PHP to call the function with the name we've given if it comes across a class that hasn't yet been included.

The `autoloader` function will be called automatically when a class is used for the first time:

```
function autoloader($className) {
  $file = __DIR__ . '/../classes/' . $className . '.php';
  include $file;
}

spl_autoload_register('autoloader');

$jokesTable = new DatabaseTable($pdo, 'joke', 'id');
$controller = new EntryPoint($jokesTable);
```

Now files will automatically be included the first time the class stored in them is used. `new DatabaseTable` will trigger the autoloader with `DatabaseTable`, as the `$className` argument and `DatabaseTable.php` will be included.

Case Sensitivity

PHP classes aren't case-sensitive, but filenames usually are. This can cause a problem with autoloaders. The first time a class is used it will be included, and `new DatabaseTable` will load `DatabaseTable.php`. However, `new databasetable` will cause an error, because the filename is case-sensitive and `databasetable.php` doesn't exist.

So a problem is caused in a situation like this:

```
$jokesTable = new DatabaseTable($pdo, 'joke', 'id');
$authorstable = new databasetable($pdo, 'author', 'id');
```

The code above will work as intended, because the first time `DatabaseTable` is loaded with the correct case, the file is successfully included, and PHP's case insensitivity allows both objects to be constructed.

However, if we reverse the order of arguments—because the autoloader is triggered with a lowercase name—we'll get an error:

```
$authorstable = new databasetable($pdo, 'author', 'id');
$jokesTable = new DatabaseTable($pdo, 'joke', 'id');
```

An alternative is to make all filenames lowercase and have the autoloader convert the class name to lowercase before loading the file. Although this is a more robust approach and arguably a better technical implementation, it goes against PHP community conventions[17], and it will cause problems if we want to share our code with other people in the future.

Implement an Autoloader

Let's implement an autoloader. To keep things organized, let's create `autoload.php` and save it in the `includes` directory:

```
<?php
function autoloader($className) {
    $file = __DIR__ . '/../classes/' . $className . '.php';
    include $file;
}

spl_autoload_register('autoloader');
```

Now we can amend `index.php` to include the autoloader, but remove the

17. http://www.php-fig.org/psr/psr-4/

include lines that explicitly include `EntryPoint.php` and `IjdbRoutes.php`:

```
<?php
include __DIR__ . '/../includes/autoload.php';

$uri = strtok(ltrim($_SERVER['REQUEST_URI'], '/'), '?');

$jokeWebsite = new JokeWebsite();
$entryPoint = new EntryPoint($jokeWebsite);
$entryPoint->run($uri);
```

We can also remove the include line for `DatabaseTable` from `JokeWebsite.php`.

Example: CMS-EntryPoint-Autoload[18]

```
<?php
class JokeWebsite {
    public function getDefaultRoute() {
        return 'joke/home';
    }

    public function getController(string $controllerName) {
        include __DIR__ . '/../includes/DatabaseConnection.php';

        include __DIR__ . '/../controllers/JokeController.php';
        include __DIR__ . '/../controllers/AuthorController.php';
```

Notice that `DatabaseConnection.php` is still included manually, because it doesn't include a class. It sets up the `$pdo` variable, which is used by both `DatabaseTable` objects. Autoloaders can only be used to load classes, and that's one of the reasons it's a good idea to structure as much of our code as possible inside classes.

Redecorating

The autoloader works for all classes that are inside the `classes` directory, but

18. https://github.com/spbooks/phpmysql7/tree/CMS-EntryPoint-Autoload

doesn't work for `JokeController` or `AuthorController`. The controllers are still loaded in `JokeWebsite.php` on several lines:

```
include __DIR__ . '/../controllers/JokeController.php';
```

We might try removing this line and having the autoloader load it automatically. If we do, we'll see an error. That error occurs because, when PHP encounters `new JokeController` and triggers the autoloader, it attempts to load `JokeController.php` from the `classes` directory, rather than from the `controllers` directory where the file is actually stored.

Earlier in this chapter, I mentioned the difference between *framework* code—the code you might want to use on every website you build—and project-specific code that exists only for one particular website.

It's a good idea to keep these separated in different directories, so that you can easily copy/paste *framework* files between websites without copying files that are specific to a single project.

Let's name our framework after the title of this book, *Ninja*. Move all the framework code into a directory called `Ninja` at the top level of the project (the same folder that the `classes` directory is in at the moment). We should move `EntryPoint.php` and `DatabaseTable.php`, as these are our two *generic* framework files.

Similarly, let's create another directory at the top level called `Ijdb`. This is where we'll keep all the code that's specific to the joke site and can't be reused on future websites. We'll move `JokeSite.php` into the `Ijdb` directory and move the `controllers` directory inside as well.

While we're moving things around, and for consistency, let's give the `Controllers` directory an uppercase "C" and ensure the `Ninja` and `Ijdb` directories all start with an uppercase letter.

When we're finished, we should have the following file/folder structure:

- `Ninja\DatabaseTable.php`
- `Ninja\EntryPoint.php`
- `Ijdb\JokeWebsite.php`
- `Ijdb\Controllers\JokeController.php`
- `Ijdb\Controllers\AuthorController.php`

Don't try loading the website yet! As we've moved all the files around, everything is broken!

If we do try loading a page at this point, we'll see some errors. That's because the autoloader is now looking in the wrong place.

To solve this, we could add some logic to the autoloader that looks at the name of the class and loads the file from the correct location, or store an array of class names mapped to filenames.

Instead, we're going to use a new tool: namespaces.

Namespaces

Each of our classes has a unique name. However, are they truly unique? If we download some code someone else has written, it may well contain a class named `DatabaseTable` or `EntryPoint`. In a world with thousands of PHP developers, these names aren't unique.

Modern PHP is great: we can find code online that does almost anything we can dream of—such as creating graphs and charts, turning web pages into PDFs, manipulating images and videos, connecting to Twitter streams, and controlling services on a Raspberry Pi. The list is almost endless.

What if we found a great-looking library we wanted to use, but it included a class named `DatabaseTable`? This name clash would present a problem.

As class names have to be unique in PHP, either our `DatabaseTable` class could be loaded or the one we downloaded could be. When we run the line

`new DatabaseTable`, PHP has to know whether you're referring to your `DatabaseTable` class or the one someone else wrote that you found online, which happens to use the same class name but contains completely different code.

One feature that has revolutionized PHP and made it much easier to share code online is **namespaces**.

Before namespaces came along, PHP developers would name their classes with a prefix. For example, we might name our classes `Ninja_EntryPoint`, `Ninja_DatabaseTable` and `Ijdb_JokeController`.

That way, when we wanted to use `SuperLibary_DatabaseTable`, it wouldn't clash with `Ninja_DatabaseTable`, and we could use both `DatabaseTable` classes on the same website.

Namespaces provide a simpler method of solving the same problem. Every class we write can (and should!) be placed within a namespace.

You can think of namespaces a bit like folders on your computer. Inside any given folder on your computer, each file has to have a unique name. For example, our `public` directory can only contain one file named `index.php`, but a different directory could also contain a different file also named `index.php`.

Let's move our framework files into the `Ninja` namespace. At the top of `EntryPoint.php` and `DatabaseTable.php`, add the following code:

```
namespace Ninja;
```

The first few lines of `DatabaseTable.php` now look like this:

```
<?php
namespace Ninja;
```

```
class DatabaseTable {

    public function __construct(private PDO $pdo, private string $table, private
    ↪ string $primaryKey) {
  // …
```

Now add the namespace `Ijdb` to `JokeWebsite.php`:

```
<?php
namespace Ijdb;

class JokeWebsite {
    public function getDefaultRoute() {
    // …
```

Before giving the final classes (`JokeController` and `AuthorController`) a namespace, I'll show you how to use the classes now that they have a namespace.

`index.php` has the code `new EntryPoint` and `new JokeWebsite`. Now that the classes are inside namespaces, this won't work. We'll need to specify the namespace when instantiating the class by using a backslash (`\`), followed by the namespace, another backslash, and then the class.

These lines:

```
$jokeWebsite = new JokeWebsite();
$entryPoint = new EntryPoint($jokeWebsite);
```

… will become this:

```
$jokeWebsite = new \Ijdb\JokeWebsite;
$entryPoint = new \Ninja\EntryPoint($jokeWebsite);
```

Similarly, in `JokeWebsite.php`, we need to change `new DatabaseTable` to `new \Ninja\DatabaseTable`:

```
$jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
$authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id');
```

At this point, we might be inclined to add the namespace `Ijdb` to `JokeController`. We'll be giving `JokeController` a namespace containing `Ijdb`, but we'll give it the namespace `Ijdb\Controllers`.

The backslash (`\`) in the namespace represents a **sub-namespace**—a namespace within a namespace. This isn't strictly necessary, but it's a good idea to keep related code together. In this case, we'll place all controllers inside the `Ijdb\Controllers` namespace and the `Ijdb/Controllers` directory.

While we're changing the `JokeController.php` file to include the namespace, we'll rename the class (and file) to `Joke`. That way, the class is `\Ijdb\Controllers\Joke` rather than `\Ijdb\Controllers\JokeController`, and the word "Controller" isn't repeated unnecessarily in the full class name.

This parallel between directory structures and namespaces is important. It allows us to write an autoloader that can use both namespaces and class names to locate the file it needs to load.

The combined namespace and class name now represent the folder structure, making it easy to autoload the classes.

This convention is known as **PSR-4** (PSR stands for PHP Standards Recommendations), and it's used by almost all modern PHP projects. Each class should be contained inside a file that directly maps to its namespace and class name. The full class name, including namespace, should exactly match the directory and filename, including case sensitivity. The PSR-4 standard also provides a few other rules that I won't go into here. To read more about PSR-4, take a look at the PHP-FIG website[19].

---

19. http://www.php-fig.org/psr/psr-4/

Autoloading with PSR-4

By using PSR-4, it's simple to convert a class name in a namespace to a file path. Let's replace `autoload.php` with this PSR-4 version:

```
<?php
function autoloader($className) {
    $fileName = str_replace('\\', '/', $className) . '.php';

    $file = __DIR__ . '/../' . $fileName;

    include $file;
}

spl_autoload_register('autoloader');
```

When the autoloader is triggered with a class inside the namespace, it's passed the entire class name, including the namespace. For example, when `EntryPoint` is loaded, the autoloader is given the class name `Ninja\EntryPoint`.

The line `str_replace('\\', '/', $className) . '.php';` replaces backslashes with forward slashes to represent the file in the file system. `Ninja\EntryPoint` becomes `Ninja/EntryPoint.php`, referencing the file.

With the autoloader in place, we can now remove the `include` lines from `IjdbRoutes.php`, which load `JokeController` and `AuthorController`:

```
include __DIR__ . '/../controllers/JokeController.php';
include __DIR__ . '/../controllers/AuthorController.php';
```

While you're editing these include lines, the database connection is being included here as well. Since it's only used in this class, let's just move the code from the `DatabaseConnection.php` into this class. Find this line:

```
include __DIR__ . '/../includes/DatabaseConnection.php';
```

Replace it with the contents of `DatabaseConnection.php`:

```
$pdo = new PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
↳ 'mypassword');
```

There should no longer be any `include` lines in the `Joke` class.

Then we should change the reference to the controllers to use the full class name:

```
$controller = new \Ijdb\Controllers\Joke($jokesTable, $authorsTable);
```

Nearly there! We're close to having the site up and running again using the new file and namespace structure. However, if we try to load one of the pages, we'll see this error:

```
Uncaught TypeError: Argument 1 passed to Ninja\DatabaseTable::__construct() must
↳be an instance of Ninja\PDO, instance of PDO given
```

To fix it, we can open up `DatabaseTable.php` and change the type hint in the constructor from `PDO` to `\PDO`. We saw this error because namespaces are *relative*. If you provide a reference to a class name—in a type hint or following the `new` keyword—PHP will look for a class with that name in the current namespace. We also need to replace `DateTime` with `\DateTime` in `Joke.php` and `DatabaseTable.php.` and `PDOException` with `\PDOException` in this `DatabaseTable.php`.

Because the `DatabaseTable` class is inside the `Ninja` prefix, PHP—without the backslash prefix—will want to load the class `\Ninja\PDO` rather than the inbuilt PHP class `PDO`.

The `PDO` class is in something called the **global namespace**—meaning it's a class that exists at the very top level, effectively not inside a namespace. To reference a class in the global namespace, we must prefix it with a backslash.

Save `DatabaseTable.php` and refresh the page. There's just one more error to fix:

```
Fatal error: Uncaught TypeError: Argument 1 passed to Ijdb\Controllers\Joke::
↪__construct() must be an instance of Ijdb\Controllers\DatabaseTable, instance
↪of Ninja\DatabaseTable given
```

This is the same problem as above. Because there's no namespace specified, PHP looks for a class called `DatabaseTable` in the *current* namespace. As the controller is in the `Ijdb` namespace, PHP will try to load the class `Ijdb\DatabaseTable`.

We could fix this the same way as above—by providing the class name with the namespace (which PHP refers to as a **fully qualified class name**): `\Ninja\DatabaseTable`. But a neater solution is to *import* the class `DatabaseTable` into the current namespace.

We can do this with the `use` keyword after the namespace declaration.

Example: CMS-EntryPoint-Namespaces[20]

```
<?php
namespace Ijdb\Controllers;
use \Ninja\DatabaseTable;

class Joke {
    // …
```

If we've done everything correctly, we should be able to refresh the website and see everything working once again.

We've made a lot of changes here, but only to the structure of the code. Most of the code is the same as before; we've just moved it around. To recap, we've done the following:

---

20. https://github.com/spbooks/phpmysql7/tree/CMS-EntryPoint-Namespaces

- split our code up into classes, recognizing the code that's specific to the joke website and the code that can be used on any future website
- organized all our classes in either the `Ijdb` directory, for project-specific files, or the `Ninja` directory for our framework files
- given all our classes namespaces
- removed all include statements for classes by implementing a PSR-4–compatible autoloader

Composer

Most modern PHP applications use a tool called Composer to handle all autoloading. It's also used to quickly and easily download and install third-party libraries. It's beyond the scope of this book, but if you follow the PSR-4 convention, your classes are good to go when you want to start using it, and you can use Composer's autoloader as a drop-in replacement for the `autoload.php` we just wrote.

When you do start using Composer, just add this code to your `composer.json` file:

```
{
  "autoload": {
    "psr-4": {
        "Ninja\": "Ninja",
        "Ijdb\": "Ijdb"
    }
  }
}
```

For an overview of Composer, take a look at the SitePoint article "Re-introducing Composer, the Cornerstone of Modern PHP Apps"[21].

And the REST

The current iteration of our router uses a very simplistic approach. Each route is a string from the URL that maps to a controller and calls a specific action.

If we continue using this approach, we'll quickly find ourselves repeating logic in the controllers.

Our edit joke form contains the following logic: *if the form is submitted, process the submission, otherwise display the form*.

This logic will be required for any form on the website. Similarly, we can envisage other features in the future requiring similar logic in the controller. For example: *display the page if the user is logged in, otherwise display the login form*.

When the edit joke form is submitted, it uses the `POST` method. Any other request to pages on the website will use the `GET` method.

PHP can detect whether the page was requested using `GET` or `POST`. The variable `$_SERVER['REQUEST_METHOD']` is created by PHP and will contain either the string `GET` or the string `POST`, depending on how the page was requested by the browser. You've already seen that `GET` and `POST` can be defined as the `action` of a form, but just navigating to a page in the browser is also a `GET` request.

We can use this to determine if the form was submitted, and call a different controller action if the form was submitted in the `EntryPoint` class:

```
//...

$controllerName = array_shift($route);
$action = array_shift($route);
```

---

21. https://www.sitepoint.com/re-introducing-composer/

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $action .= 'Submit';
}

$controller = $this->website->getController($controllerName);

$page = $controller->$action(...$route);

//...
```

This approach is very simple: if the `REQUEST_METHOD` is `POST`, it appends the word `submit` to the `$action` variable. That is, if you're on `/joke/edit` and the request type is `POST`, the `$action` variable will be set to `editSubmit` and the method ending with `Submit` will be called in the controller.

Let's split the `edit` method into two different methods—`edit` to display the form, and `editSubmit` to handle the form submission:

```
public function editSubmit() {
    $joke = $_POST['joke'];
    $joke['jokedate'] = new \DateTime();
    $joke['authorId'] = 1;

    $this->jokesTable->save($joke);

    header('location: /joke/list');
}

public function edit($id = null) {
    if (isset($id)) {
        $joke = $this->jokesTable->find('id', $id)[0] ?? null;
    }

    $title = 'Edit joke';

    return ['template' => 'editjoke.html.php',
        'title' => $title,
        'variables' => [
            'joke' => $joke ?? null
        ]
```

```
    ];
}
```

We haven't actually changed the way the website functions, as the user has already navigated to `joke/edit` when the form is displayed and when the form is submitted. This approach of having the same URL perform different actions depending on the request method is loosely known as **representational state transfer** (REST). While REST originally had a slightly stricter meaning, more recently it's become a fuzzy buzzword. You'll see it on job listings, but all most people mean by REST these days is having the same URL respond differently depending on the request method.

Remember, our `delete` method uses `$_POST` so you'll need to rename that `deleteSubmit` as well.

REST Methods

Although REST typically supports the methods `PUT` and `DELETE`, along with `GET` and `POST`, because web browsers only support `GET` and `POST`, PHP developers tend to use `POST` in place of both `PUT` and `DELETE` requests. As such, it's not worth examining the differences in this book.

Some PHP developers have found superficial ways of mimicking `PUT` and `DELETE`, but most developers just stick to using `POST` for writing data and `GET` for reading.

For more information on REST, see the SitePoint article "Build a REST API from Scratch: An Introduction"[22].

When creating a form from now on, we'll have two methods: one for displaying the form, and a `submit` version of the method that handles the form submission.

22. https://www.sitepoint.com/build-rest-api-scratch-introduction/

Non-web Applications

Our `EntryPoint` class now looks like this:

```
class EntryPoint {
    public function __construct(private $website) {
    }

    public function run($uri) {
        try {
            $this->checkUri($uri);

            if ($uri == '') {
                $uri = $this->website->getDefaultRoute();
            }

            $route = explode('/', $uri);

            $controllerName = array_shift($route);
            $action = array_shift($route);

            if ($_SERVER['REQUEST_METHOD'] === 'POST') {
                $action .= 'Submit';
            }

            $controller = $this->website->getController($controllerName);

            $page = $controller->$action(...$route);

            $title = $page['title'];

            $variables = $page['variables'] ?? [];
            $output = $this->loadTemplate($page['template'], $variables);

        } catch (\PDOException $e) {
            $title = 'An error has occurred';

            $output = 'Database error: ' . $e->getMessage() . ' in ' .
            $e->getFile() . ':' . $e->getLine();
        }

        include  __DIR__ . '/../templates/layout.html.php';
```

```
    }

    private function loadTemplate($templateFileName, $variables) {
        extract($variables);

        ob_start();
        include  __DIR__ . '/../templates/' . $templateFileName;

        return ob_get_clean();
    }

    private function checkUri($uri) {
        if ($uri != strtolower($uri)) {
            http_response_code(301);
            header('location: ' . strtolower($uri));
        }
    }
}
```

The problem with this approach is that it's not very flexible. If we ever want to use the `EntryPoint` class in an application that isn't web based, it won't work, because the `$_SERVER` variable won't be set. Like we did with the `uri` variable, it's better to pass this in as an argument:

```
public function run($uri, $method) {
        try {
            $this->checkUri($uri);

            if ($uri == '') {
                $uri = $this->website->getDefaultRoute();
            }

            $route = explode('/', $uri);

            $controllerName = array_shift($route);
            $action = array_shift($route);

            if ($method === 'POST') {
                $action .= 'Submit';
            }
```

We then provide it in `index.php`:

```
include __DIR__ . '/../includes/autoload.php';

$uri = strtok(ltrim($_SERVER['REQUEST_URI'], '/'), '?');

$jokeWebsite = new \Ijdb\JokeWebsite();
$entryPoint = new \Ninja\EntryPoint($jokeWebsite);
$entryPoint->run($uri, $_SERVER['REQUEST_METHOD']);
```

If we wanted to, we could use `EntryPoint` in a command-line application by providing `REQUEST_URI` and `REQUEST_METHOD` from a different location.

Avoiding hardcoding like this is a very good habit to get into. The trend in PHP (and software development in general) is towards **test-driven development** (TDD), and hardcoded values like `$_SERVER['REQUEST_METHOD']` make testing difficult. Although TDD is well beyond the scope of this book, I do want to teach you practices that will make your eventual move to TDD as easy as possible.

Enforcing Dependency Structure with Interfaces

In Chapter 8, when we created the `DatabaseTable` class, we wrote the constructor so that it would check the types of its arguments:

```
public function __construct(\PDO $pdo, string $table, string $primaryKey) {
```

Using this approach, it's impossible to construct an instance of the `DatabaseTable` class without supplying an instance of `PDO` as the first argument.

The `EntryPoint` class has a dependency on `JokeWebsite` provided as the `$website` class variable. The `EntryPoint`'s `run` method calls the `getDefaultRoute()` and `getController()` methods:

```
if ($uri == '') {
    $uri = $this->website->getDefaultRoute();
}
```

```
//...
$controller = $this->website->getController($controllerName);
```

However, what happens if the `$this->website` variable isn't an instance of `JokeWebsite`, or it's an object that doesn't have a `getController` method?

The `run` method takes a `$uri` and a `$method` variable. What happens if these aren't set to strings?

As we did with `DatabaseTable`, we can enforce the types using hints when defining the arguments:

```
namespace Ninja;
class EntryPoint {
    public function __construct(private \Ijdb\JokeWebsite $website) {
    }

    public function run(string $uri, string $method) {
```

With the type hints in place, it's impossible to construct the `EntryPoint` class without supplying an instance of `\Ijdb\JokeWebsite` as the constructor argument `$website`. But this breaks our flexibility! What happens when we build the online shop and we want to use a class called `\Shop\Routes`? Ideally, we want the flexibility of allowing each website to supply a class for the `$website` variable, but we also want the robustness that type checking gives us.

This can be achieved using something called an "interface". An **interface** can be used to describe what methods a class should contain, but it doesn't contain any actual logic. Classes can then *implement* the interface.

An interface for the routes would look like this:

```
<?php
namespace Ninja;
interface Website {
```

```
    public function getDefaultRoute();
    public function getController(string $controllerName);
}
```

You'll notice that it looks a little like a class. It has a namespace, a name, and a method. However, the difference between an interface and a class is that an interface only contains the method **header** (the first line). It doesn't contain any logic, and there are no braces (`{` and `}`) after the methods.

Let's save the interface in the `Ninja` directory as `Website.php`. Like classes, interface files can be loaded by the autoloader.

We can now type hint the *interface* in `EntryPoint`:

```
public function __construct(private \Ninja\Website $website) {
}
```

Currently, this will prevent us from passing an instance of `Ijdb\JokeWebsite` into `EntryPoint`'s constructor. However, we can make `Idjb\JokeWebsite` *implement* the interface.

Example: CMS-EntryPoint-Interface[23]

```
<?php
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
        //...
```

This has two effects:

- The class `Ijdb\JokeWebsite` *must* contain the methods described in the interface. If not, an error is displayed.
- The `Ijdb\JokeWebsite` class can now be type hinted using the interface.

---

23. https://github.com/spbooks/phpmysql7/tree/CMS-EntryPoint-Interface

Now, when we build the online shop or any other website, we can make a new version of the routes class by implementing the interface:

```
namespace Shop;
class JokeWebsite implements \Ninja\Website {
  public function getController(string $controllerName) {
        //...
    }

    public function getDefaultRoute() {
        //...
    }
}
```

Interfaces are very useful for the kind of generic framework we've built. By providing a set of interfaces, each website can provide classes that implement the interfaces, guaranteeing that the framework code and project-specific code fit together.

You can connect your TV to a Blu-ray player, a satellite TV receiver, a games console, or even a computer, because they all use HDMI. The makers of the TV don't know what's going to be connected to it, but anything that follows the HDMI standard will work with the TV. Similarly, anything that uses the `Ninja\Website` interface will work with the `Ninja` framework we've created.

Interfaces, when used correctly, are a very powerful tool for bridging framework and project-specific code.

There's one small gap to fill in our code above. Any class that implements the `Ninja\Website` interface must have the `getController` and `getDefaultRoute` function, but what if those functions didn't do anything?

Take a look at this code:

```
$controller = $this->website->getController($controllerName);

$page = $controller->$action(...$route);
```

Because we've used type hinting, when this code is reached it's guaranteed that the `$this->website` variable will contain a class that implements `Ninja\Website`. For the code to reach this line, `$this->website` must be an object, and that object must contain a method called `getController()`.

But what happens if the `getController` method returns a string? What if it returns nothing at all?

It's possible that, when `$controller->$action(...$route);` runs, the `$controller` variable will actually be a string.

When declaring methods—in either interfaces or classes—you can add a **return type hint**, which specifies what type of value must be *returned* from the function.

Let's look at a very simple example:

```
function add(int $num1, int $num2): int {
    $result = $num1 + $num2;
    return $result;
}
```

After the argument list, there's a return type hint—`: int`. You can specify any type here, and it tells PHP that the return value must be of this type. If the `$result` variable were a string, object or anything but `int`, the script would halt and an error would be displayed.

Like the methods and arguments, this return type can be enforced in an interface:

```
namespace Ninja;

interface Website {
    public function getDefaultRoute(): string;
    public function getController(string $controllerName): object;
}
```

By specifying the return type in the interface, any class that implements the interface must also specify the return type.

Consider when this code runs:

```
$controller = $this->website->getController($controllerName);

$page = $controller->$action(...$route);
```

The `$controller` variable can't be anything but an `object`, because if `getController()` were to return anything other than an `object`, the script would halt.

As the methods in a class must follow the interface, the `Ijdb\JokeWebsite` class must be updated to also supply the return type hints. The complete `Ijdb\JokeWebsite` class now looks like the code below.

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
    public function getDefaultRoute(): string {
        return 'joke/home';
    }

    public function getController(string $controllerName): object {
        $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ↪'ijdbuser', 'mypassword');

        $jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
        $authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id');

        if ($controllerName === 'joke') {
            $controller = new \Ijdb\Controllers\Joke($jokesTable, $authorsTable);
        }
        else if ($controllerName === 'author') {
            $controller = new \Ijdb\Controllers\Author($authorsTable);
        }

        return $controller;
    }
```

```
}
```

Error Handling

Before we finish this chapter, there's one last loose end to tie up. What happens when someone visits a URL that doesn't have a corresponding controller (such as `/doesnt/exist`)? Or when someone tries to run an action on a controller that exists but which has no corresponding method in the controller (such as `/joke/search`)?

If someone specifies a controller that doesn't exist, it will actually cause an error in the `getController` method in the `Website` class. Currently, the method header says that it will always return an object:

```
public function getController(string $controllerName): object {
```

But if you look through the code carefully, it's possible that the returned variable `$controller` never gets set. Take a look at the method as it is currently:

```
if ($controllerName === 'joke') {
    $controller = new \Ijdb\Controllers\Joke($jokesTable, $authorsTable);
}
else if ($controllerName === 'author') {
    $controller = new \Ijdb\Controllers\Author($authorsTable);
}

return $controller;
```

If someone were to visit `/doesnt/exist`, the `$controllerName` variable would be set to `doesnt`, and the `$controller` variable would never be set. We could add an else block that looks something like this:

```
if ($controllerName === 'joke') {
    $controller = new \Ijdb\Controllers\Joke($jokesTable, $authorsTable);
```

```
}
else if ($controllerName === 'author') {
    $controller = new \Ijdb\Controllers\Author($authorsTable);
}
else {
    $controller = new \Ijdb\Controllers\Error();
}
```

But this would require creating a whole controller just for this error message page. Instead, let's have it set the `$controller` variable to `null` if there's no controller for the specified `$controllerName`:

```
if ($controllerName === 'joke') {
    $controller = new \Ijdb\Controllers\Joke($jokesTable, $authorsTable);
}
else if ($controllerName === 'author') {
    $controller = new \Ijdb\Controllers\Author($authorsTable);
}
else {
    $controller = null;
}
```

If you test this, it will cause an error, because the method header contains `: object`, specifying that this method *must* return an object. If `$controller` is set to null, the statement `return $controller` will return `null` and `null` is not an object.

This is actually a very common problem in programming. Imagine a function that searches for something: it's always going to be possible that whatever you're searching for doesn't exist.

PHP allows us to easily handle cases like this by prefixing the return type with a question mark, like so:

```
public function getController(string $controllerName): ?object {
```

This says the function must return either an `object` or `null`, which lets us handle the situation above.

Now, in the `EntryPoint` class, we can check if there's an actual object in the `$controller` variable.

Rather than just checking to see if there's an object (and, in case you're wondering, the `is_object()` function can do this,), we can check to see if the controller and action contain a method that can be called.

The inbuilt function `is_callable` is used to determine whether a variable contains a reference to something that can be called. For objects, an array can be specified. For example:

```
$databaseTable = new DatabaseTable();

//check if $databaseTable->save() is a real function that can be called
var_dump(is_callable([$databaseTable, 'save'])) //true


//check if $databaseTable->otherfunction() is a real function that can be called
var_dump(is_callable([$databaseTable, 'otherfunction'])) //false
```

And this works even if `$databaseTable` is null or contains something that isn't a function. We can make use of this to display an error if the controller or action don't exist:

```
$controller = $this->website->getController($controllerName);

if (is_callable([$controller, $action])) {
     $page = $controller->$action(...$route);

    $title = $page['title'];

    $variables = $page['variables'] ?? [];
    $output = $this->loadTemplate($page['template'], $variables);
}
else {
    http_response_code(404);
    $title = 'Not found';
    $output = 'Sorry, the page you are looking for could not be found.';
}
```

If either `$controller` is null or `$action` is set to a method that doesn't exist, the `else` block will be triggered and the page will display an error. I've also set a `404` (not found) response code so that browsers know not to store the page in the history and search engines don't index the page.

The complete `EntryPoint` class now looks like this:

```
class EntryPoint {
    public function __construct(private \Ninja\Website $website) {
    }

     public function run(string $uri, string $method) {
        try {
            $this->checkUri($uri, $method);

            if ($uri == '') {
                $uri = $this->website->getDefaultRoute();
            }

            $route = explode('/', $uri);

            $controllerName = array_shift($route);
            $action = array_shift($route);

            if ($method === 'POST') {
                $action .= 'Submit';
            }

            $controller = $this->website->getController($controllerName);

            if (is_callable([$controller, $action])) {
                 $page = $controller->$action(...$route);

                $title = $page['title'];

                $variables = $page['variables'] ?? [];
                $output = $this->loadTemplate($page['template'], $variables);
            }
            else {
                http_response_code(404);
                $title = 'Not found';
```

```
                $output = 'Sorry, the page you are looking for could not be
                ↪found.';
            }

        } catch (\PDOException $e) {
            $title = 'An error has occurred';

            $output = 'Database error: ' . $e->getMessage() . ' in ' .
            $e->getFile() . ':' . $e->getLine();
        }

        include  __DIR__ . '/../templates/layout.html.php';
    }

    private function loadTemplate($templateFileName, $variables) {
        extract($variables);

        ob_start();
        include  __DIR__ . '/../templates/' . $templateFileName;

        return ob_get_clean();
    }

    private function checkUri($uri) {
        if ($uri != strtolower($uri)) {
            http_response_code(301);
            header('location: ' . strtolower($uri));
        }
    }
}
```

Example: CMS-EntryPoint-ErrorHandling[24]

Your Own Framework

Writing a framework is a rite of passage for a PHP developer. Everyone does it, and we've just written one! Through this book, I hope I've helped you avoid some of the common traps developers fall into.

In this chapter you learned:

---

24. https://github.com/spbooks/phpmysql7/tree/CMS-EntryPoint-ErrorHandling

- the difference between framework code and project-specific code
- how to differentiate these two types of code by the use of directory structures and namespaces
- how to write an autoloader
- the basics of interfaces and REST
- how to create user-friendly URLs and how to map them to controller actions

Although we haven't added any *functionality* in this chapter, the knowledge covered here will put you on a very firm footing when working with modern PHP applications and third-party code from fellow developers.

Chapter 10

Allowing Users to Register Accounts

Now that we've done all the hard work of building an extensible framework, it's time to add some new functionality to the website. We're going to make it so that users can register accounts on the website with a view to letting them post jokes themselves.

You should already have the table for `authors` in the database with some data you added via MySQL Workbench. If you don't, you can execute this query to create the table:

```
CREATE TABLE author (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50),
  email VARCHAR(100),
  password VARCHAR(255)
) DEFAULT CHARACTER SET utf8mb4 ENGINE=InnoDB
```

Don't worry about adding any records if you don't have any. We'll be creating a form that lets us add authors from the website.

You might notice that the `password` column has the type `VARCHAR(255)`. I know that 255 characters sounds like a very long password, but there's a reason for making the column so large, which we'll get to later.

The first thing that's needed is the controller code. Let's create a `Author.php` file in the `Ijdb\Controllers` directory, then create the class with the following variables, constructor and a method for loading the registration form. For registering users, the only dependency needed is the `DatabaseTable` object that represents the `authors` table:

```
<?php
namespace Ijdb\Controllers;

use \Ninja\DatabaseTable;

class Author {
    public function __construct(private DatabaseTable $authorsTable) {
    }
```

```
    public function registrationForm() {
        return [
          'template' => 'register.html.php',
            'title' => 'Register an account'
         ];
    }

    public function success() {
        return ['template' => 'registersuccess.html.php',
            'title' => 'Registration Successful'];
    }
}
```

I've added two actions: one for displaying the form, and one for displaying the *Registration Successful* page.

Here's the `register.html.php` template in the `templates` directory:

```
<form action="" method="post">
  <label for="email">Your email address</label>
  <input name="author[email]" id="email" type="text">

  <label for="name">Your name</label>
  <input name="author[name]" id="name" type="text">

  <label for="password">Password</label>
  <input name="author[password]" id="password" type="password">

  <input type="submit" name="submit" value="Register account">
</form>
```

Here's `registersuccess.html.php`:

```
<h2>Registration Successful</h2>

<p>You are now registered on the Internet Joke Database</p>
```

If we visit `https://v.je/author/registrationForm`, we should see the form. To make the form work, we'll need to add the `registrationFormSubmit` method

to handle the `POST` request. `Author.php` is shown below.

Example: Registration-Form[1]

```
public function registrationFormSubmit() {
  $author = $_POST['author'];

  $this->authorsTable->save($author);

  header('Location: /author/success');
}
```

Once the form is submitted, the `EntryPoint` will automatically append `Submit` to the end of the method and call the `registrationFormSubmit` method.

Check if it works by filling out the form, submitting it and selecting all the records from the `author` table using MySQL Workbench. If you don't see the data for the record you just added, double check all your code and try again.

We have a basic registration form working, but there are some problems with it in its current state. We need some control over what's allowed in the database. There are some rules we probably want to enforce on the data before allowing the record to be inserted:

- All fields should actually contain some data, so no blank email or name.
- The email address should be a real email address. For example, `paul@example.org` is allowed, but `abc123` isn't.
- The email address entered must not already belong to an existing account.

These validation rules need to be checked before the data is inserted, but also after the form is submitted. If there's a problem with the submission, it's good practice to show the form to the user again so they can fix their mistakes.

Each one of these validation rules needs to be applied in slightly different

---

1. https://github.com/spbooks/phpmysql7/tree/Registration-Form

ways, but with a similar result. We'll use `if` statements for each check and set a Boolean variable `$valid` to keep track of whether the data is valid or not. For example, to check that each field has a value inside it, we can use a series of `if` statements that set the `$valid` variable to false if one of the fields is empty:

```
public function registrationFormSubmit() {
  $author = $_POST['author'];

  // Assume the data is valid to begin with
  $valid = true;

  // But if any of the fields have been left blank, set $valid to false
  if (empty($author['name'])) {
    $valid = false;
  }

  if (empty($author['email'])) {
    $valid = false;
  }

  if (empty($author['password'])) {
    $valid = false;
  }

  // If $valid is still true, no fields were blank and the data can be added
  if ($valid == true) {
    $this->authorsTable->save($author);

    header('Location: /author/success');
  }
  else {
    // If the data is not valid, show the form again
    return ['template' => 'register.html.php',
          'title' => 'Register an account'];
  }
}
```

Using `empty()`

I've used `empty($author['name'])` instead of `$author['name'] == ''` above, because this will also catch invalid form submissions without causing an error.

It's possible for someone to submit a `POST` request without filling in your form! It's likely they might not supply values (an empty string is still a value!) for some of the form fields. It's better to avoid these kinds of errors than potentially alert malicious users to how the site works.

Try it yourself by submitting the form and leaving one or more fields blank. You should see the blank form again rather than the "Registration Successful" message.

If `$valid` has been set to false because one of the fields is empty, the form is shown again by returning a title and template. If you did try this for yourself, you'll have immediately noticed that whatever you typed into the box was removed, with no indication of what went wrong.

To fix this, let's first create a second array to keep a list of error messages to show to the user:

```
public function registrationFormSubmit() {
  $author = $_POST['author'];

  // Start with an empty array
  $errors = [];

  // But if any of the fields have been left blank, set $valid to false
  if (empty($author['name'])) {
    $errors[] = 'Name cannot be blank';
  }

  if (empty($author['email'])) {
    $errors[] = 'Email cannot be blank';
```

```
  }

  if (empty($author['password'])) {
    $errors[] = 'Password cannot be blank';
  }

  // If the $errors array is still empty, no fields were blank and the
  // data can be added
  if (empty($errors)) {
    $this->authorsTable->save($author);

    header('Location: /author/success');
  }
  else {
    // If the data is not valid, show the form again
    return ['template' => 'register.html.php',
          'title' => 'Register an account'];
  }
}
```

Remember the `[] =` operator for arrays? This will be added to the end of the `$errors` array, so if the user leaves all three fields blank, all three error messages will be stored in the `$errors` array. These errors will need to be shown to the user by displaying them in the template.

Instead of having an `$errors` array and a `$valid` variable, I've removed the `$valid` variable. We can determine if any validation checks failed by checking to see whether the `$errors` array is still empty. If it's empty, we know that no errors occurred.

When we created the joke list page, we supplied the template some variables using the `variables` key in the returned array. The same thing can be done here with the errors:

```
// If the data is not valid, show the form again
return ['template' => 'register.html.php',
  'title' => 'Register an account',
  'variables' => [
    'errors' => $errors
```

```
  ]
];
```

And now the `$errors` variable can be used in `register.html.php`:

```
<?php
if (!empty($errors)) :
    ?>
  <div class="errors">
    <p>Your account could not be created, please check the following:</p>
    <ul>
    <?php
            foreach ($errors as $error) :
                ?>
        <li><?= $error ?></li>
        <?php
            endforeach; ?>
    </ul>
  </div>
<?php
endif;
?>
<form action="" method="post">
  <label for="email">Your email address</label>
  <input name="author[email]" id="email" type="text">

  <label for="name">Your name</label>
  <input name="author[name]" id="name" type="text">

  <label for="password">Password</label>
  <input name="author[password]" id="password" type="password">

  <input type="submit" name="submit" value="Register account">
</form>
```

To make the errors look a bit nicer, let's add the following to `jokes.css`:

```
.errors {
  padding: 1em;
  border: 1px solid red;
  background-color: lightyellow;
```

```
  color: red;
  margin-bottom: 1em;
  overflow: auto;
}
.errors ul {
  margin-left: 1em;
}
```

If there are any errors now, they'll all be printed in a list at the top of the page before the form, and the user will know what went wrong.

To make things even easier, we'll re-fill the form with the data from `$_POST` so that the user doesn't have to retype everything into each box.

Firstly, let's supply the `$author` information to the template by amending the `return` value:

```
return ['template' => 'register.html.php',
  'title' => 'Register an account',
  'variables' => [
    'errors' => $errors,
    'author' => $author
  ]
];
```

Now print the values in the form fields if they're set. This is exactly the same approach we took in `editjoke.html.php` to fill the form with the information in the database.

Example: Registration-Validation[2]

```
<label for="email">Your email address</label>
<input name="author[email]" id="email" type="text" value="<?=$author['email'] ??
➥''?>">

<label for="name">Your name</label>
```

2. https://github.com/spbooks/phpmysql7/tree/Registration-Validation

```
<input name="author[name]" id="name" type="text" value="<?=$author['name'] ??
↳''?>">

<label for="password">Password</label>
<input name="author[password]" id="password" type="password" value="<?=$author
↳['password'] ?? ''?>">

<input type="submit" name="submit" value="Register account">
```

Validating Email Addresses

The validation above will prevent someone from leaving the email address field blank. However, it doesn't guarantee they've entered a valid email address. They can still enter "a" into the field and it will pass the validation.

To ensure they enter a valid email address, we need to do some checking. We could look at each character in the string and look for an "@" symbol, ensuring it's not the first character, and perhaps also look for a "." after the "@" to match "x@x.x".

As for most common problems, PHP includes a method of validating email addresses that's far more accurate and simpler to use than building your own. There's no need to reinvent the wheel.

To check an email address in PHP, you can use the `filter_var` function like so:

```
$email = 'tom@example.org';

if (filter_var($email, FILTER_VALIDATE_EMAIL) == false) {
  echo 'Valid  email address';
}
else {
  echo 'Invalid email address';
}
```

The `filter_var` function is provided by PHP and takes two arguments. The

first is the string to validate, and the second is the type of data to check against. There are several options, including `FILTER_VALIDATE_URL` and `FILTER_VALIDATE_INT`, for checking whether a given string is a valid URL or integer. The only one we need at this moment is `FILTER_VALIDATE_EMAIL`, which is used to validate email addresses. For a complete list of all the options supported by `filter_var`, see the function's page on the PHP website[3].

Let's implement this check in the `Author` controller.

Example: Registration-Validation-Email[4]

```
if (empty($author['email'])) {
  $valid = false;
  $errors[] = 'Email cannot be blank';
}
else if (filter_var($author['email']) == false) {
  $valid = false;
  $errors[] = 'Invalid email address';
}
```

First, we check whether the email address has been provided, and if it has, whether or not it's a valid email address using `filter_var`. If both checks pass, the email address is valid and no errors are displayed.

Preventing the Same Person from Registering Twice

There's one other check we need to perform on the email address. We need to make sure the same person can't have multiple accounts. Allowing someone to have multiple accounts can cause problems with a website. If they log in, which account do they log in to? If they can see their previous jokes, they'll only see the ones posted by the account they're logged in to, and any information tailored to them will only be displayed on one account.

---

3. http://php.net/manual/en/function.filter-var.php
4. https://github.com/spbooks/phpmysql7/tree/Registration-Validation-Email

It's very good practice to prevent the same person from registering twice with the same email address. This can be enforced in the database, but it's more consistent to use PHP to check this. We already have the `$authorsTable` object for searching for records in the `author` database table. We can make use of it to check if an email address already exists.

Let's make use of the existing `find` method to determine whether an email address already exists in the database:

```
if (count($this->authorsTable->find('email', $author['email'])) > 0) {
  $errors[] = 'That email address is already registered';
}
```

The `count` function can be used to tally the number of records returned by the `find` method. If it's greater than zero (`> 0`), there's already a record in the system with the email address being searched for, and you can display an error accordingly.

Securely Storing Passwords

Now that the validation has been added, it's possible for anyone who enters valid data into the form to sign up and have their information added to the database. Go ahead and add some test users and check that it's working correctly by verifying that the record has been added to the `author` table.

Using the approach above, if someone types "mypassword123" into the password field, that's what will be stored in the database. We might think that's not a problem, since only we have access to the database, and we're not going misuse the information. But would we really want the developers of every website we use to know our password?

If our website gets hacked, it's then possible for the hacker to see all of our users' passwords, and because people are forgetful, they tend to use the same password for every website they visit.

Someone with access to someone else's email/password combination could,

therefore, do quite a bit of damage beyond accessing the account on this one site we just built—such as reading their email or accessing their PayPal account.

The Importance of Unique Passwords

This is why security experts recommend using a unique password on every website you register on. Tools such as BitWarden[5] and LastPass[6] manage this for you and greatly increase your password security.

A good website developer will help protect their users from this kind of attack. The most common method of achieving this is using a "one-way hashing function".

A **hashing function** takes a string like `mypassword123` and converts it to an encrypted version of the string, known as a **hash**. For example, `mypassword123` would be *hashed* to produce a seemingly random string of numbers and letters such as `9c87baa223f464954940f859bcf2e233`.

To convert a string to a hash, we can use one of several available hashing functions built into PHP, including `md5` and `sha1`. Using these functions is simple:

```
echo md5('mypassword123'); // prints 9c87baa223f464954940f859bcf2e233
```

This isn't "encryption" in the true sense. There's no way to decrypt that seemingly random string of letters and numbers back into `mypassword123`.

One method of storing passwords is to store these hashes in the database. When a user types "mypassword123" into the password field, `9c87baa223f464954940f859bcf2e233` is stored in the password column in the

5. https://bitwarden.com/
6. https://www.lastpass.com/

database instead of `mypassword123`.

Now, if someone does manage to gain access to the database, all they'll see is a list of names and hashes. For example:

```
Kevin   9c87baa223f464954940f859bcf2e233
Laura   47bce5c74f589f4867dbd57e9ca9f808
Tom     9c87baa223f464954940f859bcf2e233
Jane    8d6e8d4897a32c5d011a89346477fb07
```

This solves the problem of someone with access to the database being able to read everyone's password. However, it's not perfect. What do you know about Kevin and Tom's passwords? Looking at the list, you can see that they're the same! If you can work out Kevin's password, you'll also know Tom's.

And, what's worse, we actually know what the password is, because we already discovered that `9c87baa223f464954940f859bcf2e233` is the hash for `mypassword123`. Because people all use the same common passwords, hackers will generate the hashes for common passwords in order to quickly work out which users are using them. Once you know that the hash for `password` is `5f4dcc3b5aa765d61d8327deb882cf99`, you can query the database for any user who has that hash and you'll know their password is `password`. Do this for the top 100, 200 or 1000 passwords, and on a large site you'll work out dozens of real email/password combinations.

Worse still, md5 and sha1 hashes can be generated incredibly quickly. A hacker can create a database of hashes for every possible password up to 16 characters long in just a few days. They can then look up any hash in their database and work out any password with little effort.

There are several methods for solving this problem with generated hashes, but there's a lot to consider, and making a truly secure password hash is more difficult than it seems. If you want to learn more about the theory for solving this, take a look at the SitePoint article "Hashing Passwords with the PHP 5.5 Password Hashing API"[7].

Luckily for us, PHP includes a very secure way of hashing passwords. It's created by people who know a lot more about this stuff than you or I, and it avoids developers like us needing to fully understand the security problems that can occur. For this reason, it's strongly recommended that we use the inbuilt PHP algorithm for hashing passwords rather than create our own.

Now that you understand the importance and the theory behind password hashing, let's put it into practice.

PHP contains two functions, `password_hash` and `password_verify`. For now, we're only interested in `password_hash`. We'll use `password_verify` in the next chapter when we're checking to see whether someone entered the correct username and password when logging in.

This is how we can hash a password using the `password_hash` function:

```
$hash = password_hash($password, PASSWORD_DEFAULT);
```

`$password` stores the text of the password being hashed, and `PASSWORD_DEFAULT` is the algorithm to use. It's generally best to leave the choice up to PHP, as this will choose the best algorithm currently available. (At the time of writing, this is an algorithm known as Bcrypt[8], but it may change over time.)

If we run the code above with a password such as "mypassword123" and echo the `$hash` variable, we'll see something like this:

```
$2y$10$XPtbphrRABcV95GxoeAk.OeI8tPgaypkKicBUhX/YbC9QYSSoowRq
```

I say "something like", because each time you run the function, you'll get a different result. Even if you use "mypassword123" as the password each time, you'll get a different hash as a result. If two people have the same password, different hashes will be stored in the database.

---

7. https://www.sitepoint.com/hashing-passwords-php-5-5-password-hashing-api/
8. https://en.wikipedia.org/wiki/Bcrypt

Earlier in the chapter, when we added the password column to the database table, I said to make it 255 characters. This is because hashes can be long, and if the default algorithm changes, they may grow in size.

Let's implement the `password_hash` function in the registration form. It's surprisingly easy:

```
// …
if (empty($errors)) {
  // Hash the password before saving it in the database
  $author['password'] = password_hash($author['password'], PASSWORD_DEFAULT);

  // When submitted, the $author variable now contains a lowercase value for
  // email and a hashed password
  $this->authorsTable->save($author);

  header('Location: /author/success');
}
// …
```

The `password` value in the `$author` array is replaced with the hashed version. Now, when the data is saved, the hashed password is stored in the database, instead of the value `mypassword123` (or whatever was entered into the form).

Registration Complete

The final version of the `Author` controller is shown below.

Example: Registration-Validation-Email2[9]

```
<?php
namespace Ijdb\Controllers;

use \Ninja\DatabaseTable;

class Author {
```

9. https://github.com/spbooks/phpmysql7/tree/Registration-Validation-Email2

```
public function __construct(private DatabaseTable $authorsTable) {
}

public function registrationForm() {
    return [
      'template' => 'register.html.php',
        'title' => 'Register an account'
    ];
}

public function success() {
      return [
        'template' => 'registersuccess.html.php',
        'title' => 'Registration Successful'
      ];
}

public function registrationFormSubmit() {
    $author = $_POST['author'];

// Start with an empty array
    $errors = [];

    // But if any of the fields have been left blank, write an error to the
    // array
    if (empty($author['name'])) {
        $errors[] = 'Name cannot be blank';
    }

    if (empty($author['email'])) {
        $errors[] = 'Email cannot be blank';
    } else if (filter_var($author['email'], FILTER_VALIDATE_EMAIL)
    ↪== false) {
        $errors[] = 'Invalid email address';
    } else { // If the email is not blank and valid:
        // convert the email to lowercase
        $author['email'] = strtolower($author['email']);

        // Search for the lowercase version of $author['email']
        if (count($this->authorsTable->find('email', $author['email']))
        ↪> 0) {
            $errors[] = 'That email address is already registered';
```

```
            }
        }

        if (empty($author['password'])) {
            $errors[] = 'Password cannot be blank';
        }

        // If there are no errors, proceed with saving the record in the database
        if (count($errors) === 0) {
            // Hash the password before saving it in the database
            $author['password'] = password_hash($author['password'],
            ↪PASSWORD_DEFAULT);

            // When submitted, the $author variable now contains a lowercase
            // value for email and a hashed password
            $this->authorsTable->save($author);

            header('Location: /author/success');
        } else {
            // If the data is not valid, show the form again
            return ['template' => 'register.html.php',
        'title' => 'Register an account',
        'variables' => [
            'errors' => $errors,
            'author' => $author
        ]
      ];
        }
    }
}
```

Chapter Summary

In this chapter, I showed you how to add a new controller to the website, allow users to sign up for accounts, validate the data that's entered into a form, and how to store passwords securely.

In the next chapter, we'll build a login form, using a feature called "sessions" to track whether a user is logged in or not.

Chapter 11

Cookies, Sessions, and Access Control

In the last chapter, I showed you how users can register accounts on the website. Now it's time to make those accounts functional, so that users can *log in* to the website. The process is familiar to web users: they enter a username and password, and get access to content that's unique to their account.

Although a familiar process from the perspective of someone using the website, for a developer, building a website that allows *logging in* (or *user authentication*) can seem daunting at first.

By its nature, HTTP is **stateless**. You connect to a website, and the server gives you a file. As you've already seen, you can send data from the browser to the server using `GET` variables and HTML forms. However, the information is provided to a single page, and is only available when the browser provides `GET` (or `POST`) variables.

For a login system, the user will need to send their username and password to the server once, and then maintain a "logged-in" state on every subsequent page request.

Although this information could be sent via URL parameters or HTML forms, using `GET` or `POST` alone would require the browser to send the username and password to every single page. From a user's point of view, entering their username and password each time they visited a different page on the website would be a repetitive and frustrating task.

Two technologies, *cookies* and *sessions*, can be used to store information about a particular user between pages.

Cookies and sessions are two of those mysterious technologies that are almost always made out to be more intimidating and complex than they really are. In this chapter, I'll debunk those myths by explaining in simple language what they are, how they work, and what they can do for you. I'll also provide practical examples to demonstrate each.

Finally, we'll use these new tools to give our newly registered users the ability

to navigate around the website and post jokes associated with their account.

Cookies

Most computer programs these days preserve some form of **state**, whether it be the position of the application window, or the names of the last five files you worked with. The values are usually stored in a small file on your system so they can be read back the next time the program is run. When web developers took web design to the next level—moving from static pages to complete, interactive online applications—there was a need for similar functionality in web browsers. And thus, cookies were born.

A **cookie** is sent by a website to a user's computer. Importantly, on any subsequent visits to the website, the browser sends the cookie back to the website. The cookie allows the website to store some information on the user's computer. And that information can be different for each browser—for example, the username of the person who logged in with that particular browser.

A simplified example of a login process looks like this:

1. The user enters their username `tom` and password `secret`.
2. If correct, the website creates a cookie on the user's computer that stores the login name `tom`.
3. Any time the user views a page on the website, the cookie (containing the login name) is sent back to the website.
4. The website reads the cookie and knows that this user is logged in as `tom`.

When a cookie is created, it's also given an expiration date. When this date arrives, the browser deletes the cookie, and the user will have to log in again when they next visit the site. This can be any time in the future. Users can also clear (or delete) their own cookies. The information stored in a cookie is saved

on the person's computer and is only accessible by the website that created it. So, contrary to popular belief, cookies are a relatively safe way to store personal information. Cookies in and of themselves are incapable of compromising a user's privacy.

As useful as cookies are, you shouldn't use them for logging in users on a real website, because anyone can edit the cookies stored on a computer. Someone could change `tom` in the cookie to any other account name and be logged in as any other user they choose without ever entering the password. We'll look at a secure login process later on, but now that you know the basics of how cookies work, let's take a look at a more practical demonstration of how you would go about using them.

Cookies can be useful in cases where having the information changed by a user doesn't create any potential security issues. For example, storing a list of products the user has added to their shopping basket is a nice use for cookies. If the user changes the contents of the cookie, it's not going to affect other users. But because the contents of cookies can be overridden by a user, developers need to keep in mind that cookies may not contain what they expect them to.

In PHP, cookies are created using the following process:

1. First, a web browser requests a URL that corresponds to a PHP script. Within that script is a call to the `setcookie` function that's built into PHP.

2. The page produced by the PHP script is sent back to the browser, along with an HTTP `set-cookie` header that contains the name (for example, `mycookie`) and the value of the cookie to be set.

3. When it receives this HTTP header, the browser creates and stores the specified value as a cookie named `mycookie`.

4. Subsequent page requests to that website contain an HTTP `cookie`

header that sends the name–value pair (`mycookie=value`) to the script requested.

5. Upon receipt of a page request with a `cookie` header, PHP automatically creates an entry in the `$_COOKIE` array with the name of the cookie (`$_COOKIE['mycookie']`) and its value.

In other words, the PHP `setcookie` function lets us set a variable that will automatically be set by subsequent page requests from the same browser. Each browser (or website visitor) can have a different value set in the same cookie.

Before we examine an actual example, let's take a close look at the `setcookie` function[1]:

```
bool setcookie ( string $name [, string $value = "" [, int $expire = 0 [, string
↪$path = "" [, string $domain = "" [, bool $secure = false [, bool $httponly =
↪false ]]]]]] )
```

Square Brackets

The square brackets (`[…]`) shown above indicate arguments of the function that are optional. You can omit these arguments and some defaults will be set by PHP automatically.

Like the `header` function we saw in Chapter 4, the `setcookie` function adds HTTP headers to the page, and thus *must be called before any of the actual page content is sent*. Any attempt to call `setcookie` after page content has been sent to the browser will produce a PHP error message. Typically, therefore, we'll use these functions in our controller script before any actual output is sent (by an included PHP template, for example).

The only required parameter for this function is the `name` parameter, which

1. http://php.net/manual/en/function.setcookie.php

specifies the name of the cookie. Somewhat confusingly, calling `setcookie` with only the name parameter will actually delete the cookie that's stored on the browser, if it exists. The value parameter allows you to create a new cookie, or modify the value stored in an existing one.

By default, cookies will remain stored by the browser, and thus will continue to be sent with page requests until the browser is closed by the user. If you want the cookie to persist beyond the current browser session, you must set the `expiryTime` parameter to specify the number of seconds from January 1, 1970 to the time at which you want the cookie to be deleted automatically. Although that sounds arbitrary, this is a very common time format known as a **unix timestamp**, and PHP has inbuilt functions for calculating this so you don't need to do it yourself.

The current time in this format can be obtained using the PHP `time` function. Thus, a cookie could be set to expire in one hour, for example, by setting `expiryTime` to `time() + 3600`. To delete a cookie that has a preset expiry time, change this expiry time to represent a point in the past (such as one year ago: `time() – 3600 * 24 * 365`). Here are two examples showing these techniques in practice:

```
// Set a cookie to expire in 1 year
setcookie('mycookie', 'somevalue', time() + 3600 * 24 * 365);

// Delete it
setcookie('mycookie', '', time() - 3600 * 24 * 365);
```

UNIX Timestamps

Unix timestamps will change in the future. In the year 2038, they'll suffer a similar problem to the Y2K bug[2], because the data type used to store them can't store a high enough number to count the seconds after January 19, 2038.

This may seem a long way off at the moment, but it's worth keeping in mind. To make a cookie persistent, we must set an expiration date. We might be inclined to make this a date very far in the future so that it effectively never expires—perhaps the current date + 16 years.

Doing this calculation will break if our script executes any time after January 19, 2022, because the expiration date will be after 2038 and the cookie won't be set. Selecting ten years' time will break our program in 2032. I recommend we select one year, so that we're safe until 2037—by which time there'll be a proper fix in place.

The `path` parameter lets us restrict access to the cookie to a given path on our server. For instance, if we set a path of `'/admin/'` for a cookie, only requests for pages in the `admin` directory (and its subdirectories) will include the cookie as part of the request. Note the trailing `/`, which prevents scripts in other directories beginning with `/admin` (such as `/adminfake/`) from accessing the cookie. This is helpful if you're sharing a server with other users, and each user has a web home directory. It allows us to set cookies without exposing our visitors' data to the scripts of other users on our server. On modern websites, this is usually not an issue, and you can generally omit this parameter unless you know exactly what you're doing.

The domain parameter serves a similar purpose: it restricts the cookie's access to a given domain. By default, a cookie will be returned only to the host from which it was originally sent. Large companies, however, commonly have several hostnames for their web presence (for example, `www.example.com` and

---

2. https://en.wikipedia.org/wiki/Year_2000_problem

`support.example.com`). To create a cookie that's accessible by pages on both servers, we would set the domain parameter to `'.example.com'` . Note the leading `.` , which allows anything ending in `.example.com` to access the cookie. However, cookies are never shared across different domains. Setting the domain parameter to `example2.com` won't make the cookie available on another site.

The `secure` parameter, when set to `1` , indicates that the cookie should be sent only with page requests that happen over a secure (SSL) connection (that is, with a URL that starts with `https://`). As a majority of websites use SSL these days—including the development environment you're using for this book—it's a good idea to set this.

You might be thinking like this: "My site redirects all non-HTTPS traffic to HTTPS, so this option is irrelevant, as data is only sent via SSL". However, if someone visits `http://example.com` and is redirected to `https://example.com` , the cookie data is sent over the insecure HTTP connection before the redirect happens. Anyone monitoring the connection could read the cookie data from the first (non-SSL) request. Because of this, it's good practice to set this option when your website is using SSL. And if your website has any kind of login system, it should be using SSL.

The `httpOnly` parameter, when set to `1` , tells the browser to prevent JavaScript code on our site from seeing the cookie that we're setting. Normally, the JavaScript code we include in our site can read the cookies that have been set by the server for the current page. While this can be useful in some cases, it also puts the data stored in our cookies at risk should an attacker figure out a way to inject malicious JavaScript code into our site. This code could then read your users' potentially sensitive cookie data and do unspeakable things with it. If we set `httpOnly` to `1` , the cookie we're setting will be sent to our PHP scripts as usual, but will be invisible to JavaScript code running on our site. Unless you specifically want to allow the cookie data to be read by JavaScript, it's good practice to enable this option.

While all parameters except `name` are optional, when calling the `setcookie`

function in the conventional way, using a comma-separated list of values, we must specify values for earlier parameters if we want to specify values for later ones. For instance, to call `setcookie` with a domain value, we also need to specify a value for the `expiryTime` parameter. To omit parameters that require a value, we can set string parameters (`value` , `path` , `domain`) to `''` (the empty string) and numerical parameters (`expiryTime` , `secure`) to `0` .

A useful feature of PHP 8 is **named arguments**. Rather than calling a function with a comma-separated list of values and having to remember which order the arguments should be placed in, as well as providing values for all arguments up to the one we want, PHP 8 allows us to supply arguments using their name.

The `setcookie` function can be called like this:

```
setcookie(name: 'name', value: 'value', secure: true, httpOnly: true);
```

Using Correct Names

To use named parameters, you must get the correct names for the specific function being called. For inbuilt functions like `setcookie` , you can read the variable names from the corresponding manual page[3].

Let's now look at an example of cookies in use. Imagine we want to display a special welcome message to people on their first visit to our site. We could use a cookie to count the number of times a user has been to our site before, and only display the message when the cookie hasn't been set. The code for this is shown below.

Example: Sessions-Cookie[4]

---

3. https://www.php.net/manual/en/function.setcookie.php
4. https://github.com/spbooks/phpmysql7/tree/Sessions-Cookie

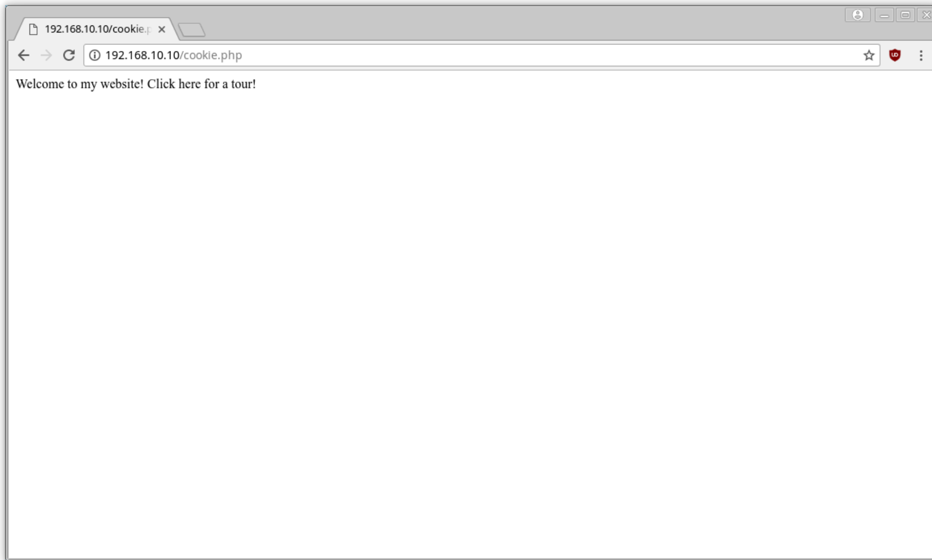
```
<?php

if (!isset($_COOKIE['visits'])) {
    $_COOKIE['visits'] = 0;
}
$visits = $_COOKIE['visits'] + 1;
setcookie(name: 'visits', value: $visits, expires: time() + 3600 * 24 * 365,
↪secure: true, httpOnly: true);

if ($visits > 1) {
    echo "This is visit number $visits.";
} else {
    // First visit
    echo 'Welcome to our website! Click here for a tour!';
}
```

This code starts by checking if `$_COOKIE['visits']` is set. If it isn't, it means the `visits` cookie has yet to be set in the user's browser. To handle this special case, we set `$_COOKIE['visits']` to `0`. The rest of our code can then safely assume that `$_COOKIE['visits']` contains the number of previous visits the user has made to the site.

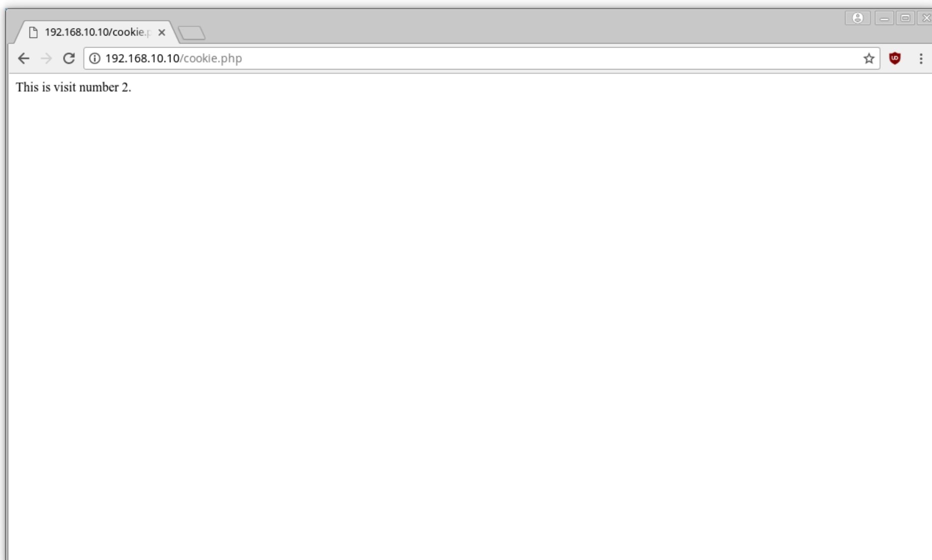
Next, to work out the number of *this* visit, we take `$_COOKIE['visits']` and add the value `1`. This `$visits` variable will be used by our PHP template.

Finally, we use `setcookie` to set the `visits` cookie to reflect the new number of visits. We set this cookie to expire in one year's time.

The images below shows what this example looks like the first time a browser visits the page and after the second visit.



11-1. The first visit



11-2. The second visit

Before we go overboard using cookies, we have to be aware that browsers place a limit on the number and size of cookies allowed per website. Some

browsers will start deleting old cookies to make room for new ones after we've set 20 cookies from our site. Other browsers will allow up to 50 cookies per site, but will *reject* new cookies beyond this limit. Browsers also enforce a maximum combined size for all cookies from all websites, so an especially cookie-heavy site might cause our own site's cookies to be deleted.

Each time someone visits our website, all of the cookies are sent to the web server. If we store a lot of information in the cookie, it can slow down the responsiveness of the website, because extra data must be transferred with each page view.

Cookies can also be read by anyone who gains access to the computer they're stored on, so cookies are only as secure as the computer being used to view the website.

For these reasons, we should do our best to keep the number and size of the cookies our site creates to a minimum.

PHP Sessions

Because of the limitations I've just described, cookies are inappropriate for storing large amounts of information. If we run an ecommerce website that uses cookies to store items in shopping carts as users make their way through our site, it can be a huge problem. The bigger a customer's order, the more likely it will run afoul of a browser's cookie restrictions.

Sessions were developed in PHP as the solution to this issue. Instead of storing all our (possibly large) data as cookies in our visitor's web browser, sessions let us store the data on our web server. The only value that's stored in the browser is a single cookie containing the user's **session ID**—a long string of letters and numbers that serves to identify that user uniquely for the duration of their visit to our site. It's a variable for which PHP watches on subsequent page requests, and uses to load the stored data that's associated with that session.

Sessions actually work using cookies behind the scenes, and the process they

use is as follows:

1. On the first visit, each user is given a random session ID (such as 1234 for this example).
2. A cookie called `PHPSESSID` is created, and it stores the value `1234`.
3. A file called `1234` is created on the server.
4. Any data written to the **session** is stored in the file `1234` on the *server*.
5. Each time the user navigates to a different page, the cookie containing `1234` is sent to the server.
6. When the server receives the session ID from the cookie, it loads the data from the file `1234`.

Sessions can be used to solve the same kind of problems as cookies, and they can even use cookies behind the scenes by default. The major difference between sessions and cookies is that the session data is stored on the server, and only the user's session ID is stored in their browser.

There are lots of configuration options available for sessions. You can change where the session ID is stored and configure the cookie. You can even have PHP store session data in a database instead of files. For almost all websites, the default configuration will suffice. For the rest of this chapter, I'll assume that you're using the default configuration. A full list of configuration options is available in the PHP manual[5].

Because each user is given a unique session ID, the website can store different information for each website user, and it knows which user is which because each user has their own unique identifier.

The session ID is a random string of numbers and letters, such as

---

5. https://www.php.net/manual/en/session.security.ini.php

`69808ef39d7420376d7b74c0447965e1`. This random string is used because it's difficult to guess. If it were just sequential numbers, an attacker could try setting their session cookie to different numbers and see who they're logged in as. Using the default session ID algorithm, there are 340,282,366,920,999,872,040,712,440,960,128,544,160 (16^32) possible session IDs. An attacker would have to spend a long time guessing to find a session ID that was being used on any particular website.

Sessions with GET Variables Instead of Cookies

Sessions can be configured to use `GET` variables instead of cookies, but this is generally a bad idea. When this option is enabled, URLs look something like `https://example.com/page.php?PHPSESSID=69808ef39d7420376d7b74c0447965e1`. This is a problem for a number of reasons:

- The URL is ugly and cluttered.
- Any malicious JavaScript on the page can read the session ID.
- If the user copies the link and shares it on social media, anyone who clicks the link is logged in as the user who copied the link initially!

We're now ready to start working with PHP sessions. Before we jump into an example, let's quickly look at the most common session management functions in PHP. To tell PHP to look for a session ID, or start a new session if none is found, we simply call `session_start`. If an existing session ID is found when this function is called, PHP restores the variables that belong to that session. Since this function attempts to create a cookie, it must come before any page content is sent to the browser, just as we saw for `setcookie` above:

```
session_start();
```

When this function is run, it does several things:

- It generates a session ID if the user doesn't have one and writes it to the `PHPSESSID` cookie.

- It reads the existing session ID if there is one.
- It loads the file that contains the session.
- It creates a special variable called `$_SESSION` that contains the data from the file in the format of a PHP array.

The contents of the `$_SESSION` variable are loaded from a file specified by the user's session ID. Because of this, the `$_SESSION` variable can contain different data for each user of the website. This is a very useful feature of a login system, as it can store the username of the individual user who's logged in.

Creating the `$_SESSION` Variable

Unlike `$_GET` and `$_POST`, which are created automatically, the `$_SESSION` variable doesn't exist until after `session_start()` has been called. If you get an `Undefined variable $_SESSION` error message, ensure that you've called `session_start()` in the script prior to using the `$_SESSION` variable.

To create a session variable that will be available on all pages on the site when accessed by the current user, we set a value in the `$_SESSION` array. For example, the following code will store the variable called `password` in the current session. The `$_SESSION` variable will be empty until we've called `session_start()`, so we need to ensure we don't read or write to it before the session has been started. After the session has been started, we can treat the `$_SESSION` variable like a normal array, reading and writing values to it:

```
$_SESSION['password'] = 'mypassword';
```

To remove a variable from the current session, we can use PHP's `unset` function:

```
unset($_SESSION['password']);
```

Finally, if we want to end the current session and delete all registered variables in the process, clear all the stored values and use `session_destroy`:

```
$_SESSION = [];
session_destroy();
```

For more detailed information on these and the other session-management functions in PHP, see the relevant section of the PHP manual[6].

Now that we have these basic functions under our belt, let's put them to work in a simple example.

Counting Visits with Sessions

I showed you how to use cookies to track the number of times someone had visited the page, and the same thing can be done with sessions.

Example: Sessions-Count[7]

```
<?php
session_start();

if (!isset($_SESSION['visits'])) {
    $_SESSION['visits'] = 0;
}
$_SESSION['visits'] = $_SESSION['visits'] + 1;

if ($_SESSION['visits'] > 1) {
    echo 'This is visit number ' . $_SESSION['visits'];
} else {
    // First visit
    echo 'Welcome to my website! Click here for a tour!';
}
```

`$_SESSION` is used in place of `$_COOKIE`, and you'll notice the code is a little simpler than the code used for cookies.

6. http://www.php.net/session
7. https://github.com/spbooks/phpmysql7/tree/Sessions-Count

For the cookie, we needed to calculate the lifetime and set an expiration time. Sessions are simpler: no expiration time is required, but any data stored in the session is lost when the browser is closed.

Access Control

One of the most common reasons for building a database-driven website is that it allows users to interact with a site from any web browser, anywhere! But in a world where roaming bands of jubilant hackers will fill our site with viruses and spam, we need to stop and think about the security of our website.

At the very least, we'll want to require username and password authentication before a visitor can post anything to the website. In this section, we'll enhance our joke database site to protect sensitive features with username/password-based authentication. In order to control which users can do what, we'll build a sophisticated role-based access control system.

"What does all this have to do with cookies and sessions?" you might wonder. Well, rather than prompting our users for login credentials every time they want to view a confidential page or perform a sensitive action, we can use PHP sessions to hold on to those credentials throughout their visit to our site.

Logging In

In the last chapter, I showed you how users could register accounts on the website and have their password stored securely. The next step is to allow those registered users to log in and post jokes to the website.

Obviously, access control is a feature that will be handy in many different PHP projects and on different pages on the website. Therefore, like the `EntryPoint` and `DatabaseTable` classes, it makes sense to write as much of our access control code as possible as a shared class, so that we can then reuse it throughout the website and in future projects.

The general process for "logging in" consists of the user supplying an email address and password. If the database contains a matching author, it means

the user filled out the login form correctly and we have to log in the user.

But what exactly does “log in the user” mean? There are two approaches to this, both of which involve using PHP sessions:

- We can log in the user by setting a session variable as a “flag” (for example, `$_SESSION['userid'] = $userId`). On future requests, we can just check if this variable is set and use it to read the ID of the logged-in user.

- We can store the supplied email address and password in the session, and then on future requests, we can check if these variables are set. If they are, we can check the values in the session against the values in the database.

The first option will give better performance, since the user’s credentials are only checked once—when the login form is submitted. The second option offers greater security, since the user’s credentials are checked against the database every time a sensitive page is requested.

In general, the more secure option is preferable, since it allows us to remove authors from the site even while they’re logged in. Otherwise, once a user is logged in, they’ll stay logged in for as long as their PHP session remains active. That’s a steep price to pay for a little extra performance.

The theory behind this is simple to implement, but it’s made more difficult because the password field in the database doesn’t store the password in plain text, as typed by the user. Instead, it stores a hashed password like this:

```
$2y$10$XPtbphrRABcV95GxoeAk.OeI8tPgaypkKicBUhX/YbC9QYSSoowRq
```

There’s no way to decrypt the password to do the comparison, but because we used `password_hash` to hash the password, we can use `password_verify` to check it.

`password_verify` takes two arguments: the plain text password to check and the hashed password from the database. It returns true or false depending on

whether or not the password is correct.

To check the password, we'll need the hash from the database before being able to use it with `password_verify`. Luckily, thanks to the `DatabaseTable` class and the existing `$authorsTable` variable, it's easy for us to look up the hashed password of a user by using their email address:

```
$author = $authorsTable->find('email', strtolower($_POST['email']);
```

Once the user's information is stored in `$author`, it's possible to check the password using the `password_verify` function, like so:

```
if (!empty($author) && password_verify($_POST['password'],
↪$author[0]['password'])) {
    // Login successful
}
else {
    // Passwords don't match, an error occurred
}
```

Note the `[0]` after `$author`. That's because the `find` function may return more than one record. We need to specifically read the first record returned.

The first condition in the `if` statement checks to see if an author has been retrieved from the database. If one has, the second condition will check that the password entered by the user matches the one in the database.

The order is important here. PHP will run and (`&&`) conditions left to right, and stop when one evaluates to false. If the order were reversed and the password check were done before `!empty($author)`, an error might occur, because `$author` might not contain an array with a `password` key! By putting the `!empty($author)` check first, we know that something is set in the `password` key for the `$author` array.

Hashing

You may have thought we could use this:

```
if (password_hash($_POST['password'], PASSWORD_DEFAULT) == $author[0]
↪['password']) { ...
```

However, `password_hash` creates a different hash each time it's called, even if it's called with the same password string! We must use `password_verify` to check the password.

Once the user has entered the email address and password, they can be logged in by setting the session variable.

After checking the password was correct using `password_verify`, it's time to write some data to the session. There are various options here. We could just store the user ID or the email address of the person who's been logged in.

However, it's good practice to store both the login name and password in the session and check them both on each page view. That way, if the user is logged in on two different computers and the password is changed, they'll be logged out and required to log back in.

This is a useful security feature for users, since if one of those logged-in locations wasn't really the user—someone having managed to get unauthorized access to their account—the attacker would be logged out as soon as the password was changed. Without storing the password in the session, the attacker could log in once, and as long as the browser was left open, they'd maintain access to the user's account.

One method of achieving this is to store both the email address and password in the session:

```
$_SESSION['email'] = $_POST['email'];
$_SESSION['password'] = $_POST['password'];
```

Then, on each page view, we'd check the information in the session against the database:

```
$author = $authorsTable->find('email', strtolower($_SESSION['email']))[0];

if (!empty($author) && password_verify($_SESSION['password'],
↪$author['password'])) {
  // Display password protected content
}
else {
  // Display an error message and clear the session, logging the user out
}
```

This is theoretically what we want to do. With this approach, if the password is changed in the database, or the author is removed from the database, the user will be logged out.

However, there's an obvious security issue here. Although sessions are stored on the server, if someone did gain access to our web server, they could see the plain text password of any logged-in users, completely forfeiting the benefit of hashing the password in the first place.

To avoid storing the plain text password of the logged-in users in the session, we'll need to adjust the logic slightly.

Instead of storing the plain-text password in the session, it's better to store the password hash from the database in the session. If someone is able to read the session data from the server, they'll only see the hash, not the real password!

To store the hash in the session, we can use the following code:

```
$_SESSION['email'] = $_POST['email'];
$_SESSION['password'] = $author['password'];
```

With the email address and hash stored, we can check the values from the database, and if either the email address or password stored in the database

have changed, the user can be logged out.

On each page, we'll need to run this code:

```
$author = $authorsTable->find('email', strtolower($_SESSION['email']));

if (!empty($author) && $author[0]['password'] === $_SESSION['password']) {
    // Display password protected content
}
else {
    // Display an error message and log the user out
}
```

The code above does three things:

- It searches the database for the user with the email address from the session, which would have been set when the user submitted the login form.

- It checks a record has been retrieved from the database. After all, it's possible the user entered an email address that doesn't actually exist in the database.

- It compares the password in the session to the password that's currently in the database. If it's changed between logging in and viewing this page, the user will be logged out.

As this check will need to be done on every page we want to password protect, let's move it into a class for easy reuse. We'll need three methods:

- a method that's called when the user tries to log in with an email address and password
- a method that will be called on each page to check whether the user is logged in or not (with the check that ensures the password hasn't changed in the database)
- a method for logging out

Since this is something that's going to be useful on any website we build, we'll place it in the `Ninja` framework namespace:

```
<?php
namespace Ninja;
class Authentication {

    public function __construct(private DatabaseTable $users, private string $usernameColumn,
    ➥ $usernameColumn, private string $passwordColumn) {
        session_start();
    }

    public function login(string $username, string $password): bool {
        $user = $this->users->find($this->usernameColumn, strtolower($username));

        if (!empty($user) && password_verify($password, $user[0][$this->
        ➥passwordColumn])) {
            session_regenerate_id();
            $_SESSION['username'] = $username;
            $_SESSION['password'] = $user[0][$this->passwordColumn];
            return true;
        } else {
            return false;
        }
    }

    public function isLoggedIn(): bool {
        if (empty($_SESSION['username'])) {
            return false;
        }

        $user = $this->users->find($this->usernameColumn, strtolower($_SESSION
        ➥['username']));

        if (!empty($user) && $user[0][$this->passwordColumn] === $_SESSION
        ➥['password']) {
            return true;
        } else {
            return false;
        }
    }

    public function logout() {
```

```
        unset($_SESSION['username']);
        unset($_SESSION['password']);
        session_regenerate_id();
    }
}
```

Let's save this as `Authentication.php` in the `classes/Ninja` directory so it can be used later.

This is mostly the same code I already showed you, with a few minor changes. Firstly, take a look at the constructor. There are three variables required for this class:

- a `DatabaseTable` instance, which is configured for the table that stores user accounts
- the name of the column that stores the login names
- the name of the column that stores the password

As this class will be useful on multiple websites, we need to code it so that it can be used in as many situations as possible. Although the column names that store the login names and passwords are `email` and `password` on this website, on another website the login name might be stored under `username` or `customer_login`, or any other name you can think of. The same is true for the password column.

By making these constructor arguments rather than hardcoding them into the class, it's now possible to use this class on any website, regardless of whether the column names in the database are `email` and `password` or something else.

Each time we want to read the password from the database, the class variables are used. The code is slightly more complex— `$user[0][$this->passwordColumn]` instead of `$user[0]['password']` —but the added flexibility of being able to use this on websites that have the password stored in a column with a different name heavily outweighs the small additional complexity in this class.

When the `Authentication` class is created, it starts the session in the constructor. This avoids us needing to manually call `session_start` on each page. As long as the `Authentication` class has been instantiated, a session will have been started. Later, when `login` or `isLoggedIn` are called, the session must have already been started.

Both `login` and `isLoggedIn` return true or false, which we can later call to determine whether the user has entered valid credentials or is already logged in.

There's also an initial check in the `isLoggedIn` method, ensuring there's data in the session. If not, it returns false, as without a session variable `username`, the user isn't logged in.

Finally, I've used type hints and return type hints throughout to ensure that the methods must be called with the correct types (so only strings can be used for usernames and passwords) and the methods must return a Boolean value (true or false).

A final security measure that's worth implementing is changing the session ID after a successful login. Earlier I mentioned that session IDs shouldn't be easily guessable. Otherwise, hackers could pretend to be someone else—an attack commonly known as **session fixation**. All the hacker needs to be able to steal someone else's session is the session ID.

It's good practice to change the session ID after a successful login just in case someone managed to get hold of the session ID before the user logged in. PHP makes this very easy: the single function `session_regenerate_id` does this by picking a new random ID for that user.

This can be placed in the `if` block that runs when the login is successful:

```
if (!empty($user) && password_verify($password, $user[0][$this->
↪passwordColumn])) {
    session_regenerate_id();
```

```
    $_SESSION['username'] = $username;
    $_SESSION['password'] = $user[0][$this->passwordColumn];
//...
```

If you follow the logic through, you may have realized that frequently changing the session ID can increase security. In fact, it would be very secure to change the user's session ID on every page load.

However, doing so causes several practical problems. If someone has different pages open in different tabs, or the website uses a technology called Ajax, they effectively get logged out of one tab when they open another! These problems are worse than the minor security benefit of changing the session ID on every page, and sessions should be regenerated only when a user logs in or changes their username/password.

Protected Pages

Now that the `Authentication` class is complete, it's time to use it on the website. Before creating a login page, let's secure some of the existing pages so they can't be viewed unless as user is logged in (the `isLoggedIn` function returns true).

Currently, we only have the `Joke` controller. The list jokes page should be visible without logging in, but the facility to add, edit or delete a joke should only be available to users who are logged in.

To achieve this, we'll need to determine whether or not a user is logged in. If they are, the page is displayed as normal. If not, an error message is displayed in its place.

We already have the `Authentication` class, which allows us to determine whether or not someone is logged in. We could pass the `$authentication` instance into each controller and add a check to each controller action, like so:

```
public function edit() {
```

```
  if (!$this->authentication->isLoggedIn()) {
    return ['template' => 'error.html',
            'title' => 'You are not authorized to view this page'];
  }
  else {
    // Display the form as normal
  }

  // …
```

We'd also require a relevant `error.html.php` to display an error message such as "You must be logged in to view this page".

Although this approach will work, it will result in repeated code. Every controller action that should only be available to logged-in users would require repeating this same `if` statement. As you already know, if you find yourself repeating very similar code in multiple places, it's usually better to move the code so it can be written once and reused.

In this case, a better approach is to adjust the router to perform the login check and either use the requested route or display an error page.

Open up the `Ijdb\JokeWebsite` class. It currently looks like this:

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
    public function getDefaultRoute(): string {
        return 'joke/home';
    }

    public function getController(string $controllerName): ?object {
        $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ↪'ijdbuser', 'mypassword');

        $jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
        $authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id');

        if ($controllerName === 'joke') {
            $controller = new \Ijdb\Controllers\Joke($jokesTable, $authorsTable);
```

```
        }
        else if ($controllerName === 'author') {
            $controller = new \Ijdb\Controllers\Author($authorsTable);
        }
        else {
            $controller = null;
        }

        return $controller;
    }
}
```

There are several approaches to adding a login check. We could add a method that returns a list of pages that require a login to view. This will need to be in the `Website` class and not the `EntryPoint` class, because the list of URLs will be the same on each website.

Here's an approach that would work but that isn't ideal. We could have the `Website` class provide a list of pages that require a list of URLs only available to logged-in users:

```
class JokeWebsite implements \Ninja\Website {
    public function getSecurePages(): array {
        return ['joke/edit', 'joke/delete'];
    }
```

Then we could have the `EntryPoint` class call this method and apply some logic when the current page is in the list:

```
class EntryPoint {
 public function run(string $uri, string $method) {
        try {
            $this->checkUri($uri, $method);

            if ($uri == '') {
                $uri = $this->website->getDefaultRoute();
            }

            $requiresLogin = $this->website->getSecurePages();
```

```
            $authorsTable = new \Ninja\Database($pdo, 'authors', 'id');
            $authentication = new Authentication($authorsTable, 'email',
            ➥'password');

            if (in_array($uri, $this->website->getSecurePages()) &&
            ➥!$authentication->isLoggedIn()) {
                header('location: /login');
                exit();
            }
```

The code above creates an instance of the `Authentication` class from earlier, then fetches a list of pages that require the user to be logged in to view from the `Website` class, and then uses `in_array()` to check if the current URI is in the list of login-restricted pages. If the URL is in the list of pages that require a login and the user isn't logged in (`$authentication->isLoggedIn()` returns false) then the user is redirected to the login page.

The login check we've added looks like this:

```
$authorsTable = new \Ninja\DatabaseTable($pdo, 'authors', 'id');
$authentication = new Authentication($authorsTable, 'email', 'password');
if (in_array($uri, $this->website->getSecurePages()) &&
➥!$authentication->isLoggedIn()) {
    header('location: /login');
    exit();
}
```

This code has a lot of hardcoded values that are joke site specific. We could add relevant methods to the `Website` class to deal with this:

```
$authentication = $this->website->getAuthentication(); //get the Authentication
➥instance for this particular website from the `website` instance
if (in_array($uri, $this->website->getSecurePages()) && !$authentication->
➥isLoggedIn()) {
    header('location: ' . $this->website->getLoginUrl()); //get the URL of the
    ➥login page for this particular website from the `website` instance
    exit();
}
```

I haven't given you the corresponding methods in the `Website` class, as we're not going to use this approach. However, in this example, the `Authentication` class is configured and provided by the `Website` class. The `Joke` website can provide its configuration of the `Authentication` class (linked to the `author` table with the `email` and `password` column names) and another website could provide an `Authentication` instance with a different configuration.

While this would work, and while it would be generic, it's actually very limiting. Although up to now we've been placing any code that might be repeated on different websites in the `EntryPoint` class, in this case that approach would actually work against us. It's possible that this exact login check would be used on every website, and there are still some problems with this approach:

- It assumes that every website has pages that can only be viewed when logged in.
- It ties the `EntryPoint` class to the `Authentication` class we created earlier.
- It assumes that we should only redirect to the login page if someone tries to access a restricted page while not logged in.

It doesn't have the ability to do any of these alternatives:

- Use different login code written by someone else instead of our `Authentication` class.
- Display an error message instead of redirecting when someone tries to view a restricted page.
- Log attempts to view restricted pages when not logged in.
- Have different permissions for different users so that some users can view some pages and other logged-in users can't.

Instead of trying to put this in the `EntryPoint` class, we'll create a method called `checkLogin` in the `Website` class, which takes a URI and then determines what should happen next. `EntryPoint` will just call the method and provide the URI of the page being viewed:

```
class EntryPoint {
 public function run(string $uri, string $method) {
        try {
            $this->checkUri($uri, $method);

            if ($uri == '') {
                $uri = $this->website->getDefaultRoute();
            }

            $route = explode('/' $uri);

            $controllerName = array_shift($route);
            $action = array_shift($route);

            $this->website->checkLogin($controllerName . '/' . $action);
```

The `Website` class will provide the `checkLogin` method, which takes the `$uri` and determines whether the person viewing it should be able to view the page or not. Because the `$uri` variable may contain arguments like `joke/edit/1` I've reconstructed it using just `$controllerName` and `$action` so it becomes just `controller/action` without any additional arguments.

Before we add the method, we have a potential problem. The `$authorsTable` instance will need to be available in both the `getController` method (where it's currently being created) and the soon-to-be-added `checkLogin` method. To solve this, we'll promote the variable to a class variable and have the objects (and the database connection which is required by them) created in the constructor:

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
    private \Ninja\DatabaseTable $jokesTable;
    private \Ninja\DatabaseTable $authorsTable;

    public function __construct() {
        $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ↪'ijdbuser', 'mypassword');

        $this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
```

```
        $this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id');
    }

    public function getDefaultRoute(): string {
        return 'joke/home';
    }

    public function getController(string $controllerName): ?object {


        if ($controllerName === 'joke') {
            $controller = new \Ijdb\Controllers\Joke($this->jokesTable,
            ➥$this->authorsTable);
        }
        else if ($controllerName === 'author') {
            $controller = new \Ijdb\Controllers\Author($this->authorsTable);
        }
        else {
            $controller = null;
        }

        return $controller;
    }

}
```

By promoting `$authorsTable` and `$jokesTable` to class variables, they'll be available to any method in this class. Because we're not using constructor argument promotion to create the variables, both need to be defined as class variables with the `private` keyword. By convention, class variables are defined at the top of the class. After making these class variables, they need to be accessed via the `$this->` prefix in the `getController` method.

Now that we can access the `$this->authorsTable` object in any method in the class, let's add the new `checkLogin` method, which will take the URI of the current page and determine what happens next:

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
    private \Ninja\DatabaseTable $jokesTable;
```

```
    private \Ninja\DatabaseTable $authorsTable;
    private \Ninja\Authentication $authentication;

    public function __construct() {
        $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ➥'ijdbuser', 'mypassword');

        $this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
        $this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id');
        $this->authentication = new \Ninja\Authentication($this->authorsTable,
        ➥'email', 'password');
    }
    //...
    public function checkLogin(string $uri): ?string {
        $restrictedPages = ['joke/edit', 'joke/delete'];

        if (in_array($uri, $restrictedPages) && !$this->authentication->
        ➥isLoggedIn()) {
            header('location: /login/login');
            exit();
        }

        return $uri;
    }
    //...
```

By placing this logic in the `Website` class, different websites can follow completely different processes. The joke website can redirect users to the `/login` page if they try to view a restricted page while not logged in. Another website could display an error, use a different `Authentication` class, log all attempts to view restricted pages, or even have a more advanced permissions system where different users have different access levels.

You might be wondering why I made the `$authentication` variable a class variable and didn't just create it in the `checkLogin` method. I'm thinking ahead a little: the controller that handles logging in will also need access to this object. By storing it in a class variable, it can be used in the `checkLogin` method and passed to a controller in the `getController` method when we implement the login page a little later on.

It's now up to individual websites to handle login checks, and only a single additional line has been added to the `EntryPoint` class. As the `EntryPoint` expects any `$website` instance to contain a `checkLogin` method, we'll also add it to the `Website` interface to ensure that any website contains this method. If a website doesn't have a login check, it will still need to provide the method, but it will be empty.

Example: Sessions-LoginCheck[8]

```
namespace Ninja;

interface Website {
    public function getDefaultRoute(): string;
    public function getController(string $controllerName): ?object;
    public function checkLogin(string $uri): ?string;
}
```

As there's currently no way of logging in, you can test that this works by trying to edit a joke. If you've made all the necessary changes, you'll get redirected to the login page any time you try to edit or delete a joke.

Creating a Login Form

Now that the login check is in place and we know it works, it's time to build a form for logging in. As it stands, there's no way to add or edit a joke, because there's no facility for logging in.

Let's create a new controller called `Login` to handle the login form and submission. As we'll need to call the `login` method in the `Authentication` class, the `Authentication` class will need to be provided as a constructor argument. We'll also need two methods:

- `login`, to display the login form
- `loginSubmit`, to handle the login process

---

8. https://github.com/spbooks/phpmysql7/tree/Sessions-LoginCheck

These methods will follow a similar structure to the add joke and register forms. The main difference here is that, instead of writing data to the database when the form is submitted, it will use the `login` method of the `Authentication` class to log the user in:

```
namespace Ijdb\Controllers;

class Login {
    public function __construct(private \Ninja\Authentication $authentication) {

    }

    public function login() {
        return ['template' => 'loginform.html.php',
                'title' => 'Log in'
               ];
    }

    public function loginSubmit() {
        $success = $this->authentication->login($_POST['email'],
        ↪$_POST['password']);

        if ($success) {
            return ['template' => 'loginSuccess.html.php',
                    'title' => 'Log In Successful'
                   ];
        }
        else {
            return ['template' => 'loginForm.html.php',
                    'title' => 'Log in',
                    'variables' => [
                        'errorMessage' => true
                    ]
                   ];
        }

    }

    public function logout() {
        $this->authentication->logout();
        header('location: /');
    }
```

```
}
```

The logic here is fairly straightforward. The `login` method displays the login form. When the form is submitted, the `loginSubmit` method will be called by the `EntryPoint` class (the same thing as happens with the registration form and add joke form). The `loginSubmit` method takes the email address and password that were typed into the form and passed into the `login` method. The `logout` method calls the `logout` method from the `authentication` class and redirects the user back to the home page.

Use POST for Logging In

As sensitive information is being sent to the server, be sure to use `POST` for handling logging in instead of `GET`, as we've done above.

If the login is successful (indicated by the `login` method of the `Authentication` class returning true), a success page is displayed. Otherwise, the form is shown again. If the form is shown again, I've set a variable called `errorMessage` to true to flag to the template that it should display an error message.

Speaking of the templates, they're the final piece needed to get this to work. Let's create the `loginForm.html` template in the templates directory:

```
<?php
if (isset($errorMessage)):
    echo '<div class="errors">Sorry, your username and password could not be
    ↪found.</div>';
endif;
?>
<form method="post" action="">
  <label for="email">Your email address</label>
  <input type="text" id="email" name="email">

  <label for="password">Your password</label>
  <input type="password" id="password" name="password">
```

```
  <input type="submit" name="login" value="Log in">
</form>

<p>Don't have an account? <a href="/author/registrationForm">Click here to
➥register</a></p>
```

Let's also add a `loginSuccess.html.php` template to the templates directory:

```
<h2>Log in successful</h2>

<p>You are now logged in, welcome back.</p>
```

Failed Login Messages

You may be tempted to display different messages to the user depending on why their login failed—such as "Invalid email address" when the email address doesn't exist, and "Invalid password" when the email address is registered but the passwords don't match.

Although this helps the user by letting them see what went wrong, it's a privacy breach and widely considered bad practice, as anyone can type in someone else's email address and see if they're registered on your website based on the error message that's shown.

The same is true for "forgotten password" pages. These shouldn't indicate whether an email address is registered on the website either.

Finally, we need to add the controller to `Website.php` so that its methods can be called via the `/login/` URI. As it needs access to the `Authentication` class, we'll pass that in as an argument:

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
```

```
private \Ninja\DatabaseTable $jokesTable;
private \Ninja\DatabaseTable $authorsTable;
private \Ninja\Authentication $authentication;

public function __construct() {
    $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
    ↪'ijdbuser', 'mypassword');

    $this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
    $this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id');
    $this->authentication = new \Ninja\Authentication($this->authorsTable,
    ↪'email', 'password');
}

public function getController(string $controllerName): ?object {

    if ($controllerName === 'joke') {
        $controller = new \Ijdb\Controllers\Joke($this->jokesTable,
        ↪$this->authorsTable);
    }
    else if ($controllerName === 'author') {
        $controller = new \Ijdb\Controllers\Author($this->authorsTable);
    }
    else if ($controllerName == 'login') {
        $controller = new \Ijdb\Controllers\Login($this->authentication);
    }
    else {
        $controller = null;
    }

    return $controller;
}


public function checkLogin(string $uri): ?string {
    $restrictedPages = ['joke/edit', 'joke/delete'];

    if (in_array($uri, $restrictedPages) && !$this->authentication->
    ↪isLoggedIn()) {
        header('location: /login/login');
        exit();
    }
```

```
        return $uri;
    }
    //...
```

With these changes in place, you can now check that it works. Try visiting `https://v.je/login/login` and log in with one of the username/password combinations you created in the previous chapter. You can verify everything's working by making sure you can't add a new joke before you've logged in, testing the login process, and ensuring you can add a joke once logged in.

Logging Out

Let's add a new button to the site layout to allow either logging in or logging out depending on the current login status. For logged-in users, a "Log out" button should show. For users who aren't logged in, the button should show "Log in".

This change will require editing `layout.html` to add the menu link. However, it will require some logic to display one of the two links. Add the link in `layout.html.php`:

```
//...
<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/joke/list">Jokes List</a></li>
  <li><a href="/joke/edit">Add a new Joke</a></li>

  <?php if ($loggedIn): ?>
  <li><a href="/login/logout">Log out</a></li>
  <?php else: ?>
  <li><a href="/login/login">Log in</a></li>
  <?php endif; ?>

</ul>
//...
```

This code is fairly straightforward. There's an `if` statement that displays a "Log out" link if you're logged in, and a "Log in" link if you aren't logged in.

However, before this can work as intended, we'll need some way to get the user's login status into the new `$loggedIn` variable in `layout.html.php`.

Like the `$title` and `$output` variables that already exist in `layout.html.php`, we'll need to set the `$loggedIn` variable before the layout is loaded. Currently, `layout.html.php` is just included directly with this line from `EntryPoint.php`:

```
include  __DIR__ . '/../../templates/layout.html.php';
```

We could read the logged-in status in `EntryPoint.php` like so:

```
$authentication = new \Ninja\Authentication();
$loggedIn = $authentication->isLoggedIn();
include  __DIR__ . '/../../templates/layout.html.php';
```

However, as I mentioned earlier, making the `Authentication` class available in the `EntryPoint` class is a bad idea, as it ties the generic `EntryPoint` class to this specific login mechanism. But with the code above, if we wanted to use a different login system—such as allowing someone to log in using their Facebook or Google account (a process called OAuth2)—we'd also need to create a different version of `EntryPoint`. Ideally, our `EntryPoint` class should work with any login system we choose to use.

We could solve this using the same approach we used with the `Website` class—by creating a corresponding `Authentication` interface to let us handle different authentication methods. However, thinking ahead a little, we might want to display all kinds of website-specific things in the layout—such as a shopping basket, the latest news down the side, or different buttons for different users. These would require variables with names like `$shoppingBasket` and `$latestNews`—variables that won't exist on every website we build.

As such, unlike the `$title` and `$output` variables, we can't create them in the `EntryPoint` class, as they won't all exist on all websites. We can solve this

the same way we solved all the *different things on different websites* problem: by putting the different logic in the `Website` class, enabling each website to provide its own set of required variables. Let's add a method called `getLayoutVariables` to the `Website` class, which will return an array of variables needed by `layout.html.php`:

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
    private \Ninja\DatabaseTable $jokesTable;
    private \Ninja\DatabaseTable $authorsTable;
    private \Ninja\Authentication $authentication;

    public function __construct() {
        $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ↪'ijdbuser', 'mypassword');

        $this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
        $this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id');
        $this->authentication = new \Ninja\Authentication($this->authorsTable,
        ↪'email', 'password');
    }

    public function getLayoutVariables(): array {
        return [
            'loggedIn' => $this->authentication->isLoggedIn()
        ];
    }
```

Because we've previously promoted all our `DatabaseTable` instances to class variables, we now have the ability to send website-specific data to the layout—such as a list of the latest jokes in the sidebar by returning an array like `['latestJokes' => $this->jokesTable->find(...)]`, then looping through the `latestJokes` variable in `Layout.html.php`. However, as that would require a few modifications to `DatabaseTable.php`, let's keep it simple for now and just stick with login status.

The final step in getting this to work is for the `EntryPoint` class to make any variables from `$website->getLayoutVariables()` available in `layout.html.php`.

One of the first functions we created in this book was the `loadTemplate` function. It takes the name of a template file and creates variables in it from an array. We can use the same function to load the layout. In `EntryPoint.php`, find this line:

```
include  __DIR__ . '/../templates/layout.html.php';
```

Replace it with the code below.

Example: Sessions-LoginLogout[9]

```
$layoutVariables = $this->website->getLayoutVariables();
$layoutVariables['title'] = $title;
$layoutVariables['output'] = $output;

echo $this->loadTemplate('layout.html.php', $layoutVariables);
```

In addition to reading the variables provided by the `getLayoutVariables()` function, I've added the original variables `$title` and `$output` to the array. Without this, those variables wouldn't be available in the template when it's loaded. Finally, `echo` is required instead of `include` to send the complete page to the browser.

With this change in place, you should see a "Log in" link if you're not logged in and a "Log out" link if you're logged in. Try logging in and out a few times to check that everything's working as intended. You can even try registering a new account and then logging in with it!

Assigning Added Jokes to the Logged-in User

Now that users can register and log in, it's time to make it so that when a joke is posted, it's associated with the user who's logged in. We already have an `authorId` column in the `joke` table. All we need to do is give it a value when the joke is added. The `editSubmit` method in the `Joke` controller currently

9. https://github.com/spbooks/phpmysql7/tree/Sessions-LoginLogout

contains this code:

```
public function editSubmit() {
    $joke = $_POST['joke'];
    $joke['jokedate'] = new \DateTime();
    $joke['authorId'] = 1;

    $this->jokesTable->save($joke);

    header('location: /joke/list');
}
```

At the moment, `authorId` is always set to `1`. To get the ID of the logged-in user, we'll need to amend the `Authentication` class slightly to provide a way of retrieving the record for the logged-in user.

In the `Authentication` class, add the following method:

```
public function getUser(): ?array {
  if ($this->isLoggedIn()) {
    return $this->users->find($this->usernameColumn, strtolower($_SESSION
    ↪['username']))[0];
  }
  else {
    return false;
  }
}
```

This function checks to see if the user is logged in, and if they are, returns an array that contains the record representing the user who's logged in. As with the `login` and `isLoggedIn` methods, we need `[0]` after the `find` method call to return the first record retrieved.

It's possible just to return the ID of the logged-in user, but later on we might want to know the name or the email address of the user. Returning the entire record gives more flexibility moving forward.

Let's make the `Authentication` class available in the `Joke` controller by the

class variable and constructor argument:

```
<?php
namespace Ijdb\Controllers;

class Joke {

  public function __construct(private \Ninja\DatabaseTable $jokesTable, private
  ↪\Ninja\DatabaseTable $authorsTable, private \Ninja\Authentication
  ↪$authentication) {
  }
```

Let's also pass the `$authentication` instance when the `Joke` controller is instantiated in `JokeWebsite`:

```
//...
if ($controllerName === 'joke') {
    $controller = new \Ijdb\Controllers\Joke($this->jokesTable,
    ↪$this->authorsTable, $this->authentication);
}
//...
```

Once `JokesController` has access to the authentication class, assigning an author to a joke when it's created is very easy.

Example: Sessions-AuthorId[10]

```
public function editSubmit() {
    $author = $this->authentication->getUser();

    $joke = $_POST['joke'];
    $joke['jokedate'] = new \DateTime();
    $joke['authorId'] = $author['id'];

    $this->jokesTable->save($joke);

    header('location: /joke/list');
}
```

10. https://github.com/spbooks/phpmysql7/tree/Sessions-AuthorId

Whenever a joke is added, the author's ID is assigned to the joke in the database. The currently logged-in user is retrieved from the database and their ID is copied to the `authorId` column of the `joke` table.

But there's a concern here: what if `$author` contains `false` because the user isn't logged in? Because the login check already happens in `EntryPoint`, there's no way for the `editSubmit` method to be called unless someone is logged in.

Go ahead and add a few jokes to the website. You'll need to be logged in, but the jokes should get attributed to the account you're logged in with. You can even try logging in as different accounts and checking that the joke is correctly associated with each account.

You now have a fully functional login system. You can add pages to the website and make them visible only to logged-in users. Your users can register for accounts and log in to the website!

User Permissions

If you've been testing out the login system and have been playing around editing, deleting and adding jokes, you'll have noticed a problem: anyone can delete or edit anyone else's jokes!

For most websites, when someone posts something, they have complete control over it and only they are able to delete it or make changes to it. Imagine how confusing Facebook or Twitter would be if people could edit and delete each other's posts!

Let's add some checks to the site that prevent users from being able to add or edit each other's jokes.

The first thing to do is hide the **Edit** and **Delete** options from the joke list for jokes that don't belong to the logged-in user.

To achieve this, the underlying logic will be as follows:

- iterate over each joke
- for each joke, compare the joke's `authorId` to the `id` of the logged-in user
- if they're the same, this user posted the joke and the **Edit**/**Delete** options should be displayed

To achieve this, we need to make the ID of the currently logged-in user available in the template by passing it from the controller's `list` method and provide the user ID of the person who posted each joke:

```
public function list() {
    $result = $this->jokesTable->findAll();

    $jokes = [];
    foreach ($result as $joke) {
        $author = $this->authorsTable->find('id', $joke['authorId'])[0];

        $jokes[] = [
            'id' => $joke['id'],
            'joketext' => $joke['joketext'],
            'jokedate' => $joke['jokedate'],
            'name' => $author['name'],
            'email' => $author['email'],
            'authorId' => $author['id'],
        ];

    }

    $title = 'Joke list';

    $totalJokes = $this->jokesTable->total();

    $user = $this->authentication->getUser();

    return ['template' => 'jokes.html.php',
      'title' => $title,
      'variables' => [
          'totalJokes' => $totalJokes,
          'jokes' => $jokes,
          'userId' => $user['id'] ?? null
      ]
```

```
    ];

  }
```

This reads the currently logged-in user from `$this->authentication->getUser();`, then passes the user's ID as the `userId` variable to the template.

As the person viewing the page may not be logged in, there may not be an author ID associated with the current user. To account for that, I've used `$author['id'] ?? null` to set the `userId` variable in the template to `null` when no user is logged in.

Finally, in the `jokes.html.php`, add an `if` statement inside the loop that iterates over the jokes. If the currently logged-in user is the user who posted the joke, display the **Edit** and **Delete** options. Otherwise, don't show them.

Example: Sessions-CheckUser[11]

```
// …
echo $date->format('jS F Y');
?>)

<?php if (empty($joke) || $userId == $joke['authorId']): ?>
  <a href="/joke/edit/<?=$joke['id']?>">Edit</a>
  <form action="/joke/delete" method="post">
    <input type="hidden" name="id" value="<?=$joke['id']?>">
    <input type="submit" value="Delete">
  </form>
<?php endif; ?>
  </p>
</blockquote>
<?php endforeach; ?>
```

The clever part here is the `if` statement. It checks that `$userId`—which stores the ID of the currently logged-in user—is equal to the `authorId` of the joke being printed, and it only shows the edit and delete options for jokes that

11. https://github.com/spbooks/phpmysql7/tree/Sessions-CheckUser

were posted by the currently logged-in user.

Mission Accomplished?

If you test the site at this point, you might think that's done. Users can't edit or delete each other's jokes. However, that's not quite true. Users can't see the **Edit** or **Delete** options for jokes they didn't post, but there's nothing to stop them from visiting the edit page directly.

Try visiting `https://v.je/joke/edit/1` and changing the ID in the URL. You'll see the edit page for any of the jokes, regardless of whether or not your account posted them.

To fix this, we need to add a check to this page in the same way we did for the joke list.

Firstly, like the joke list page, we supply the `editjoke.html.php` template with the ID of the logged-in user:

```
public function edit($id = null) {
  // ...
  $author = $this->authentication->getUser();

  $title = 'Edit joke';

  return ['template' => 'editjoke.html.php',
    'title' => $title,
    'variables' => [
      'joke' => $joke ?? null,
      'userId' => $author['id'] ?? null
    ]
  ];
}
```

Then, in `editjoke.html.php`, only display the form if the `userId` matches the joke's `authorId`:

```
<?php if (empty($joke) || $userId == $joke['authorId']): ?>
<form action="" method="post">
  <input type="hidden" name="joke[id]" value="<?=$joke['id'] ?? ''?>">
    <label for="joketext">Type your joke here:</label>
    <textarea id="joketext" name="joke[joketext]" rows="3" cols="40"><?=
    ↪$joke['joketext'] ?? ''?></textarea>
    <input type="submit" name="submit" value="Save">
</form>
<?php else: ?>

<p>You may only edit jokes that you posted.</p>

<?php endif; ?>
```

I've included an error message so that something is displayed in case someone does try to edit a joke they didn't create.

Now the user can't see the edit form for jokes they didn't create. Before celebrating the new security of our site, there's one more thing we need to do.

A sneaky attacker could create an HTML file with a form that posted data to your website. For example, they could create the file `editjoke.html`:

```
<form action="http://v.je/joke/edit/1" method="post">
  <input type="hidden" name="joke[id]" value="1">
  <label for="joketext">Type your joke here:</label>
  <textarea id="joketext" name="joke[joketext]" rows="3" cols="40"></textarea>
  <input type="submit" name="submit" value="Save">
</form>
```

As long as they're logged in to the website, they could submit this form and, regardless of who they're logged in as, edit the joke with the ID `1`. We need to add the same check to the method that handles the form submission.

To do this, we need to read the existing joke from the database and check that the ID matches the ID of the existing user:

```
public function editSubmit($id = null) {
```

```
  $author = $this->authentication->getUser();

  if (isset($id)) {
    $joke = $this->jokesTable->find('id', $id)[0] ?? null;

    if ($joke['authorId'] != $author['id']) {
      return;
    }
  }

  $joke = $_POST['joke'];
  $joke['jokedate'] = new \DateTime();
  $joke['authorId'] = $author['id'];

  $this->jokesTable->save($joke);

  header('location: /joke/list');
}
```

The check here issues the `return` command if the joke's `authorId` column isn't the same as the ID of the currently logged-in user. The `return` command will exit the method, and the rest of the code won't run.

The same thing needs to be done in the `delete` method to prevent someone creating a form to delete other people's jokes.

Example: Sessions-CheckUser-Secured[12]

```
public function delete() {

  $author = $this->authentication->getUser();

  $joke = $this->jokesTable->find('id', $_POST['id'])[0];

  if ($joke['authorId'] != $author['id']) {
    return;
  }

  $this->jokesTable->delete('id', $_POST['id']);
```

12. https://github.com/spbooks/phpmysql7/tree/Sessions-CheckUser-Secured

```
    header('location: /joke/list');
}
```

That's it! You've secured all the relevant features of the website so that jokes can only be edited or deleted by the person who posted them.

It's very easy to forget that hiding a link is not enough to make something secure. You also need to ensure that people can't find the URL and access the page anyway.

The Sky's the Limit

In this chapter, you learned about the two main methods of creating persistent variables—those variables that continue to exist from page to page in PHP. The first stores the variable in the visitor's browser in the form of a cookie. By default, cookies terminate at the end of the browser session, but by specifying an expiry time, they can be preserved indefinitely. Unfortunately, cookies are fairly unreliable, because you have no way of knowing when the browser might delete your cookies, and because some users occasionally clear their cookies out over concern for their privacy.

Sessions, on the other hand, free you from all the limitations of cookies. They let you store an unlimited number of potentially large variables. Sessions are an essential building block in modern ecommerce applications, as we demonstrated in our simple shopping cart example. They're also a critical component of systems that provide access control, like the one we built for our joke content management system.

At this point, you should be equipped with all the basic skills and concepts you need to build your very own database-driven website. While you may be tempted to skip the challenge of building a complete system for safely accepting public submissions, I encourage you to give it a try. You already have all the skills necessary to build it, and there's no better way to learn than to make a few mistakes of your own. At the very least, set this challenge aside for now and come back to it when you've finished this book.

If you can tackle it with confidence, you may wish to try another challenge. Perhaps you'd like to let users rate the jokes on the site? How about letting joke authors make changes to their jokes, but with the backup of requiring an administrator to approve the changes before they go live on the site? The power and complexity of the system is limited only by your imagination.

For the remainder of this book, I'll cover more advanced topics that will help optimize our site's performance and solve some complex problems using less code. Oh, and of course we'll explore more exciting features of PHP and MySQL!

In Chapter 12, we'll learn about handling related data, such as how to associate jokes with the user who posted them, and how put jokes inside categories.

Relationships

Chapter 12

As you've worked through the construction of the Internet Joke Database website, you've had opportunities to explore most aspects of Structured Query Language (SQL). You've been introduced to many SQL commands now—from the basic form of a `CREATE TABLE` query to the two syntaxes of `INSERT` queries.

In Chapter 5, I showed you how to perform basic `JOIN` s using SQL to fetch data from more than one table at a time. A lot of the time, you'll come across situations where you want to do this—for example, finding information about an author as well as all the jokes they've posted, or finding a category and all the jokes that reside inside it.

SQL `JOIN` is one of many solutions to this problem. Although there are performance advantages to using `JOIN`, unfortunately `JOIN` s don't work well with object-oriented programming. The *relational* approach used by databases is generally incompatible with the nested structure of object-oriented programming. In object-oriented programming, objects are stored in a hierarchical structure. An author **contains**—or, in the correct OOP terminology, **encapsulates**—a list of their jokes, and a category also encapsulates a list of the jokes within the category.

A `SELECT` query that fetches an author along with all of their jokes can be written like this:

```
SELECT author.name, joke.id, joke.joketext
FROM author
INNER JOIN joke ON joke.authorId = author.id
WHERE authorId = 123
```

This is a *relational* structure, where the `authorId` column in the `joke` table references a record in the `author` table by its ID, as pictured below.

id	jokeText	jokeDate	authorId
1	Why did the programmer quit his job? He ...	2021-11-05	1
2	!false - it's funny because it's true	2021-11-02	2
3	How many programmers does it take to c...	2021-11-06	1

id	name
1	Tom Butler
2	Kevin Yank

12-1. Relational data

Using an object-oriented approach, there are various practical ways of achieving this, which we'll look into shortly. In an object-oriented approach, data is hierarchical, one object being nested inside another. The example above could be expressed as arrays like so:

```
$jokes = [
  [
    'jokeText' => 'Why did the programmer quit his job? He didn\'t get arrays',
    'jokeDate' => '2021-11-05',
    'author' => [
      'name' => 'Tom Butler'
    ]
  ],
  [
    'jokeText' => '!false - it\'s funny because it\'s true',
    'jokeDate' => '2021-11-02',
    'author' => [
      'name' => 'Kevin Yank'
    ]
  ],
  [
    'jokeText' => 'How many programmers does it take to change a lightbulb? None,
    ➥it\'s a hardware problem.',
    'jokeDate' => '2021-11-06',
    'author' => [
      'name' => 'Tom Butler'
    ]
  ]
];
```

In the hierarchical structure above, the author's name can be read by extracting the author from the array and then reading the corresponding name:

```
echo $jokes[0]['author']['name'];
```

Each entry is one level down the data structure. In a hierarchical structure, there are no levels in the data structure, as nothing is nested. Data is related by repeating the identifier in multiple tables.

Taking this to the next level in object-oriented programming, it's possible to represent relational data as a hierarchical data stricture. After this is implemented, fetching a list of jokes will be expressed like this:

```
// Find the author with the id `123`
$author =  $authors->find('id', 123)[0];

// Get all the jokes by this author
$jokes = $author->getJokes();

// Print the text of the first joke by that author.
echo $jokes[0]->joketext;
```

Notice that there's no SQL here. The data will be coming from the database, but it all happens behind the scenes.

We could fetch all the information by using the SQL query I provided above, but this doesn't work well with the `DatabaseTable` class we've used so far (or any object approach in general). It would be very difficult to design the class in such a way that it would account for every possible set of relationships we may want.

So far, we've dealt with the relationship between jokes and authors in a *relational* way. If we wanted to get the information about an author, along with a list of all their jokes, we'd do this:

```
// Find the author with the ID 123
$author = $this->authors->find('id', 123)[0];

// Now find all the jokes posted by the author with that ID
$jokesByAuthor = $this->jokes->find('authorId', $author['id']);
```

This runs two separate `SELECT` queries: the first fetches the author from the `author` table, and the second fetches a list of jokes by that particular `author`. Using this approach, the two `DatabaseTable` instances are entirely separate. Neither is aware that any other tables even exist.

We also used a similar approach to handle two tables when inserting a joke into the database:

```
public function editSubmit($id = null) {
  $author = $this->authentication->getUser();

  if (!empty($id)) {
    $joke = $this->jokesTable->find('id', $id)[0];

    if ($joke['authorId'] != $author['id']) {
      return;
    }
  }

  $joke = $_POST['joke'];
  $joke['jokedate'] = new \DateTime();
  $joke['authorId'] = $author['id'];

  $this->jokesTable->save($joke);

  header('location: /joke/list');
}
```

This code uses the `authentication` class to fetch the record that stores the currently logged-in user. Behind the scenes, this fetches a record from the `author` table. It then reads the author's `id` in order to provide it when the joke is added to the `joke` table: `$joke['authorId'] = $author['id'];`.

Whoever writes this code must know about the underlying structure of the database and that the authors and jokes are stored in a *relational* way.

In object-oriented programming, it's preferable to hide the underlying implementation, and the code above would be expressed like this:

```
public function editSubmit($id = null) {
  $author = $this->authentication->getUser();

  if (!empty($id)) {
    $joke = $this->jokesTable->find('id', $id)[0];
```

```
    if ($joke['authorId'] != $author['id']) {
      return;
    }
  }

  $joke = $_POST['joke'];
  $joke['jokedate'] = new \DateTime();

  $author->addJoke($joke);

  header('location: /joke/list');
}
```

Not a lot has changed, so look closely! Firstly, the `$joke['authorId'] = $author['id'];` line has been removed. Secondly, instead of saving the joke to the `$jokesTable` object, it's being passed to the `author` object: `$author->addJoke($joke);`.

What is the advantage of this approach? The person who writes this code doesn't have to know anything about what happens behind the scenes, or how the relationship is modeled in the database—that there's an `authorId` column in the `joke` table.

Instead of modeling the relationships in a relational way, object-oriented programming takes a *hierarchical* approach, using data structures nested inside other data structures. A category contains a list of jokes, and each joke in the list contains an author.

In the example above, the `$author->addJoke($joke)` method call *might* be writing the joke data to a database. Alternatively, it might be saving the data to a file. And that file could be in JSON format, XML format, or be an Excel spreadsheet. The developer who writes this `editSubmit` method doesn't need to know anything about the underlying storage mechanism—*how* data is being stored—but only that the data *is* being stored somehow, and that it's being stored *inside* the author instance.

In object-oriented programming terminology, this is known as

implementation hiding, and it has several advantages. Different people can work on different sections of the code. The developer who writes the `editSubmit` method doesn't have to be familiar with how `addJoke` actually works. They only need to know that it exists and that it *saves data*, and that the data can be retrieved later.

When you use the `$pdo->query` method, you don't need to know how `$pdo` actually communicates with the database. You only need to know what the method returns and what arguments it requires. We can imagine what the following lines of code do, knowing *how* each of them works:

```
$jokes = $author->getJokes();
```

```
echo $joke->getAuthor()->name;
```

```
$joke = $_POST['joke'];
$joke['jokedate'] = new \DateTime();

$author->addJoke($joke);
```

In the middle example, as long as you know there's a `getAuthor` method that you can call on the `$joke` instance, it doesn't matter how it works. The author name could be hardcoded into the class, or it could go off and fetch it from the database.

This is particularly useful, because the storage system can be changed at any time, and the code above doesn't need to change. The methods `getJokes` and `getAuthor` can be completely rewritten to write to/read from a file, for example, but the code above will still work without any further changes.

If the `editSubmit` method contained the `INSERT` SQL query with all the relevant database fields and the calls to `$pdo->prepare()` and `$pdo->execute()`, and if we wanted to store the joke in a file instead of a database, the entire method would need to be rewritten. On the other hand, using the code above, where nothing about *how* the data is being stored is present in

the class, we could modify the code inside the `$author->addJoke($joke)` to save data to a file instead of the database.

This approach to splitting up the logic is loosely known as **separation of concerns**. The process of *saving a joke* is different from the process of *writing data to the database*. Each of these is a different *concern*. By splitting out the two concerns, you have a lot more flexibility. The `addJoke` method can be called from anywhere without needing to repeat the logic. The way the `addJoke` method works can be completely rewritten to work in a different way, but the code that calls it can remain unchanged.

This added flexibility is incredibly useful for an increasingly popular development methodology called **test-driven development** (**TDD**). In TDD, you'd test the code by writing a version of the `DatabaseTable` class that worked as a placeholder for the test, rather than needing a working database containing relevant test data.

Although TDD is beyond the scope of this book, by thinking about separation of concerns, you'll be able to start writing automated tests for your code without making large-scale changes.

A common problem that mid-level developers have when first learning about TDD is that they understand the advantages of it, but their code isn't written in such a way that makes it easy to test. By considering separation of concerns earlier on in your programming career, you'll make it easier for yourself moving forward.

For more information on TDD, check out the introductory article "Re-Introducing PHPUnit: Getting Started with TDD in PHP"[1].

Object Relational Mappers

The `DatabaseTable` class we've built step by step so far is a type of library called an **object relational mapper** (or **ORM**). There are a lot of ORM

---

1. https://www.sitepoint.com/re-introducing-phpunit-getting-started-tdd-php/

implementations available, such as Doctrine[2], Propel[3] and ReadBeanPHP[4]. These all do essentially the same job as the `DatabaseTable` class we've been building—providing an object-oriented interface for a relational database. They bridge the gap between the relational database's SQL queries and the PHP code we're using for everything else on the website.

Generally, ORMs deal with *objects*. Using our `DatabaseTable` class to find an author and print their name, we can use this code:

```
$author = $authors->find('id', 123)[0];

echo $author['name'];
```

Here, the `$author` variable is an array with keys for each of the columns in the database. Arrays can't contain functions, so implementing an `addJoke` method on the `$author` instance isn't possible.

Arrays Containing Functions

OK, what I said above about arrays not being able to contain functions isn't entirely true. They can contain a special type of function called a **closure**, but doing so has several severe limitations compared to the OOP-based approach I'm going to show you instead.

If we want the ability to call methods on the `$author` instance, such as `$author->addJoke($joke)` like above, the `$author` variable needs to be an *object* rather than an array. The first thing we need to do is create the relevant class to represent authors. Firstly, some properties for each of the columns from the database:

---

2. http://www.doctrine-project.org/
3. http://propelorm.org/
4. http://redbeanphp.com/index.php

```
namespace Ijdb\Entity;

class Author {
  public int $id;
  public string $name;
  public string $email;
  public string $password;
}
```

As the properties of an author are unique to the joke website, I've put the class in the `Ijdb` namespace.

A class like this, which is designed to map directly to a record in the database, is commonly known as an **Entity Class**—which is why I've used the name `Entity` in the namespace. We'll have a different entity class for each of the database tables we need to represent. Let's create the directory `Entity` inside the `Ijdb` folder and save the class inside it with the name `Author.php`.

Although the variables don't *need* to be declared here, and what we're doing will work identically whether they are or aren't, it makes the code easier to read and understand if the variables are included.

There's some repetition here: every time you add a column to the database table, you'll need to add it to this entity class. Because of this, many ORMs provide a method of generating these entity classes from the database schema, or even creating the database table from the object!

I'm not going to show you how to do that, but if you want to try something similar, you should take a look at the MySQL `DESCRIBE` query[5] to retrieve a list of columns in a table, or the PHP Reflection library[6] to get a list of properties in a class.

5. https://dev.mysql.com/doc/refman/5.7/en/getting-information.html
6. http://php.net/manual/en/book.reflection.php

Public Properties

Every time we've created a class variable so far, it's been *private*, so that the data used is only accessible to methods within the class. The advantage of this is that class variables can be easily added, renamed or removed without potentially breaking any of the code that uses the class.

It also prevents developers accidentally breaking the functionality of the class. If the `pdo` variable were public in the `DatabaseTable` class, it would be possible to do this:

```
$this->jokesTable->pdo = 1234;
```

Any further call to methods on the `DatabaseTable` class would break:

```
$this->jokesTable->find('id', 1);
```

The `find` method would call `$this->pdo->prepare('…')`, but because the `pdo` variable is no longer a `PDO` instance, it would break!

In most cases, private properties are strongly preferred over public ones. However, in the case of entity classes, you should use public properties.

The sole purpose of an entity class is to make some data available. It's no good having a class representing an author if you can't even read the author's name!

Nine times out of ten—in fact, ninety-nine times out of a hundred—public properties are the wrong solution to any given problem. However, if the responsibility of the class is to represent a *data structure*, and it's interchangeable with an *array*, then public properties are fine.

As a rule of thumb, if your class does anything but represent a *data structure*, then it should contain only private properties.

Methods in Entity Classes

A *class* is used to store the data about authors instead of an *array*, because the class can contain methods, and we can do things like this:

```
// Find the author with the id 1234
$author = $this->authorsTable->find('id', 1234)[0];

//Print the author's name
echo $author->name;

// Find all the jokes by that author
$jokes = $author->getJokes();

// Add a new joke and associate it with the author represented by $author
$author->addJoke($joke);
```

Let's take a moment to think about what the `getJokes` method might look like. Assuming the `id` property in the `$author` class is set, it would be possible to do this:

```
public function getJokes(): array {
  return $this->jokesTable->find('authorId', $this->id);
}
```

Assuming all the class properties were set up correctly, these two pieces of code are equivalent:

```
// (A) Using the current array based DatabaseTable class:

$author = $authorsTable->find('email', 'tom@r.je')[0];

echo $author['name'];

$tomsJokes = $jokesTable->find('authorId', $author['id']);

// (B) Using a `getJokes` method in the entity class:

$author = $authorsTable->find('email', 'tom@r.je')[0];
```

```
echo $author->name;

$tomsJokes = $author->getJokes();
```

In method *(A)*, using arrays, the author's ID is extracted from the `$author` array and then passed into the `find` method for the `$jokesTable` instance.

In method *(B)*, this process is hidden behind the `getJokes` method in the entity class. The advantages of this approach are as follows:

- Whatever code needs the list of jokes only needs access to the `$author` variable and not also the `$jokesTable` variable. This is particularly important in template files, as we'll see shortly.
- The way the relationship is modeled is expressed in a single location. If we want to change the relationship to use the `email` field, it can change in this single location.

To implement method *(B)*, the author class needs access to the `jokesTable` instance of the `DatabaseTable` class. Let's add the `getJokes` method, along with a constructor and class variable, to store the reference to the `jokesTable` instance:

```
<?php
namespace Ijdb\Entity;

class Author {
    public $id;
    public $name;
    public $email;
    public $password;

    public function __construct(private \Ninja\DatabaseTable $jokesTable) {

    }

    public function getJokes() {
```

```
        return $this->jokesTable->find('authorId', $this->id);
    }
}
```

We're going to amend the `DatabaseTable` class to return an instance of this class instead of an array. But before we do that, let's take a look at how the `Author` class can be used on its own:

```
$jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');

$author = new \Ijdb\Entity\Author($jokesTable);

$author->id = 123;

$jokes = $author->getJokes();
```

Firstly, the `$jokesTable` instance is created, as we've been doing since I first introduced objects. Now that the `Author` entity class requires it as a constructor argument, it's passed in as one when the `$author` instance is created.

The `$author->id = 123;` line sets the ID of the author we're representing. Normally, this would come from the `id` column in the database, but for this simplified example I've set it manually.

With the ID set, when the `$jokes = $author->getJokes();` line runs, it executes the `getJokes` method in the `Author` entity class, fetching all the jokes associated with the author with the ID specified on the instance.

In addition to reading the list of jokes, let's add functionality for adding a joke using an object-oriented approach. The relationship will be created behind the scenes. The programmer using the class will add a joke, but won't have to specify the ID of the author manually.

Once this is implemented, adding a joke will be done like this:

```
$joke = [
  'jokeText' => 'Why did the programmer quit his job? He didn\'t get arrays',
  'jokeDate' => '2021-11-05'
];

$author->addJoke($joke);
```

Notice that the `$joke` array doesn't contain any information about who posted the joke. A joke is associated by calling a method called `addJoke` on an `Author` instance. This method will then create the association between the joke and the author, then add the joke to the database.

Let's add this new method to the `Author` entity class. It will take a joke as an argument, then set the `authorId` property and finally insert the joke into the database:

```
public function addJoke(array $joke) {
  // set the `authorId` in the new joke to the id stored in this instance
  $joke['authorId'] = $this->id;

  $this->jokesTable->save($joke);
}
```

Let's use the new class inside the `Joke` controller's `editSubmit` method to save the jokes using `$author->addJoke($joke)`.

Example: Relationships-Author[7]

```
public function editSubmit() {
  // Get the currently logged in user as the $author to associate the joke with
  $author = $this->authentication->getUser();

  // Create an `Author` entity instance
  $authorObject = new \Ijdb\Entity\Author($this->jokesTable);

  // Copy the values from the `$author` array into the corresponding properties
```

7. https://github.com/spbooks/phpmysql7/tree/Relationships-Author

```
  // in the entity object
  $authorObject->id = $author['id'];
  $authorObject->name = $author['name'];
  $authorObject->email = $author['email'];
  $authorObject->password = $author['password'];

  // Read the form submission and set the date
  $joke = $_POST['joke'];
  $joke['jokedate'] = new \DateTime();

  // Save the joke using the new addJoke method
  $authorObject->addJoke($joke);

  header('location: /joke/list');
}
```

Because the database returns data as an array, I've copied the data from the `$author` array returned by `getUser()` to an instance of the newly created `Author` class.

Both the `$author` array and the `$authorObject` object will represent the same author. The only difference is that one is an object and the other is an array.

Most of the lines of code in the method simply copy data from the array to an object. This is obviously inefficient, and the problem can be avoided if we can construct the `Author` object outside the `editSubmit` method, and have `getUser` return the constructed object, like so:

```
public function editSubmit() {
  $authorObject = $this->authentication->getUser();

  $joke = $_POST['joke'];
  $joke['jokedate'] = new \DateTime();

  $authorObject->addJoke($joke);

  header('location: /joke/list');
}
```

Using Entity Classes from the DatabaseTable Class

You might consider moving the logic that copies the array to an object into the `getUser` method, but we can take it a step further than that. Any author or joke created would be useful as an object, so we can amend the `find` method in the `DatabaseTable` class to always return an object.

The `find` method currently looks like this:

```
public function find($column, $value) {
    $query = 'SELECT * FROM `' . $this->table . '` WHERE `' . $column . '` =
    ↪:value';

    $values = [
        'value' => $value
    ];

    $stmt = $this->pdo->prepare($query);
    $stmt->execute($values);

    return $stmt->fetchAll();
}
```

Here, `$stmt->fetchAll()` returns an array of records returned by the query. Each entry in the array is itself an array that represents a row from the table. For example, when fetching an author, this array contains the keys `id`, `email`, `name` and `password`.

We could add some code here to create the relevant `Author` instance, then copy all the values from the array into the object (as I did above). However, the developers of the PDO library are one step ahead of this, and the `fetchAll` method can take some arguments to configure it to return an array where each record is represented by an object instead of an array.

To tell the `fetchAll` method to return an `Author` instance, we can use these arguments:

```
return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
↪  '\Ijdb\Entity\Author', [$jokesTable]);
```

The first argument, `\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE`, tells PDO to create an instance of a class instead of an array. The second argument, `'\Ijdb\Entity\Author'` in this case, is the name of the class it should create an instance of (with the namespace and in quotes). The final argument is a list of constructor arguments needed to create an instance of the class named in the second argument.

The final argument looks a bit strange, because it's a variable—`$jokesTable`—inside square brackets. This code creates an array with `$jokesTable` in index zero and is equivalent to the following:

```
$array = [
  0 => $jokesTable
];

return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
↪'\Ijdb\Entity\Author', $array);
```

The third argument has to be an array, because the class listed in the second argument might require more than one argument in its constructor.

Because the `Author` class requires the `$jokesTable` class as a constructor argument, this must also be provided as an argument when `fetchAll` is called. However, we can't amend the `DatabaseTable` class to use the `return` line above—firstly, because it doesn't have access to the `$jokesTable` variable, and secondly, because we're using the `DatabaseTable` class to interact with different database tables. When the `find` method is called, it may be on the `authorsTable` instance, the `jokesTable` instance, or an instance of the `DatabaseTable` class that represents some other database table.

We'll want a different entity class for each table, and they almost certainly won't have the same constructor arguments.

Instead of hardcoding the class name and constructor argument, we can amend the constructor of the `DatabaseTable` class to take two *optional* arguments—the name of the class to create, and any arguments to provide to it:

```
class DatabaseTable {

  public function __construct(private \PDO $pdo, private string $table, private
  ↪string $primaryKey, private string $className = '\stdClass', private array
  ↪$constructorArgs = []) {

  }
```

Any time an instance of `DatabaseTable` is created, the corresponding entity class name and array of constructor arguments for that entity can optionally be provided. These values will then be used in the `find` and `findAll` methods to return objects instead of arrays.

Notice that I've given *default values* to each of the new arguments. The `stdClass` class is an inbuilt PHP empty class that can be used for simple data storage.

By specifying this as a default value, if no class name is specified, it will use this generic inbuilt one. The advantage this gives us is that we don't need to create a unique entity class for every database table, but only those we want to add methods to!

The `find` method can now be changed to read the new class variables and use those in place of hardcoded ones in the `fetchAll` method call:

```
public function find($column, $value) {
    $query = 'SELECT * FROM `' . $this->table . '` WHERE `' . $column . '` =
    ↪:value';

    $values = [
        'value' => $value
    ];
```

```
    $stmt = $this->pdo->prepare($query);
    $stmt->execute($values);

    return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
    ↪$this->className, $this->constructorArgs);
}
```

Finally, make the same change in the `findAll` method:

```
public function findAll() {
    $query = 'SELECT * FROM `' . $this->table . '`';

    $stmt = $this->pdo->prepare($query);
    $stmt->execute();

    return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
    ↪$this->className, $this->constructorArgs);
}
```

Now that we've amended the `DatabaseTable` class, we can change the `Website` class where the `DatabaseTable` class is instantiated, and provide the class name and the arguments for the `$authorsTable` instance:

```
$this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
$this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id',
↪'\Ijdb\Entity\Author', [$this->jokesTable]);
```

With this change in place, when the `$authorsTable` instance is used to retrieve a record, like this:

```
$author = $authorsTable->find('id', 123)[0];
```

… the `$author` variable will be an instance of the `\Ijdb\Entity\Author` class, and any methods in the class (such as `addJoke`) will available for us to call.

As we've made a change to the `DatabaseTable` class, this will affect every instance of the class. When jokes are retrieved from the database, an object

will also be retrieved. As we haven't specified an entity class for jokes (yet!), objects returned from the `find` method on the `$jokesTable` instance will be an instance of the generic `stdclass` class.

If you load up the joke list page in your browser, at this point you'll see this error:

```
Fatal error: uncaught Error: cannot use object of type stdClass as array in
↪/app/classes/Ijdb/Controllers/Joke.php on line 21
```

Because we've replaced the arrays with objects, we'll need to tweak anywhere we read a column from the database. We'll fix this later, but first let's get `editSubmit` working. Amend the `editSubmit` method to avoid all the value copying. Remove the following code:

```
public function editSubmit($id = null) {
    $author = $this->authentication->getUser();

    if (!empty($id)) {
      $joke = $this->jokesTable->find('id', $id)[0];

      if ($joke['authorId'] != $author['id']) {
       return;
      }
    }

    $joke = $_POST['joke'];
    $joke['jokedate'] = new \DateTime();
    $joke['authorId'] = $author['id'];

    $this->jokesTable->save($joke);

    header('location: /joke/list');
}
```

Replace it with this:

```
public function editSubmit() {
```

```
    $author = $this->authentication->getUser();

    if (!empty($id)) {
      $joke = $this->jokesTable->find('id', $id)[0];

      if ($joke->authorId != $author->id) {
       return;
      }
    }

    $joke = $_POST['joke'];
    $joke['jokedate'] = new \DateTime();

    $author->addJoke($joke);

    header('location: /joke/list');
}
```

This is because `getUser` now returns an instance of `Author` with all the properties already set and containing the `addJoke` method. Rather than manually describing the relationship by manually setting the `authorId`, the relationship is modeled in the `Author` class. I've also updated the check inside the second `if` statement to use the object notation for `$joke->authorId` and `$author->id`.

Before we can test that this works, we'll also need to amend the `Authentication` class to use an object rather than an array. In the `isLoggedIn` method, remove this line:

```
if (!empty($user) && $user[0][$this->passwordColumn] === $_SESSION['password']) {
```

Replace it with this:

```
if (!empty($user) && $user[0]->{$this->passwordColumn} === $_SESSION
↪['password']) {
```

This code looks complicated! However, let's take a moment to understand what's happening here.

The `$user[0]` variable now stores an instance of the `Author` class. The `Author` class has a property called `password`. To read this property, you could use this:

```
$user[0]->password
```

However, if you remember back to Chapter 11, you'll recall that the `Authentication` class has a class variable, which stores the name of the database column that contains the password. It might not be `password` in every website you build.

Assuming the password column *is* `password`, once the variables have been evaluated, all we're doing here is replacing `$user[0]['password']` with the object variant `$user[0]->password`. But this is complicated by the fact that we're using a variable for the column name. We really need `$user[0][$this->passwordColumn]` to be replaced with `$user[0]->$this->passwordColumn`.

It's possible to use a variable to access a property on an object using a string, as we do with an array:

```
$columnName = 'password';

// Read value stored under key 'password' from array
$password = $array[$columnName];

// Read value stored under property 'password' from object
$password = $object->$columnName;
```

Because the column name is stored in the class variable `$this->passwordColumn`, we need to perform something like `$user[0]->$this->passwordColumn`. However, this expression confuses PHP, because the arrow operator is being chained. This will evaluate `$user[0]->$this` first. Starting with `$this`, it will look up the value of `$this` and then look for a property with that name. As `$this` evaluates to an object, not a valid property name, we need to tell PHP to evaluate `$this->passwordColumn` prior to `$user[0]->$this->passwordColumn`.

By including braces around `{this->passwordColumn}` in the expression, PHP will evaluate the `$this->passwordColumn` column first. This is the equivalent of the following:

```
$columnName = $this->passwordColumn;

$user[0]->$columnName;
```

Now that the `isLoggedIn` method is using objects, we can do the same with this line in the `login` method:

```
public function login(string $username, string $password): bool {
    $user = $this->users->find($this->usernameColumn, strtolower($username));

    if (!empty($user) && password_verify($password, $user[0][$this->
    ↪passwordColumn])) {
        session_regenerate_id();
        $_SESSION['username'] = $username;
        $_SESSION['password'] = $user[0][$this->passwordColumn];
        return true;
    } else {
        return false;
    }
}
```

The login method above needs the same modification as the `isLoggedIn` method, changing the array notation to the object notation.

Example: Relationships-DatabaseTableEntity[8]

```
public function login(string $username, string $password): bool {
    $user = $this->users->find($this->usernameColumn, strtolower($username));

    if (!empty($user) && password_verify($password, $user[0]->{$this->
    ↪passwordColumn})) {
        session_regenerate_id();
        $_SESSION['username'] = $username;
```

8. https://github.com/spbooks/phpmysql7/tree/Relationships-DatabaseTableEntity

```
        $_SESSION['password'] = $user[0]->{$this->passwordColumn};
        return true;
    } else {
        return false;
    }
}
```

Finally, update the `getUser` method's return value from `?array` to `?object` as the user is now being represented by an `author` object instead of an array.

After submitting the form, you'll see an error on the list page, but you can check the new `editSubmit` method is working by logging in to the website and adding a joke. When you're redirected to the list page, you'll see an error, but you can check that the joke has been added by viewing the contents of the `joke` table in MySQL Workbench.

Joke Objects

Now we'll fix the joke list page. At the moment, it displays an error, thanks to this code in the controller:

```
$author = $this->authorsTable->find('id', $joke['authorId'])[0];

$jokes[] = [
  'id' => $joke['id'],
  'joketext' => $joke['joketext'],
  'jokedate' => $joke['jokedate'],
  'name' => $author['name'],
  'email' => $author['email']
];
```

The error occurs because `$author` and `$joke` are no longer arrays. This is a simple fix. Let's change the syntax that reads from an array to use the object syntax:

```
$author = $this->authorsTable->find('id', $joke->>authorId])[0];
```

```
$jokes[] = [
  'id' => $joke->id,
  'joketext' => $joke->joketext,
  'jokedate' => $joke->jokedate,
  'name' => $author->name,
  'email' => $author->email,
  'id' => $author->id,
];
```

We'll also need to change the method's `return` statement to use the object syntax for the `$author` variable:

```
$totalJokes = $this->jokesTable->total();

$user = $this->authentication->getUser();

return ['template' => 'jokes.html.php',
  'title' => $title,
  'variables' => [
    'totalJokes' => $totalJokes,
    'jokes' => $jokes,
    'userId' => $user->id ?? null
  ]
];
```

Now let's also amend the `delete` method to read the joke's `authorId` from the new object.

Example: Relationships-Objects[9]

```
if ($joke->authorId != $author->id) {
  return;
}
```

Let's go back to the list page and re-examine the logic here. The updated method using objects looks like this:

---

9. https://github.com/spbooks/phpmysql7/tree/Relationships-Objects

```
public function list() {
  $result = $this->jokesTable->findAll();

  $jokes = [];
  foreach ($result as $joke) {
    $author = $this->authorsTable->find('id', $joke->authorId);

    $jokes[] = [
      'id' => $joke->id,
      'joketext' => $joke->joketext,
      'jokedate' => $joke->jokedate,
      'name' => $author->name,
      'email' => $author->email,
      'authorId' => $author->id
    ];

  }
```

Although this solution works, now that we're using an object-oriented approach, it can be solved in a much nicer way. Currently, each value from either the `author` or `joke` table is stored under an equivalent key in the `$jokes` array.

The `$jokes` array is used to provide the template access to each joke and its author.

The process currently looks like this:

- query the database and select all the jokes
- loop over each joke and:
  - select the related author
  - create a new array containing all the information about the joke and the corresponding author
- pass this constructed array to the template for display

This is a very long-winded process for something we can make a lot simpler using OOP.

At the moment, we can fetch all the jokes by a specific author using `$author-`

`>getJokes()`. However, we can also model the inverse relationship and do something like this:

```
echo $joke->getAuthor()->name;
```

This would let us get the author for any given joke, and this code could even be run from the template.

If the *`$this->jokesTable->findAll();`* call returned an array of joke *objects*, each with their own *`getAuthor`* method, this process of creating an array with both sets of data would be unnecessary!

Firstly, let's create the *`Joke`* entity class in *`Ijdb/Entity/Joke.php`*:

```
<?php
namespace Ijdb\Entity;

class Joke {
    public int $id;
    public int $authorId;
    public string $jokedate;
    public string $joketext;

    public function __construct(private \Ninja\DatabaseTable $authorsTable) {
    }

    public function getAuthor() {
        return $this->authorsTable->find('id', $this->authorId)[0];
    }
}
```

The *`Joke`* class works in the same way as the *`Author`* class: it has a constructor that asks for an instance of a *`DatabaseTable`* class that represents related data. In this case, it will be passed the *`DatabaseTable`* instance, which can be used to interact with the *`author`* database table.

The *`getAuthor`* method returns the author for the current joke. If *`$this->authorId`* is *`5`*, it will return an *`Author`* object that represents the author

with the ID `5`.

Using the Joke Class

To use the new `Joke` class, we'll need to update the `Website` class to provide the `authorsTable` instance as a constructor argument. The constructor creates the two `DatabaseTable` instances and looks like this:

```
public function __construct() {
    $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
    ↪'mypassword');

    $this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id');
    $this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id',
    ↪'\Ijdb\Entity\Author', [$this->jokesTable]);
    $this->authentication = new \Ninja\Authentication($this->authorsTable,
    ↪'email', 'password');
}
```

When the `authorsTable` instance is created, the fourth and fifth arguments provide the name of the class that should be created (`\Ijdb\Entity\Author`), and an array of constructor arguments for that class (`[$this->jokesTable]`).

To apply the same logic for the `jokesTable` instance, we need to supply the class name and constructor arguments when the instance is created:

```
$this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id',
↪'\Ijdb\Entity\Joke', [$this->authorsTable]);
$this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id',
↪'\Ijdb\Entity\Author', [$this->jokesTable]);
```

This will pass the `authorsTable` instance to the `jokesTable` instance when `jokesTable` is instantiated.

Think about what's going on here for a moment. It poses a problem that's not immediately obvious.

If the `authorsTable` instance constructor requires an instance of `jokesTable`, and the `jokesTable` constructor requires an `authorsTable` instance, we have a catch-22: to create the `jokesTable` instance, we need an existing `authorsTable` instance. To create the `authorsTable` instance, we need an existing `jokesTable` instance. Both instances require the other instance to exist before they do!

If you try the code above, the PDO library will throw an exception: “Cannot call constructor”. Although the message isn’t very clear, it’s because of the problem I just highlighted.

This catch-22 occurs sometimes in object-oriented programming. Luckily, in this case (but not always) it can be fairly easily solved using something called a “reference”.

References

A **reference** is special type of variable—a bit like a shortcut in Windows, or a symlink in macOS or Linux. A **shortcut** on your computer is a special type of file that doesn’t contain any data itself. Instead, it points to another file. When you open the shortcut, it actually opens the file the shortcut is pointing to.

References work in a similar way. Instead of a variable containing a specific value, it can contain a *reference* to another variable. When you read the value of a variable that stores the reference, it will read the value of the variable being referenced.

To create a reference, you prefix the variable you want to create a reference to with an ampersand (`&`):

```
$originalVariable = 1;
$reference = &$originalVariable;
$originalVariable = 2;
echo $reference;
```

The code above will print `2`. Without the `&`, it would print `1`! That’s because

the variable `$reference` contains a reference to the `$originalVariable` variable. Whenever the value of `$reference` is read, it will actually go off and read the value of the variable `$originalVariable` as it is at that moment in time.

This is important, because it allows us to solve the catch-22 we encountered earlier. By providing references as the constructor arguments for the `Joke` and `Author` classes, by the time the `authorsTable` and `jokesTable` instances are needed, they'll have been created:

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
    private ?\Ninja\DatabaseTable $jokesTable;
    private ?\Ninja\DatabaseTable $authorsTable;
    private \Ninja\Authentication $authentication;

    public function __construct() {
        $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4', 'ijdbuser',
        ↪'ijdbuser', 'mypassword');

        $this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id',
        ↪'\Ijdb\Entity\Joke', [&$this->authorsTable]);

        $this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id',
        ↪'\Ijdb\Entity\Author', [&$this->jokesTable]);

        $this->authentication = new \Ninja\Authentication($this->authorsTable,
        ↪'email', 'password');

    }

   public function getDefaultRoute(): string {
        return 'joke/home';
    }

    public function getController(string $controllerName): ?object {

        if ($controllerName === 'joke') {
            $controller = new \Ijdb\Controllers\Joke($this->jokesTable,
            ↪$this->authorsTable);
        }
```

```
        else if ($controllerName === 'author') {
            $controller = new \Ijdb\Controllers\Author($this->authorsTable);
        }
        else {
            $controller = null;
        }

        return $controller;
    }
}
```

Note that because we are trying to access the `authorsTable` and `jokesTable` class variables before they are initialized, we need to specify that they can be null in the type hint by setting it to `?\Ninja\DatabaseTable`.

Now, when the `DatabaseTable` class creates a `Joke` or `Author` object and has to provide it the `authorsTable` or `jokesTable` instance, it will read what's stored in the `authorsTable` or `jokesTable` class variables at the time any `Author` or `Joke` entity is instantiated.

This does make the code potentially unstable. Because we're using references, if `$this->jokesTable` is overwritten in the `Website` class, any other class with a reference to it could potentially break. For example, if the website class later contained the line `$this->jokesTable = false`, the `authorsTable` instance would break.

When using references like this, it's important to understand the possible side effects. References should generally be used sparingly.

Simplifying the List Controller Action

The following is now possible inside a controller:

```
$joke = $this->jokesTable->find('id', 123)[0];

echo $joke->getAuthor()->name;
```

Now that a joke is an *object*, we can pass the whole object into the template and read the author from there.

The `list` method currently looks like this:

```
public function list() {
  $result = $this->jokesTable->findAll();
   $totalJokes = $this->jokesTable->total();
  $jokes = [];
  foreach ($result as $joke) {
    $author = $this->authorsTable->find('id', $joke->authorId)[0];

    $jokes[] = [
      'id' => $joke->id,
      'joketext' => $joke->joketext,
      'jokedate' => $joke->jokedate,
      'name' => $author->name,
      'email' => $author->email,
      'authorId' => $author->id
    ];

  }

  return ['template' => 'jokes.html.php',
          'title' => 'Joke List',
          'variables' => [
          'jokes' => $jokes,
          'totalJokes' => $totalJokes,
          'userId' => $user->id ?? null
        ]
       ];
}
```

This code was necessary when using arrays, because for each joke, we needed to fetch the information about the joke, and the information about the author of that particular joke, then combine them into a single data structure.

However, now that we're using objects, this ugly copying of code is redundant. We can replace the above code with this much simpler code:

```
public function list() {
  $jokes = $this->jokesTable->findAll();

  $user = $this->authentication->getUser();
  $totalJokes = $this->jokesTable->total();

  return ['template' => 'jokes.html.php',
        'title' => 'Joke List',
        'variables' => [
          'jokes' => $jokes,
          'totalJokes' => $totalJokes,
          'userId' => $user->id ?? null
        ]
       ];
}
```

Because the `DatabaseTable` class now provides an array of *objects* rather than arrays, each `Joke` object has a `getAuthor` method, which returns the author of the joke.

Rather than fetching the corresponding author for each joke in the controller, we can now do it in the `jokes.html.php` template. Let's update the template to use the new objects rather than the arrays.

Example: Relationships-JokeObject[10]

```
<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>

<?php foreach ($jokes as $joke): ?>
<blockquote>
  <p>
  <?=htmlspecialchars($joke->joketext, ENT_QUOTES, 'UTF-8')?>

    (by <a href="mailto:<?=htmlspecialchars(
    $joke->getAuthor()->email,
    ENT_QUOTES,
          'UTF-8'
); ?>">
```

---

10. https://github.com/spbooks/phpmysql7/tree/Relationships-JokeObject

```
        <?=htmlspecialchars(
                $joke->getAuthor()->name,
                ENT_QUOTES,
            'UTF-8'
            ); ?></a> on
<?php
$date = new DateTime($joke->jokedate);

echo $date->format('jS F Y');
?>)

<?php if ($userId == $joke->authorId):
    ?>
  <a href="/joke/edit/<?=$joke->id?>">Edit</a>
  <form action="/joke/delete" method="post">
    <input type="hidden" name="id" value="<?=$joke->id?>">
    <input type="submit" value="Delete">
  </form>
<?php endif; ?>
  </p>
</blockquote>
<?php endforeach; ?>
```

For the fields from the `joke` table, this is fairly simple. We just change the syntax from an array to an object. For example, `$joke['joketext']` becomes `$joke->joketext`.

It's slightly more complicated where we want to read information about each joke's author. Before reading the author's email address, we need to fetch the author instance. To read the author's email, we previously used `$joke['email']`, which now becomes `$joke->getAuthor()->email`.

This actually fetches the author from within the template! Previously, when writing the controller, we had to anticipate exactly which variables were needed by the template.

Now, the controller just provides a list of jokes. The template can now read any of the values it needs, including information about the author. If we added a new column in the database—for example, a joke category—we could amend the template to show this value without needing to change the controller.

If you took this example to its extreme and implemented the relevant entity classes, you could fetch the corresponding manufacturer name on an online shop by chaining the relevant calls:

```
echo $customer->getOrders()[0]->getItems()[0]->getManufacturer()->name;
```

Once the relationships are modeled in entity classes, you can traverse the relationships with method calls, each call querying the database for the relevant related records. By putting these calls in the template, the code in the controller doesn't need to be aware of exactly what data is required by the template.

Tidying Up

Now that we've changed the way the `DatabaseTable` class works, we've broken the edit joke page. To fix it, open up the template `editjoke.html.php` and replace the array syntax with object syntax for accessing the joke's properties.

Example: Relationships-EditJoke[11]

```
<?php if (empty($joke->id) || $userId == $joke->authorId):
    ?>
<form action="" method="post">
  <input type="hidden" name="joke[id]" value="<?=$joke->id ?? ''?>">
    <label for="joketext">Type your joke here:</label>
    <textarea id="joketext" name="joke[joketext]" rows="3" cols="40">
      <?=htmlspecialchars($joke->joketext ?? '', ENT_QUOTES, 'UTF-8')?>
    </textarea>
    <input type="submit" name="submit" value="Save">
</form>
<?php else:
        ?>

<p>You may only edit jokes that you posted.</p>
```

11. https://github.com/spbooks/phpmysql7/tree/Relationships-EditJoke

```
<?php endif; ?>
```

Finally, make the same change in the controller's edit method so that `'userId' => $author['id'] ?? null` becomes `'userId' => $author->id ?? null`.

We’ve now got an almost entirely object-oriented website! All the entities have their own class, and we can add any methods we like to each entity class.

Caching

You might have noticed a potential performance problem with the `Joke` entity class. The `getAuthor` method looks like this:

```
public function getAuthor() {
  return $this->authorsTable->find('id', $this->authorId)[0];
}
```

Although this works fine, it’s unnecessarily slow. Each time this `getAuthor` method is called, it will send the same query to the database and retrieve the same result. The following code will send three queries to the database:

```
echo $joke->getAuthor()->name;
echo $joke->getAuthor()->email;
echo $joke->getAuthor()->password;
```

Querying the database is considerably slower than just reading a value from a variable. Each time a query is sent to the database, it will slow down the page’s speed slightly. Although each query adds only a tiny overhead, if this is done inside a loop on a page, it can cause a noticeable slowdown.

To avoid this problem, we can fetch the author object once, then use the existing instance:

```
$author = $joke->getAuthor();
```

```
echo $author->name;
echo $author->name;
echo $author->password;
```

By doing this, we avoid sending three queries to the database, because `getAuthor` is only called once. This method works, but it's rather crude. You have to remember to implement this technique, and on a larger website you'd have to mentally keep track of all the places you'd need to do this.

Instead, it's better to implement a technique called **transparent caching**. The term **caching** refers to storing some data for quicker access later on, and the technique I'm about to show you is called *transparent caching*, because the person using the class doesn't even need to know it's happening!

To implement caching, add a property to the `Joke` entity class to store the author between method calls:

```
class Joke {
  // …
  public string $joketext;
  private ?object $author;

  // …
```

The `?object` type hint means that the variable can either store an object or `null`. Then, in the `getAuthor` method, we can add logic that will do the following:

- check to see if the `author` class variable has a value
- if it's empty, fetch the author from the database and store it in the class variable
- return the value stored in the `author` variable

Example: Relationships-Cached[12]

---

12. https://github.com/spbooks/phpmysql7/tree/Relationships-Cached

```
public function getAuthor() {
  if (empty($this->author)) {
    $this->author = $this->authorsTable->find('id', $this->authorId)[0];
  }

  return $this->author;
}
```

With this simple `if` statement in place, the database will only be queried the first time the `getAuthor` method is called on any given joke instance. Subsequent calls will use the value stored in the `$this->author` variable.

Now, the following code will only send a single query to the database:

```
echo $joke->getAuthor()->name;
echo $joke->getAuthor()->email;
echo $joke->getAuthor()->password;
```

By solving the potential performance issue inside the class, it no longer matters how it's used externally. There will only ever be a single query for each instance of the class.

Joke Categories

Now that you know how to add relationships between different tables and model them using entity classes, let's add a new relationship.

At the moment, we have a single list of jokes. As we only have half a dozen jokes on the website, this works fine. But moving forward, as more people register for the website and post jokes, the joke list page will keep getting longer!

People viewing the website may want to view a specific type of joke—such as *programming jokes*, *knock-knock jokes*, *one-liners*, *puns*, and so on.

The most obvious way to achieve this is the way we modeled the relationship between jokes and authors: create a `category` table to list the different

categories, then create a `categoryId` column in the `joke` table to allow each joke to be placed in a category.

However, a joke may fall into more than one category.

Before modeling the relationship, let's add a new form that allows creating new categories and storing them in the database. It's been a while since we've added a new page to the website, and we've made a few changes since we last did, so I'll go through this in detail.

Firstly, let's create the table to store the categories:

```
CREATE TABLE `ijdb`.`category` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(255) NULL,
  PRIMARY KEY (`id`));
```

Either use the SQL code above or use MySQL Workbench to create the `category` table with two columns: `id` and `name`. `id` is the primary key, so make sure it's `AUTO_INCREMENT`, and the type should be `VARCHAR`.

Create a new controller called `Category` in `Ijdb/Controllers/Category.php`. This controller will need access to a `DatabaseTable` instance that allows interacting with the new `category` table:

```
<?php
namespace Ijdb\Controllers;

class Category {
    public function __construct(private \Ninja\DatabaseTable $categoriesTable) {

    }
}
```

Like the controller for jokes, we'll need several actions: `list` to display a list of categories, `delete` to delete a category, `edit` to display the add/edit form, and `editSubmit` to handle the form submission.

Let's add the form first, by creating a new `editcategory.html.php` template. As with the add/edit joke template, this will handle both adding and editing categories, so we need to remember to include the hidden input and print the existing value into the name box if it's set.

Relationships-AddCategory[13]

```
<form action="" method="post">
  <input type="hidden" name="category[id]" value="<?=$category->id ?? ''?>">
  <label for="categoryname">Enter category name:</label>
  <input type="text" id="categoryname" name="category[name]"
  ↪value="<?=$category->name ?? ''?>" />
  <input type="submit" name="submit" value="Save">
</form>
```

Next, we'll add the `edit` method to the new `Category` class:

```
public function edit(?string $id = null) {

  if (isset($id)) {
    $category = $this->categoriesTable->find('id', $id)[0];
  }

  return ['template' => 'editcategory.html.php',
    'title' =>  'Edit Category',
    'variables' => [
      'category' => $category ?? null
    ]
  ];
}
```

Open up the `Website` class. There are two things to do here:

- In the constructor, create the `categoriesTable` instance of `DatabaseTable` linking to the new `category` table.
- Create the `Category` controller we just wrote the code for and return it in

13. https://github.com/spbooks/phpmysql7/tree/Relationships-AddCategory

the `getController()` when `$controllerName` is `category`, placing the edit page on the URI `/category/edit/`.

Once these changes are in place, your website class will look like this:

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
    private ?\Ninja\DatabaseTable $jokesTable;
    private ?\Ninja\DatabaseTable $authorsTable;
    private ?\Ninja\DatabaseTable $categoriesTable;
    private \Ninja\Authentication $authentication;

    public function __construct() {
        $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ↪'ijdbuser', 'mypassword');

        $this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id',
        ↪'\Ijdb\Entity\Joke', [&$this->authorsTable]);

        $this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id',
        ↪'\Ijdb\Entity\Author', [&$this->jokesTable]);

        $this->categoriesTable = new \Ninja\DatabaseTable($pdo, 'category',
        ↪'id');

        $this->authentication = new \Ninja\Authentication($this->authorsTable,
        ↪'email', 'password');

    }

    public function getDefaultRoute(): string {
        return 'joke/home';
    }

    public function getController(string $controllerName): ?object {
        if ($controllerName === 'joke') {
            $controller = new \Ijdb\Controllers\Joke($this->jokesTable,
            ↪$this->authorsTable, $this->authentication);
        }
        else if ($controllerName === 'author') {
            $controller = new \Ijdb\Controllers\Author($this->authorsTable);
        }
```

```
        else if ($controllerName === 'category') {
            $controller = new \Ijdb\Controllers\Category($this->categoriesTable);
        }
        else {
            $controller = null;
        }

        return $controller;
    }
}
```

Notice that I haven't specified an entity class for the `category` table. Unless we need to add some functionality to it, it can use the `stdClass` class that we set as the default.

While we're here, the `if … else` statement in the `getController` method is starting to become a little long and difficult to manage.

When faced with this kind of problem, it's often better to remove the `if` statement entirely. You might be wondering how to do this kind of decision-making without an `if` statement, but it's actually very straightforward and requires even less code.

To solve this without an `if`, we can move the `if … else` into an array:

```
public function getController(string $controllerName): ?object {

  $controllers = [
    'joke' => new \Ijdb\Controllers\Joke($this->jokesTable, $this->authorsTable,
    ↪$this->authentication),
    'author' => new \Ijdb\Controllers\Author($this->authorsTable),
    'login' => new \Ijdb\Controllers\login($this->authentication),
    'category' => new \Ijdb\Controllers\Category($this->categoriesTable)
  ];

  return $controllers[$controllerName] ?? null;
}
```

This creates an array of all the possible controllers, then looks up the

controller by its name—which is a lot simpler than a long `if … else` statement. To add new controllers, we simply add an entry to the array with the name and the corresponding controller object.

Processing Power vs Memory

Using an array like this requires less processing, but it will use more memory. All the classes must be loaded into memory, and objects must be created up front for the array.

Loading classes has very little overhead as long as PHP's OPcache is turned on. The OPcache stores parsed code in a single file for all classes in the project. In the old days of PHP, a `require` statement (even one triggered by an autoloader) caused PHP to look the file up on disk—a relatively slow process. This is no longer the case.

However, constructing all three controllers will use slightly more memory. Ultimately, you have to choose between additional processing power or additional memory usage. Since both are going to be negligible, I prefer the latter, as the code is cleaner.

If you've followed the steps above, visit `https://v.je/category/edit` and you should see the form. To get it working, add the `editSubmit` method to add a category to the database when the form is submitted.

Example: Relationships-AddCategory-Save[14]

```
public function editSubmit() {
  $category = $_POST['category'];

  $this->categoriesTable->save($category);

  header('location: /category/list');
}
```

---

14. https://github.com/spbooks/phpmysql7/tree/Relationships-AddCategory-Save

If you test the form and press **Save**, you'll see an error message, because we haven't built the list page yet. However, you can check the form is working by selecting all the records from the table in MySQL Workbench.

Once a category has been added, you should even be able to edit it by visiting `https://v.je/category/edit/1`.

At this point, it's worth momentarily stepping back and considering the benefit of the classes and framework we've built. With fairly little effort, we've created a form that allows inserting data into the database, loading a record into it and editing it. Think back to Chapter 4, when we were manually writing `INSERT` and `UPDATE` queries, and consider how much more code you would have needed to write with that approach!

List Page

We can also add the list page with fairly little code. Firstly, let's create a `categories.html.php` template that loops through a list of categories and displays an **Edit**/**Delete** option for each of them.

Example: Relationships-ListCategories[15]

```
<h2>Categories</h2>

<a href="/category/edit">Add a new category</a>

<?php foreach ($categories as $category): ?>
<blockquote>
  <p>
  <?=htmlspecialchars($category->name, ENT_QUOTES, 'UTF-8')?>

  <a href="/category/edit?id=<?=$category->id?>">Edit</a>
  <form action="/category/delete" method="post">
    <input type="hidden" name="id" value="<?=$category->id?>">
    <input type="submit" value="Delete">
  </form>
```

15. https://github.com/spbooks/phpmysql7/tree/Relationships-ListCategories

```
  </p>
</blockquote>

<?php endforeach; ?>
```

Here's the controller action in `Category.php`:

```
public function list() {
  return ['template' => 'categories.html.php',
    'title' => 'Joke Categories',
    'variables' => [
      'categories' => $this->categoriesTable->findAll()
    ]
  ];
}
```

The **Edit** link from the template already works. Add the route and controller action for the **Delete** button to finish the category management pages.

Example: Relationships-DeleteCategory[16]

```
public function deleteSubmit() {
  $this->categoriesTable->delete('id', $_POST['id']);

  header('location: /category/list');
}
```

Assigning Jokes to Categories

Now that categories can be added to the website, let's add the ability to assign a joke to a category.

As I mentioned previously, the *simplest* approach would be to have a `<select>` box on the add joke page, which sets a `categoryId` column in the `joke` table.

---

16. https://github.com/spbooks/phpmysql7/tree/Relationships-DeleteCategory

However, this approach is the least flexible. It's possible that a joke will fall into more than one category. Instead, we're going to model this relationship using a **join table**—that is, a table with just two columns.

In our case, the columns will be `jokeId` and `categoryId`.

Let's create the following table using either MySQL Workbench or by running this query:

```
CREATE TABLE `ijdb`.`joke_category` (
  `jokeId` INT NOT NULL,
  `categoryId` INT NOT NULL,
  PRIMARY KEY (`jokeId`, `categoryId`));
```

Note that both the `jokeId` and `categoryId` columns are the primary key. As I explained in Chapter 4, this is to prevent the same joke being added to a category twice.

Before continuing, let's add some categories to the database. Using the form at `https://v.je/category/edit`, add the categories "programming jokes" and "one-liners".

Currently, we have three jokes, each of which falls into the "programming jokes" category:

> *How many programmers does it take to screw in a lightbulb? None. It's a hardware problem.*
> *Why did the programmer quit his job? He didn't get arrays.*
> *Why was the empty array stuck outside? It didn't have any keys.*

Here are some more jokes that fit in the two categories:

> *Bugs come in through open Windows. (“programming jokes” and “one-liners” categories)*
> *How do functions break up? They stop calling each other. (“programming jokes” only)*
> *You don’t need any training to be a litter picker, you pick it up as you go. (“one-liners” only)*
> *Venison’s dear, isn’t it? (“one-liners” only)*
> *It’s tricky being a magician. (“one-liners” only)*

Don’t add these just yet. We’ll amend the add joke page to allow selecting categories and then add the jokes.

Instead of using a `<select>` box, we’ll use a series of checkboxes to allow users to select which categories a joke falls into.

As the `Joke` controller will now need to be able to pass a list of categories to the `editjoke.html.php` template, let’s amend the controller with a new constructor argument and class variable:

```
<?php
namespace Ijdb\Controllers;
use \Ninja\DatabaseTable;
use \Ninja\Authentication;

class Joke {
  public function __construct(private DatabaseTable $jokesTable, private
  ↪DatabaseTable $authorsTable, private DatabaseTable $categoriesTable,
  ↪private Authentication $authentication) {
  }

  // …
```

Aesthetic Choices

For consistency, I've placed the argument before the `Authentication` argument. There's no practical reason for this; it's an entirely aesthetic choice to put all the `DatabaseTable` instances together.

Now we should pass in the `DatabaseTable` instance of the `categoriesTable` when the `Joke` controller is instantiated in the `Websites` class:

```
'joke' => new \Ijdb\Controllers\Joke($this->jokesTable, $this->authorsTable,
  ➥$this->categoriesTable, $this->authentication),
```

In the `edit` method, pass the list of categories to the template:

```
public function edit($id = null) {
  $author = $this->authentication->getUser();
  $categories = $this->categoriesTable->findAll();

  if (!empty($id))) {
    $joke = $this->jokesTable->find('id', $id);
  }
  else {
    $joke = null;
  }

  $title = 'Edit joke';

  return ['template' => 'editjoke.html.php',
    'title' => $title,
    'variables' => [
      'joke' => $joke ?? null,
      'userId' => $author->id ?? null,
      'categories' => $categories
    ]
  ];
}

}
```

The next step is to set up the list of categories in the `editjoke.html.php` template and create a checkbox for each:

```
<form action="" method="post">
  <input type="hidden" name="joke[id]" value="<?=$joke->id ?? ''?>">
  <label for="joketext">Type your joke here:</label>
  <textarea id="joketext" name="joke[joketext]" rows="3" cols="40">
  ↪<?=$joke->joketext ?? ''?></textarea>

  <p>Select categories for this joke:</p>
  <?php foreach ($categories as $category): ?>
  <input type="checkbox" name="category[]" value="<?=$category->id?>" /> <label>
  ↪<?=$category->name?></label>

  <?php endforeach; ?>

  <input type="submit" name="submit" value="Save">
</form>
```

The code here should be familiar by now, but I'll quickly go over the additions I've made:

- `<?php foreach ($categories as $category): ?>` : this loops over each of the categories.
- `<input type="checkbox" name="category[]" value="<?=$category->id?>" />` : this creates a checkbox for each category with the `value` property set to the ID of the category.
- `name="category[]"` : by setting the name of the checkbox to `category[]` , an array will be created when the form is submitted. For example, if you checked the checkboxes with the values `1` and `3` , the variable `$_POST['category']` would contain an array with the values 1 and 3 (`['1', '3']`).

If you try the code above, the formatting will look strange. Add the following CSS to `jokes.css` to fix it:

```
form p {clear: both;}
input[type="checkbox"] {float: left; clear: left; width: auto; margin-right:
↪10px;}
input[type="checkbox"] + label {clear: right;}
```

Now that the categories are listed and selectable on the add joke page, we need to change the `editSubmit` method to handle new data from the form submission.

It's important to understand what needs to happen here. When the form is submitted, an array of the IDs of the checked categories will be sent as a `$_POST` variable. Each category ID and the ID of the new joke will then be written to the `joke_category` table.

Before we can add records to the `joke_category` table, we'll need a `DatabaseTable` instance for it. Add the class variable and create the instance in the `Website` class, a process you're now familiar with:

```
namespace Ijdb;
class JokeWebsite implements \Ninja\Website {
    private ?\Ninja\DatabaseTable $jokesTable;
    private ?\Ninja\DatabaseTable $authorsTable;
    private ?\Ninja\DatabaseTable $categoriesTable;
    private ?\Ninja\DatabaseTable $jokeCategoriesTable;
    private \Ninja\Authentication $authentication;

    public function __construct() {
        $pdo = new \PDO('mysql:host=mysql;dbname=ijdb;charset=utf8mb4',
        ↪'ijdbuser', 'mypassword');

        $this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id',
        ↪'\Ijdb\Entity\Joke', [&$this->authorsTable]);

        $this->authorsTable = new \Ninja\DatabaseTable($pdo, 'author', 'id',
        ↪'\Ijdb\Entity\Author', [&$this->jokesTable]);

        $this->categoriesTable = new \Ninja\DatabaseTable($pdo, 'category',
        ↪'id');
```

```
        $this->authentication = new \Ninja\Authentication($this->authorsTable,
        ↪'email', 'password');

        $this->jokeCategoriesTable = new \Ninja\DatabaseTable($pdo,
        ↪'joke_category', 'categoryId');

    }
```

We *could* pass the `jokeCategoriesTable` instance to the `Joke` controller. However, like we did with the `$author->addJoke` method, it's better to implement this using an object-oriented approach, where a category is assigned to a joke using this code:

```
$joke->addCategory($categoryId);
```

To do this, we'll need a `Joke` entity instance in the `editSubmit` method in the `Joke` controller. The code for the updated `editSubmit` method will look like this:

```
public function editSubmit() {
  $author = $this->authentication->getUser();

  if (!empty($id)) {
     $joke = $this->jokesTable->find('id', $id)[0];

    if ($joke->authorId != $author->id) {
      return;
    }
  }

  $joke = $_POST['joke'];
  $joke['jokedate'] = new \DateTime();

  $jokeEntity = $author->addJoke($joke);

  foreach ($_POST['category'] as $categoryId) {
    $jokeEntity->addCategory($categoryId);
  }

  header('location: /joke/list');
```

```
}
```

The important change to what we have at the moment is the `$jokeEntity = $author->addJoke($joke);` line.

The `addJoke` method, which currently doesn't have a return value, will need to return a `Joke` entity instance, which represents the joke that has just been added to the database.

As a first thought, a simple approach would be to fetch the joke from the `jokesTable` instance after it's been created, using the following process:

- take the data for the new joke from `$_POST`
- pass it to the `addJoke` method in the `Author` entity class
- retrieve the newly added joke from the database using a `SELECT` query (or the `find` method)

For example:

```
public function addJoke($joke) {

  $joke['authorId'] = $this->id;

  // Store the joke in the database
  $this->jokesTable->save($joke);

  // Fetch the new joke as an object
  return $this->jokesTable->find('id', $joke['id'])[0];
}
```

There are two problems with this:

- If a joke is being added rather than updated, `$joke['id']` won't be set, so we won't be able to find the corresponding joke unless it's an existing joke being edited.

- It adds additional overhead. We're actually making two trips to the

database—once to run an `INSERT` query to send the data about the new joke, and then a `SELECT` query to fetch that very same information back out of the database immediately afterwards! We already have the information in the `$joke` variable, so there's no need to fetch it from the database here.

Although it would be possible to create the `Joke` entity instance in the `addJoke` method, it makes more sense to place this functionality in the `DatabaseTable` class. Whenever `save` is called, it can return the relevant entity instance. By placing this logic inside the `save` method, any time any data is written to any database table, the `save` method will return an object representing that newly added record.

The `addJoke` method above should be changed to this:

```
public function addJoke($joke) {

  $joke['authorId'] = $this->id;

  return $this->jokesTable->save($joke);
}
```

This will just return the return value of the `save` method from the `DatabaseTable` class. That is, whatever the `save` method returns to the `addJoke` method will also be returned by the `addJoke` method.

We haven't looked at the code for this class for a long time, but open up the `DatabaseTable` class and find the `save` method:

```
public function save($record) {
    try {
       if (empty($record[$this->primaryKey]) {
         unset($record[$this->primaryKey]);
       }
       $this->insert($record);
    } catch (PDOException $e) {
       $this->update($record);
```

```
    }
}
```

This calls either the `insert` or `update` method. The first thing we need to do here is create an instance of the relevant entity class. For jokes, it will be the `Joke` class; for authors, it will be the `Author` class; and so on.

To create the relevant entity, we can use this code:

```
$entity = new $this->className(...$this->constructorArgs);
```

This looks complicated. There are lots of variables in this line of code. It's important to understand what's happening here, so I'll show you how this works.

The instance is created using `new $this->className`. Importantly, PHP evaluates `$this->className` prior to trying to create an object. If `$this->className` contains `\Ijdb\Entity\Joke`, the code evaluated will be `new \Ijdb\Entity\Joke(...)`.

Then, the `$this->constructorArguments` is unpacked using the *argument unpacking operator* we saw in Chapter 9. This takes the constructor arguments required for an entity instance and converts them into arguments.

Once this code has run, the `$entity` variable will store an object of the type defined in `$this->className`.

This line can be placed inside the `save` method, along with a line to return the newly created entity object:

```
public function save($record) {
  $entity = new $this->className(...$this->constructorArgs);
  try {
     if (empty($record[$this->primaryKey]) {
       unset($record[$this->primaryKey]);
```

```
    }
    $this->insert($record);
  } catch (PDOException $e) {
    $this->update($record);
  }

  return $entity;
}
```

With those lines in place, the `save` method will instantiate the relevant, empty entity class. The next stage is writing the data that was sent to the database to the class. This can be done with a simple `foreach`:

```
public function save($record) {
  $entity = new $this->className(...$this->constructorArgs);
  try {
    if (empty($record[$this->primaryKey]) {
      unset($record[$this->primaryKey]);
    }
    $this->insert($record);
  } catch (PDOException $e) {
    $this->update($record);
  }

  foreach ($record as $key => $value) {
    if (!empty($value)) {
      if ($value instanceof \DateTime) {
        $value = $value->format('Y-m-d H:i:s');
      }
      $entity->$key = $value;
    }
  }
  return $entity;
}
```

The important line here is `$entity->$key = $value;`. Each time `foreach` iterates, the `$key` variable is set to the column name—for example, `joketext`—and the `$value` variable is set to the value being written to that column. By using the `$key` variable after the object access operator (`->`), it will write to the property with the name of the column.

The `if (!empty($value))` check is in place to prevent values that are already set on the entity being overwritten with `null`. With this change in place, the `$entity` variable contains a fully constructed object after a record is saved.

Finally, `DateTime` instances are converted back into strings for storing in the entity.

It's equivalent to manually setting each property on the object like so:

```
$entity->joketext = 'Why did the empty array get stuck outside? It didn\'t have
↪ any keys';

$entity->authorId = 1;

$entity->jokedate = '2018-06-22';
```

However, using the generic approach above, the property names and corresponding values come from the `$record` array. Any values in `$record` will be copied into the `$entity` object.

Converting Arrays to Objects

This approach is a common method of converting an array to an object.

With the `foreach` in place, writing the values to the entity object, the `save` method will return the object with all the values that were passed as an array to the method.

This will work fine for records being updated, as the `$record` variable will include keys for all the columns in the database table.

A newly created record, however, won't have the primary key set. When a joke is added, we pass the `save` method values for the `joketext`, `jokedate` and `authorId` columns, while the `id` is left blank.

The `id` primary key is actually created by MySQL inside the database. For `INSERT` queries, we'll need to read the value from the database immediately after the record has been added.

Luckily, the PDO library provides a very simple method of doing this. After an `INSERT` query has been executed by the database, you can call the `lastInsertId` method on the `PDO` instance to read the ID of the last record inserted.

To implement this, let's amend the `insert` method in our `DatabaseTable` class to return the last insert ID:

```
private function insert($values) {
    $query = 'INSERT INTO `' . $this->table . '` (';

    foreach ($values as $key => $value) {
        $query .= '`' . $key . '`,';
    }

    $query = rtrim($query, ',');

    $query .= ') VALUES (';

    foreach ($values as $key => $value) {
        $query .= ':' . $key . ',';
    }

    $query = rtrim($query, ',');

    $query .= ')';

    $values = $this->processDates($values);

    $stmt = $this->pdo->prepare($query);
    $stmt->execute($values);

    return $this->pdo->lastInsertId();
}
```

The only amendment here is the final line, returning the last insert ID. Now, the

`save` method can read this value and set the primary key on the created entity object:

```
public function save($record) {
  $entity = new $this->className(...$this->constructorArgs);
  try {
     if (empty($record[$this->primaryKey]) {
       unset($record[$this->primaryKey]);
     }
     $insertId = $this->insert($record);

     $entity->{$this->primaryKey} = $insertId;
  } catch (PDOException $e) {
     $this->update($record);
  }

  foreach ($record as $key => $value) {
    if (!empty($value)) {
      if ($value instanceof \DateTime) {
        $value = $value->format('Y-m-d H:i:s');
      }
      $entity->$key = $value;
    }
  }
  return $entity;
}
```

As with the other methods in the `DatabaseTable` class, the name of the primary key column is stored in the `primaryKey` class variable. This column, and the newly generated value, are stored in the entity object. I've used the shorthand method with braces to set the primary key, but it's the same as this:

```
$insertId = $this->insert($record);

$primaryKey = $this->primaryKey;

$entity->$primaryKey = $insertId;
```

The `save` method is now complete. Any time the `save` method is called, it will return an entity instance representing the record that's just been saved.

Assigning Categories to Jokes

This functionality is required to enable the `joke` controller to assign categories to a joke instance. We can amend the `editSubmit` method in the `Joke` controller to look like this:

```
public function editSubmit() {
  $author = $this->authentication->getUser();

  if (!empty($id)) {
    $joke = $this->jokesTable->find('id', $id)[0];

    if ($joke->authorId != $author->id) {
      return;
    }
  }

  $joke = $_POST['joke'];
  $joke['jokedate'] = new \DateTime();

  $jokeEntity = $author->addJoke($joke);

  foreach ($_POST['category'] as $categoryId) {
    $jokeEntity->addCategory($categoryId);
  }

  header('location: /joke/list');
}
```

Now that `$author->addJoke($joke);` returns a `Joke` entity object, we can call methods on an entity representing the record that has just been inserted. In this case, `$jokeEntity->addCategory($categoryId);` can be used to assign a category to the joke that was just added to the database.

Of course, for this to work we'll need to make some changes to the `Joke` entity class.

As the `addCategory` method will write a record to the new `joke_category` table, it will need a reference to the `jokeCategoriesTable` `DatabaseTable`

instance. You know the drill here: add a class variable and constructor argument:

```
namespace Ijdb\Entity;

class Joke {
    public int $id;
    public int $authorId;
    public string $jokedate;
    public string $joketext;
    private ?object $author;

    public function __construct(private ?\Ninja\DatabaseTable $authorsTable,
    ↪private ?\Ninja\DatabaseTable $jokeCategoriesTable) {
    }

    public function getAuthor() {
      if (empty($this->author)) {
        $this->author = $this->authorsTable->find('id', $this->authorId)[0];
      }

      return $this->author;
    }
}
```

Then amend `Ijdb\JokeWebsite` to provide the instance as an argument to the constructor of the `$jokesTable` instance:

```
$this->jokesTable = new \Ninja\DatabaseTable($pdo, 'joke', 'id',
  ↪'\Ijdb\Entity\Joke', [&$this->authorsTable, &$this->jokeCategoriesTable]);
```

By adding `&$this->jokeCategoriesTable` to the array passed as the fifth argument, each time an instance of `\Ijdb\Entity\Joke` is created inside the `$jokesTable` instance, the constructor will be called with the `authorsTable` and `jokeCategoriesTable` instances.

Next, add the `addCategory` method to the `Joke` entity class.

Example: Relationships-AssignCategory[17]

```
public function addCategory($categoryId) {
  $jokeCat = ['jokeId' => $this->id, 'categoryId' => $categoryId];

  $this->jokeCategoriesTable->save($jokeCat);
}
```

This code is fairly simple. The first line creates an array that represents the record to be added. The `jokeId` is the `id` of the joke that we're adding the category to (which comes from `$pdo->lastInsertId()`), and the `categoryId` comes from the argument.

With this code in place, whenever a joke is added to the website, it's assigned to the categories that were checked.

Go ahead and add the jokes I supplied earlier, or your own, and verify that the records have been added to the `joke_category` `join` table by selecting the records from the table in MySQL Workbench.

Displaying Jokes by Category

Now that we have jokes in the database that are assigned to a category, let's add a page that allows selecting jokes by category.

On the joke list page, let's add a list of categories to allow filtering of the jokes.

The first part is fairly simple: we need a list of categories as links on the joke list page. This involves two fairly simple steps.

Firstly, amend the `list` action to pass a list of categories to the template:

```
public function list() {
  $jokes = $this->jokesTable->findAll();

  $title = 'Joke list';
```

---

17. https://github.com/spbooks/phpmysql7/tree/Relationships-AssignCategory

```
  $totalJokes = $this->jokesTable->total();

  $author = $this->authentication->getUser();

  return ['template' => 'jokes.html.php',
    'title' => $title,
    'variables' => [
      'totalJokes' => $totalJokes,
      'jokes' => $jokes,
      'userId' => $author->id ?? null,
      'categories' => $this->categoriesTable->findAll()
    ]
  ];
}
```

Secondly, loop through the categories in the `jokes.html.php` template and create a list with links for each one:

```
<ul class="categories">
  <?php foreach ($categories as $category): ?>
    <li><a href="/joke/list/<?=$category->id?>"><?=$category->name?></a><li>
  <?php endforeach; ?>
</ul>
```

To make it look a little nicer, you can also add a containing `<div>` and a `<div>` around the list of jokes:

```
<div class="jokelist">

<ul class="categories">
  <?php foreach ($categories as $category): ?>
    <li><a href="/joke/list?/<?=$category->id?>"><?=$category->name?></a><li>
  <?php endforeach; ?>
</ul>

<div class="jokes">

<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>

<?php foreach ($jokes as $joke): ?>
```

```
    // …
<?php endforeach; ?>

</div>
```

Then apply the following CSS:

```
.jokelist {display: table;}
.categories {display: table-cell; width: 20%; background-color: #333; padding:
↪1em; list-style-type: none;}
.categories a {color: white; text-decoration: none;}
.categories li {margin-bottom: 1em;}
.jokelist .jokes {display: table-cell; padding: 1em;}
```

This example can be found in **Relationships-CategoryList**.

That's the list done, but the links currently don't do anything. Each link passes a third level to the URL. If you click on one of the new category links, you'll see the page you visit is `/jokes/list/1` or similar.

You've likely already worked out what we need to do now. We need to use third URL parameter to filter the jokes that are associated with the selected category.

For example, if the category with the ID `1` is *programming jokes* and the user visits `/jokes/list/1`, it should show only jokes in the *programming jokes* category. If the user visits `/jokes/list/2` and the category with the ID of `2` is *one-liners*, only jokes in the *one-liners* category should be displayed.

Back in Chapter 9, we amended the `edit` method to use an argument that can be provided with the ID of the joke being edited. We've already set up the `EntryPoint` class to convert any URL into `/[controller]/[method]/[argument1]/[argument2]./...`, so to access the category in the method, we simply need to add the argument to the list method, remembering that we also want to be able to get a list of all jokes, so we'll need to give it a default value:

```
public function list($categoryId = null) {
```

When visiting `/joke/list/2`, the `$categoryId` argument will be set to `2`.

If we had modeled the relationship as a *one-to-one* relationship using a `categoryId` column in the `joke` table, this would be relatively simple. In the `list` controller action, we'd just amend the way the `$jokes` variable was set:

```
if (!empty($categoryId)) {
  $jokes = $this->jokesTable->find('categoryId', $categoryId);
}
else {
  $jokes = $this->jokesTable->findAll();
}
```

However, as we have a *many-to-many* relationship, it's not quite so simple. One option is to pass the `jokeCategoriesTable` to the controller query all the corresponding jokes from that:

```
if (!empty($categoryId)) {
  $jokeCategories = $this->jokeCategoriesTable->find('categoryId', $categoryId);

  $jokes = [];

  foreach ($jokeCategories as $jokeCategory) {
    $jokes[] = $this->jokesTable->find('id', $jokeCategory->jokeId)[0] ?? null;
  }
}
else {
  $jokes = $this->jokesTable->findAll();
}
```

In this example, we're getting a list of all the records from the `joke_cagegory` table for the chosen category by its `id`—such as `4`. This gives us a set of records, each with a `categoryId` and a `jokeId`.

In our example, the `categoryId` will always be `4`, because we've only selected the records from that category, but each record has a unique

`jokeId`. We then loop through all the records and find the relevant record from the `joke` table, adding each `joke` record to the `$jokes` array.

If you want to go ahead and test this approach for yourself, use the code above in the `list` method, create the `jokeCategoriesTable` class variable, and constructor argument, and pass the instance into the `Joke` controller from `IjdbRoutes`.

I haven't given you the code for this—because it's not a great solution. One of the most difficult parts of programming is placing code in the right place. The logic above is correct. It works, and it was fairly simple to work out. However, it would be better if we could get a list of jokes from a category like this:

```
$category = $this->categoriesTable->find('id', $categoryId);

$jokes = $category->getJokes();
```

Doing so would allow us to get a list of jokes from a category anywhere in the program, not just the `list` method.

You already know how to achieve this: we did the same thing with `$joke->getAuthor()`. In principle, this is the same.

We'll need a `Category` entity class that has access to the `jokesTable` instance, the `jokeCategoriesTable` instance, and that has a method called `getJokes`:

```
<?php
namespace Ijdb\Entity;

class Category {
    public $id;
    public $name;

    public function __construct(private ?\Ninja\DatabaseTable $jokesTable,
    ↪private ?\Ninja\DatabaseTable $jokeCategoriesTable) {
```

```
    }

    public function getJokes() {
        $jokeCategories = $this->jokeCategoriesTable->find('categoryId',
        ↪$this->id);

        $jokes = [];

        foreach ($jokeCategories as $jokeCategory) {
            $joke = $this->jokesTable->find('id', $jokeCategory->jokeId)[0] ??
            ↪null;
            if ($joke) {
                $jokes[] = $joke;
            }
        }

        return $jokes;
    }
}
```

Save this in `classes/Ijdb/Entity/Category.php`. You'll notice the code in the `getJokes` method is almost the same as the code I showed you earlier. The only differences are that it uses `$this->id` instead of `$categoryId` and returns the `$jokes` array. One minor change I've added is a safety precaution: the `if ($joke)` check ensures that a joke is only added to the `$jokes` array if it can be retrieved from the database, as it's possible the joke was deleted.

Amend the `Website` class to set the `categoriesTable` instance to use the new `Category` entity class and provide the two constructor arguments:

```
$this->categoriesTable = new \Ninja\DatabaseTable($pdo, 'category', 'id',
↪'\Ijdb\Entity\Category', [&$this->jokesTable, &$this->jokeCategoriesTable]);
```

Finally, use the new `getJokes` method to retrieve the jokes in the `list` controller action.

Example: Relationships-CategoryList2[18]

18. https://github.com/spbooks/phpmysql7/tree/Relationships-CategoryList2"

```
if (isset($categoryId)) {
  $category = $this->categoriesTable->find('id', $categoryId)[0];
  $jokes = $category->getJokes();
}
else {
  $jokes = $this->jokesTable->findAll();
}
```

Using this approach, any time you need a list of jokes that exist in a category, you can find the category and then use `$category->getJokes()`.

If you visit the joke list page of the website, you'll be able to click on the category links to filter the jokes.

Editing Jokes

We've got most of the functionality for placing jokes in categories, but you'll notice a problem if you try to edit a joke. In fact, there are two problems.

The first, and the most obvious, can be found by editing a joke: try to edit one of the jokes that's already in a category and you'll immediately notice that the boxes aren't checked.

To fix this, we need to amend the code that prints the checkboxes:

```
<p>Select categories for this joke:</p>
<?php foreach ($categories as $category): ?>
<input type="checkbox" name="category[]" value="<?=$category->id?>" /> <label>
↪<?=$category->name?></label>
<?php endforeach; ?>
```

To check a checkbox, you add the attribute `checked` to the `input element`:

```
<input type="checkbox" checked name="category[]" value="<?=$category->id?>" />
```

This is simple to add with an `if` statement to display `checked` if the joke is inside the category. The difficult part is determining if the joke is inside any

given category.

We'll also solve this in an object-oriented way. Let's add a method in the `joke` entity so we can use this:

```
if ($joke->hasCategory($category->id))
```

Add the `hasCategory` method to the `Joke` entity class:

```
public function hasCategory($categoryId) {
  $jokeCategories = $this->jokeCategoriesTable->find('jokeId', $this->id);

  foreach ($jokeCategories as $jokeCategory) {
    if ($jokeCategory->categoryId == $categoryId) {
      return true;
    }
  }
}
```

This works by finding all the categories that are associated with a joke, looping through them, and checking to see whether one of those matches a given `$categoryId`.

With that in place, we can use it in the `editjoke.html.php` template:

```
<p>Select categories for this joke:</p>
<?php foreach ($categories as $category): ?>

<?php if ($joke && $joke->hasCategory($category->id)): ?>
<input type="checkbox" checked name="category[]" value="<?=$category->id?>" />
<?php else: ?>
<input type="checkbox" name="category[]" value="<?=$category->id?>" />
<?php endif; ?>

<label><?=$category->name?></label>
<?php endforeach; ?>
```

If you go to edit a joke, the relevant category boxes will now be checked, solving the first problem.

The second problem is a little more subtle. If you edit a joke but don't change the categories, everything will appear to work. However, if you uncheck a box, the changes won't be saved!

Try editing one of the jokes and unchecking all the category boxes before pressing **Save**. When you go back to edit the joke, the boxes will still be checked.

The reason for this issue is a common one when dealing with checkboxes in this way. Although we have some logic that says, "If the box is checked, add a record to the `joke_category` table", we don't have anything to remove that record after it's been added and the checkbox has been unchecked.

We could use this process:

- loop through every single category
- check to see if the corresponding checkbook box wasn't checked
- if the box wasn't checked and there's a corresponding record, delete the record

We'd need do this check for every category, and it would take a fairly large amount of code to achieve.

Instead, a much simpler approach is to delete all the records from the `joke_category` table that are related to the joke we're editing, then apply the same logic as before: loop through the checked boxes and insert records for each category that was checked.

Admittedly, this isn't entirely efficient. If the joke is edited and the category checkboxes aren't changed, this will cause unnecessary deletion and reinsertion of identical data. However, it's still the simplest approach.

Our `DatabaseTable` class has a `delete` method that allows deleting records, but rather than calling that directly from the `editSubmit` method, we'll add a `clearCategories` method to the `Joke` entity class to simplify the process.

When called, it will remove all category associations from the relevant joke:

```
public function clearCategories() {
  $this->jokeCategoriesTable->delete('jokeId', $this->id);
}
```

When `$joke->clearCategories()` is called, it will remove every record from the `joke_category` table that represents the joke stored in `$joke`. Add this call to the `editSubmit` method in the `Joke` controller.

Example: Relationships-ChangeCategories[19]

```
public function editSubmit($id = null) {
  $author = $this->authentication->getUser();

  if (!empty($id)) {
    $joke = $this->jokesTable->find('id', $id)[0];

    if ($joke->authorId != $author->id) {
      return;
    }
  }

  $joke = $_POST['joke'];
  $joke['jokedate'] = new \DateTime();

  $jokeEntity = $author->addJoke($joke);

  $jokeEntity->clearCategories();

  foreach ($_POST['category'] as $categoryId) {
    $jokeEntity->addCategory($categoryId);
  }

  header('location: /joke/list');
}
```

Test it for yourself by editing a joke and unchecking some categories, then going back and adding them again. If you've followed the steps here, you'll be

---

19. https://github.com/spbooks/phpmysql7/tree/Relationships-ChangeCategories

able to change the categories as you'd expect.

You're now getting very familiar with programming in an *object-oriented* style. By moving this functionality into the relevant entity classes, it can be easily reused in other parts of the application.

User Roles

We've now got a fully functional jokes website where users can register, post jokes, edit/delete their own submissions and view jokes by category.

But what happens if someone posts something you want to delete, or you want to fix a spelling mistake in someone else's joke?

At the moment, you can't do that! There's a check in place that only allows authors to edit their own jokes. If someone else posts something, there's currently no way for you to amend it.

The website is also set up so that anyone can add new categories. It would be better if only you (the website owner) were able to do that.

This is a very common problem on websites, and it's generally solved with **access levels**, where different accounts can perform different tasks.

On our website, we would need at minimum the following access levels:

- **Standard users**: can post new jokes and edit/delete jokes they've posted.
- **Administrators**: can add/edit/remove categories, post jokes, and edit/delete jokes anyone has posted. They should also be able to promote other users to administrators.

The simplest way to do this is to have a column in the `author` table that represents the author's access level. A `1` in the column could represent a normal user, and a `2` could represent an administrator. It would then be easy to add a check on any page to determine whether or not the logged-in user

was an administrator:

```
$author = $this->authentication->getUser();

if ($author->accessLevel == 2) {
  // They're an administrator
}
else {
  // Otherwise, they're not
}
```

This method is very simple to understand, and we could even very easily abstract it to `if ($author->isAdmin())` to improve the readability.

This implementation of access levels is fine for small websites with only a few users, or when there's only one administrator.

However, on larger, real-world websites, you often need to give users different levels of access. For example, we may want someone to be able to add categories, but not be able to grant other people administrator access, or worse, revoke your administrator privileges so they have complete control of the website!

A more flexible approach is to give each user individual permissions for each action. For example, we could set a user up to be able to edit a category but not add an administrator.

For this website, we've already considered these permissions:

- edit other people's jokes
- delete other people's jokes
- add categories
- edit categories
- remove categories
- edit user access levels

Before we model this in the database, let's think about how we'd check these

in the existing code.

Our `Author` entity class could have a method called `hasPermission` that takes a single argument and returns `true` or `false`, depending on whether or not the user has a specific permission.

We could assign numbers to each of the permissions above, so that the following code could be used to check whether or not they have permission to edit other people's jokes:

```
if ($author->hasPermission(1))
```

`2` could represent deleting other users' jokes, `3` could determine whether or not members are allowed to add categories, and so on.

This is roughly what we want to do, but looking at the line of code above, it's really not clear what's happening. If you saw the code `$author->hasPermission(6)`, you'd have to go away and look up what `6` meant.

To make the code much easier to read, each value can be stored inside a **constant**—which, like a variable, is a label given to a value. The difference, however, is that a constant always has the same value. The value is set once at the beginning of the program and can't be changed.

In object-oriented programming, constants are defined in classes like so:

```
<?php
namespace Ijdb\Entity;

class Author {

  const EDIT_JOKE = 1;
  const DELETE_JOKE = 2;
  const LIST_CATEGORIES = 3;
  const EDIT_CATEGORY = 4;
  const DELETE_CATEGORY = 5;
```

```
  const EDIT_USER_ACCESS = 6;
```

Constant Conventions

By convention, constants are written in uppercase, with words separated by underscores. Although it's possible to use lowercase letters, it's almost universal in every programming language that constants are written like this!

I've defined the constants in the `Author` entity class, since the permissions are related to authors.

We'll also define a method called `hasPermission` in the class:

```
<?php
namespace Ijdb\Entity;

class Author {

  const EDIT_JOKE = 1;
  const DELETE_JOKE = 2;
  const LIST_CATEGORIES = 3;
  const EDIT_CATEGORY = 4;
  const DELETE_CATEGORY = 5;
  const EDIT_USER_ACCESS = 6;
  // …

  public function hasPermission(int $permission) {
    // …
  }
```

Before we write the code for this method, I'm going to show you how it will be used:

```
$author = $this->authentication->getUser();
```

```
if ($author->hasPermission(\Ijdb\Entity\Author::LIST_CATEGORIES)) {
  // …
}
```

Notice that the constant is prefixed with the namespace and class name, then a `::`. The `::` operator is used to access constants in a given class.

When accessing a constant, you don't need an instance to be created, as each instance would have the same value for the constant anyway.

There are two different places we'll need to implement this. The first is page-level access. As we did with the login check, a check can be done at the `EntryPoint` level like we did with the login check previously.

In Chapter 11, we added a `checkLogin` method to the `Website` class that looks like this:

```
public function checkLogin(string $uri): ?string {
    $restrictedPages = ['joke/edit', 'joke/delete'];

    if (in_array($uri, $restrictedPages) && !$this->authentication->
    ↪isLoggedIn()) {
        header('location: /login/login');
        exit();
    }

    return $uri;
}
```

We'll amend this to use permissions instead of a binary, *only-allowed-to-view-if-logged-in* check.

Relationships-PermissionsCheck[20]

```
public function checkLogin(string $uri): ?string {
    $restrictedPages = [
```

20. https://github.com/spbooks/phpmysql7/tree/Relationships-PermissionsCheck

```
                        'category/list' => \Ijdb\Entity\Author::LIST_CATEGORIES
                       ];

     if (isset($restrictedPages[$uri])) {
       if (!$this->authentication->isLoggedIn()
          || !$this->authentication->getUser()->hasPermission
          ↪($restrictedPages[$uri])) {
         header('location: /login/login');
         exit();
       }
     }

     return $uri;

 }
```

There are three changes here. Firstly, the `$restrictedPages` array now has keys and values. Instead of a list of pages that can't be viewed unless you're logged in, each page now has a corresponding permission.

The second change is a new `if` block: `if (isset($restrictedPages[$uri])) {`. If this page has a permissions check, the rest of the code is executed. Otherwise, pages without a permission set will be open to everyone.

The final change is the logic for checking whether someone can view the page. There are two checks—one to see if the user is logged in, and one to check if the user has the corresponding permission. The permission required for the current page is read using `$restrictedPages[$uri]` and is passed in to the `hasPermission` method we're soon going to add to the `Author` entity class. If the user isn't logged in, or doesn't have the relevant permission, they're redirected to the login page.

Since we haven't written the code for the `hasPermission` method in the `Author` entity class yet, any permission check will always return false (or more accurately, `null`, which is converted to `false` for the purposes of the `if` condition). Go ahead and try viewing the category list by visiting `https://v.je/category/list` and you'll be redirected to the login page.

Creating a Form to Assign Permissions

Before we can implement the *`hasPermission`* method, the website needs a page that allows assigning permissions to any given user and some changes to the database to store information about user permissions.

We'll need two pages—one that lists all the authors, so we can select the one we want to give permissions to, and a second that contains a form with checkboxes for each permission for the selected user.

To begin with, we'll leave these pages open for anyone to view, as there's currently no way to assign permissions to users.

Rather than adding a new controller, we'll use the *`Author`* controller that already exists and is used for handling changes to user accounts.

Author List

Let's create the list first. Add a method in the *`Author`* controller called *`list`* that fetches a list of all the registered users and passes them to the template:

```
public function list() {
  $authors = $this->authorsTable->findAll();

  return ['template' => 'authorlist.html.php',
    'title' => 'Author List',
    'variables' => [
      'authors' => $authors
    ]
  ];
}
```

And here's the *`authorlist.html.php`* template for listing the users:

```
<h2>User List</h2>

<table>
```

```
  <thead>
    <th>Name</th>
    <th>Email</th>
    <th>Edit</th>
  </thead>

  <tbody>
    <?php foreach ($authors as $author): ?>
    <tr>
      <td><?=$author->name;?></td>
      <td><?=$author->email;?></td>
      <td><a href="/author/permissions/<?=$author->id;?>">Edit Permissions</a>
      ➥</td>
    </tr>
    <?php endforeach; ?>
  </tbody>
</table>
```

If you visit `https://v.je/author/list`, you'll see the list of registered authors, each with a link for editing their permissions.

The **Edit Permissions** link goes to the `/author/permissions` page, and provides it with the `id` of the author whose permissions we want to change—the same approach we used for passing the `id` of the joke being edited to the edit joke page.

Edit Author Permissions

There's nothing on the edit permissions page yet, because we haven't created it. It's quite a simple page: it will display a checkbox for each permission in the system, and it will be checked if the author currently has that permission.

The template could look like this:

```
<input type="checkbox" value="1" <?php if ($author->hasPermission(EDIT_JOKE)) {
    echo 'checked';
} ?> Edit Jokes
<input type="checkbox" value="2" <?php if ($author->hasPermission(DELETE_JOKE)) {
    echo 'checked';
```

```
} ?> Delete Jokes
<input type="checkbox" value="3" <?php if ($author->hasPermission
↪(LIST_CATEGORIES)) {
    echo 'checked';
} ?> Add Categories
// etc.
```

This would work, but it requires storing the information about the permissions in two different places—the constants in the `Author` entity class, and in the template. We also need to write out all the HTML and PHP for each checkbox.

Like most cases when we find repetition like this, there's a much easier way!

The class already contains a list of permissions and their values in the class constants:

```
class Author {

  const EDIT_JOKE = 1;
  const DELETE_JOKE = 2;
  const LIST_CATEGORIES = 3;
  const EDIT_CATEGORY = 4;
  const DELETE_CATEGORY = 5;
  const EDIT_USER_ACCESS = 6;
  // …
```

Rather than repeating this information inside the template file by creating a checkbox for every permission, it would be better to read the list of available permissions from the class and loop through them to create the checkboxes. Then adding new permissions involves a single change: adding a constant to the class. With the approach above, you'd need to add the constant to the class and also add the checkbox to the template file.

You can read information about variables, methods and constants that are contained inside a class using a tool called **Reflection**. PHP makes this fairly simple. To **reflect** the `Author` entity class and read all its properties, you can use the following code:

```
$reflected = new \ReflectionClass('\Ijdb\Entity\Author');

$constants = $reflected->getConstants();
```

The `$constants` array will contain information about all the constants that are defined within the class. If you use `var_dump` to print out the contents of the `$constants` variable, you'll see an array with the constant names as the key and the constant values as the value:

```
array (size=6)
  'EDIT_JOKE' => int 1
  'DELETE_JOKE' => int 2
  'LIST_CATEGORIES' => int 3
  'EDIT_CATEGORY' => int 4
  'DELETE_CATEGORY' => int 5
  'EDIT_USER_ACCESS' => int 6
```

Reflection

Reflection can be a very powerful tool, and I've only scratched the surface of what you can do with it. For more information on Reflection, see the PHP manual[21].

By passing this array to the template, we can loop over the list of constants and create the checkboxes automatically.

Firstly, add this `permissions` method to the `Author` controller:

```
public function permissions($id = null) {

  $author = $this->authorsTable->find('id', $id)[0];

  $reflected = new \ReflectionClass('\Ijdb\Entity\Author');
  $constants = $reflected->getConstants();
```

21. http://php.net/manual/en/book.reflection.php

```
  return ['template' => 'permissions.html.php',
    'title' => 'Edit Permissions',
    'variables' => [
      'author' => $author,
      'permissions' => $constants
    ]
  ];
}
```

Secondly, change the template to loop over the `$permissions` array, which is a list of constants from the class:

```
<h2>Edit <?=$author->name?>'s Permissions</h2>

<form action="" method="post">

  <?php foreach ($permissions as $name => $value): ?>
  <div>
    <input name="permissions[]" type="checkbox" value="<?=$value?>"
    ↪<?php if ($author->hasPermission($value)): echo 'checked'; endif; ?> />
    <label><?=$name?>
  </div>
  <?php endforeach; ?>

  <input type="submit" value="Submit" />
</form>
```

Adding a constant to the class will now result in the checkbox for that constant appearing on the user permissions page.

Like we did with categories, I've made the checkbox list an array. If you go to edit one of the user's permissions, you'll see a checkbox for each of the constants in the `Author` class!

This makes future developments a lot easier. Once the constant has been added to the `Author` entity class, it will automatically appear on the list without us having to manually go and edit the template each time we want to add a new permission to the website.

The code for this can be found in **Example: Relationships-EditPermissions**[22].

A Challenge

Here's a challenge for you, which you should find easy.

See if you can make the text on the page look nicer. The uppercase constants with underscores don't look very nice! Try to convert **EDIT_CATEGORY** to **Edit Categories**.

Hint: You should use the PHP functions `str_replace`, `strtolower` and `ucwords`.

Setting Permissions

The next stage is storing the user's permissions once you press **Save**. Each user will have a set of permissions, and there are many different ways to represent this in the database.

We could do it in the same way we did it with categories: create a `user_permission` table with two columns, `authorId` and `permission`. Then we could write a record for each permission. A user with the `id` of `4` and the permissions `EDIT_JOKE`, `LIST_CATEGORIES` and `DELETE_CATEGORY` would have the following records:

```
authorId | permission
       4 |          1
       4 |          3
       4 |          5
```

You already know how to implement this. You'll need to create the table, the relevant `DatabaseTable` instance, and write the `permissionsSubmit` method to create a record in the `user_permission` table for each checked box. Finally, the `hasPermission` method in the `Author` entity class would look something like this:

22. https://github.com/spbooks/phpmysql7/tree/Relationships-EditPermissions

```
public function hasPermission($permission) {
  $permissions = $this->userPermissionsTable->find('authorId', $this->id);

  foreach ($permissions as $permission) {
    if ($permission->permission == $permission) {
      return true;
    }
  }
}
```

Before you implement this, I'm going to show you an alternative approach.

Storing Permissions in the Database

Imagine if we set up the `author` table with a column for each permission:

```
CREATE TABLE `author` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(255) DEFAULT NULL,
  `email` VARCHAR(255) DEFAULT NULL,
  `password` VARCHAR(255) DEFAULT NULL,
  `permissionEditJoke` TINYINT(1) NOT NULL DEFAULT 0,
  `permissionDeleteJoke` TINYINT(1) NOT NULL DEFAULT 0,
  `permissionListCatgories` TINYINT(1) NOT NULL DEFAULT 0,
  `permissionAddCatgory` TINYINT(1) NOT NULL DEFAULT 0,
  `permissionRemoveCategory` TINYINT(1) NOT NULL DEFAULT 0,
  `permissionEditUserAccess` TINYINT(1) NOT NULL DEFAULT 0,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB CHARSET=utf8;
```

Each column is `TINYINT(1)`, which means it can store a single bit. It can either be a `1` or `0`, and I've set it to default to `0`.

We could then set any author's permissions using an `UPDATE` statement. For the user I mentioned earlier with the `id` of `4`, this could be sent to the database like so:

```
UPDATE `author` SET `permissionEditJokes` = 1, `permissionListCategories` = 1,
↪ `permissionRemoveCategories` = 1 WHERE `id` = 4
```

It would be possible to name our checkboxes with the names of the columns and use them in the same way as any other field.

The advantage of this approach is that it's simpler. To find out if a user has a permission, you can just use this:

```
// Can this author edit jokes?
if ($author->permissionEditJoke == 1)
```

Or this:

```
// Can this author remove categories?
if ($author->permissionRemoveCategory == 1)
```

This is a nicer approach than a `join` table, as the same check using a `join` table would require querying the database for all the user's permissions, looping through the records, and checking to see whether the permission we're looking for has a corresponding record.

The downside to this approach is that, every time we add a permission to the website, we need to add a column to the table.

In principle, though, this is a lot simpler. It avoids a database table, and checking whether a user has a permission requires significantly less code and fewer database queries.

Each permission is a simple `1` or `0` in the database.

If you examined a user's record in MySQL Workbench, you might see something like `0 1 0 0 1 0 0` in the permissions columns.

A Crash Course in Binary

This sequence of ones and zeros looks a lot like binary. In fact, every number stored in a database is actually stored as a series of ones and zeros behind the scenes.

If you have an `INT` column and it stores the number `6` for a particular record, it will actually store the binary value `0110` on the hard disk. Everything stored on a computer is binary!

Each binary number is a little like a database column. Each one or zero is in a specific column, and that column represents a particular value.

In fact, with normal *decimal* numbers, when you see the value `2395`, you know that the `3` (since it's in the third column from the right) represents `300`.

The above number can actually be expressed like this:

```
2 x 1000 +
3 x  100 +
9 x   10
5 x    1
```

You're so familiar with this process that it's second nature, and you don't need to think about it. Binary works in the same way. The only differences are that only the digits one and zero are available, and the column numbers are different.

In both binary and decimal, numbers are created right to left. As an extra digit is added, it's added to the left-hand side. To add `300` to `27`, you add a `3` to the left. The same thing is true of binary: digits further to the left have a higher value.

In the case of the binary number `0110`, the third column from the right represents `4`, and the second column from the right represents `2`.

To translate the value of the binary number `0110` into decimal, you can do the same kind of calculation as long as you know the values of each column. In decimal, each column is multiplied by ten each time you move to the left. In binary, each column is multiplied by two.

To work out the total of `0110`, we can do the same calculation:

```
0 x 8 +
1 x 4 +
1 x 2 +
0 x 1
```

If you calculate this, you'll get 6. Each column is called a **bit**, and in this example we can describe the `8` bit as not being set (because it's set to zero) and the `4` bit as being *set* (because it's set to one).

See if you can convert the following binary numbers to decimal:

- 1000
- 0010
- 1010

(You can find the answers on CodePen[23].)

Be Bit-wise

You're probably wondering why any of this matters and what it has to do with user permissions.

In short, binary has nothing to do with permissions. But neither do `if` statements or checkboxes. All three are tools we can use to solve the problem of permissions.

What's useful is that you can use PHP (and almost every programming language) to inquire about whether or not any given integer has a 1 or 0 set for any of the bits that make up the number.

Bitwise Permissions

Rather than using a different database column to store each one or zero, we could use a single binary number to store those ones and zeros in a single column.

23. https://codepen.io/SitePoint/full/xxXbRjo

By assigning a column to a *permission*, a binary number can represent which permissions any user has:

```
EDIT_USER_ACCESS | DELETE_CATEGORY | EDIT_CATEGORY | LIST_CATEGORIES | DELETE_JOKE | EDIT_JOKE
              32 |              16 |             8 |               4 |           2 |         1
```

12-2. Binary permissions

The binary number `000001`, which has a `1` in the `EDIT_JOKE` column, would represent a user with `EDIT_JOKE` permissions.

`111111` would represent a user that had all the permissions, and `011111` would represent a user that had every permission apart from being able to edit the permissions of other users (`EDIT_USER_ACCESS`).

This process is identical to using multiple columns in a database, each with a one or a zero. We're just using a binary number to represent the same data. Rather than one column per bit, we can store multiple bits in a single `INT` column.

Let's convert the binary numbers to decimal: `000001` becomes `1`, `111111` becomes `63` and `011111` becomes `31`. We can easily store these numbers as integers in a database!

If someone has the permissions value `63`, we know they have all the permissions that are available.

Back to PHP

The difficult part is extracting the individual permissions. What if we want to know whether the user has the `EDIT_CATEGORY` permission? On the chart above, I've assigned the `8` bit to mean `EDIT_CATEGORY`. If a user has the permissions value `13`, it's not clear whether or not the `8` bit is set.

In a database with multiple columns, we can use `SELECT * FROM author WHERE id = 4 AND editCategories = 1` to determine whether the `editCategories`

column is set to `1` for a specific user—in this case, the user with the `id` of `4`.

Most programming languages, including PHP and MySQL, support something called **bitwise** operations. These allow you to inquire whether or not a specific bit is set in any integer. Using a single `permissions` column, the query above can be expressed as follows:

```
SELECT * FROM author WHERE id = 4 AND 8 & permissions
```

The clever part here is `AND 8 & permissions`. This uses the bitwise `&` operator to inquire whether the `8` bit is set in the number stored in the `permissions` column for that record.

The bitwise `&` operator is used to check which bits are set in both numbers provided. The `&` operator is actually a mathematical operation like `+` or `-` that takes two operands and produces an output. The output of the bitwise `&` operator is a list of bits that are set in both operands. For example:

```
0010 & 1111 = 0010
0111 & 1111 = 0111
1000 & 1111 = 1000
1100 & 1001 = 1000
0000 & 1111 = 0000
```

These same expressions work in decimal numbers in both PHP and MySQL:

```
2 & 6 // will calculate to 2
```

This says, “Which bits are set in both the number 2 and 6?”, and calculates the result of 2. If you’re wondering why, let’s do the same check with binary numbers:

```
0010 & 0110
```

The only bit that is set in both numbers is the second bit from the right, which represents the number 2. In PHP this expression will evaluate to true:

```
if (2 & 6 == 2) {

}
```

We can use this to store a list of permissions. If a user had the permissions `011111` (31 in decimal) and we wanted to know if they had the permission associated with `010000` (16 in decimal) we can use the expression `if (16 & 32 == 16)`

Because an `if` statement will evaluate to true on any non-zero integer, if any of the bits on one side of the expression are set in the number on the other, the if statement will evaluate to true. As such, these are equivalent:

```
if (2 & 6) {

}
//equivalent to
if (2 & 6 == 2) {

}
```

Be wary using this shorthand. It only works if one of the numbers represents a single bit (so any binary number where there is a single 1, such as `00100` or `10000`, but not `01100`).

Binary operations like this are actually fairly common in PHP! When you set the `error_reporting` variable in PHP to `E_WARNING | E_NOTICE`, what you're doing is setting the bits that represent warnings and notices. PHP will then internally check which bits are set when it encounters an error.

Internally, PHP will do something like this:

```
if (E_NOTICE & ini_get('error_reporting')) {
  display_notice($notice);
}
```

We can apply this to permissions. Imagine the `author` table had a column

called `permissions`: it's possible to determine whether an author has the permission `EDIT_CATEGORY` if we associate `EDIT_CATEGORY` with the bit `8` by using this code:

```
if (8 & $author->permissions) {

}
```

This code has the same problem I mentioned earlier: it's not clear exactly what's happening here to anyone looking at the code. Again, we could represent the bits as constants:

```
const EDIT_JOKE = 1;
const DELETE_JOKE = 2;
const LIST_CATEGORIES = 4;
const EDIT_CATEGORY = 8;
const DELETE_CATEGORY = 16;
const EDIT_USER_ACCESS = 32;
```

And we could write the permissions check like so:

```
// Does the author have the EDIT_CATEGORY permission?
if (EDIT_CATEGORY & $author->permissions) {

}

// Does the author have the DELETE_JOKE permission?
if (DELETE_JOKE & $author->permissions) {

}
```

If you don't understand all the binary theory I've presented above, it doesn't matter. You don't need to understand the underlying theory to understand what's happening here, and as you've seen, you never even need to deal with the individual numbers or binary value. As a programmer setting this up, you just need to associate the constants with the corresponding bits.

Storing Bitwise Permissions in the Database

Let's implement this on the website. Amend the `author` table by adding a column called `permissions`, and set it to `INT(64)` so we can store a maximum of 64 different permissions.

Change the constant values in the `Author` entity class, as I've done above. Adding a new permission means working out the next bit; just double the previous one.

We don't need to make any changes to the edit permissions form page, but we need to add the `permissionsSubmit` method and have it store the binary permissions in the database.

Before we do that, let's consider what will happen when the form is submitted. If you check the boxes labeled `EDIT_JOKE` and `DELETE_CATEGORY`, the variable `$_POST['permissions']` will be an array containing the numbers `1` and `16` (`[1, 16]`).

We want to convert those to the binary representation, where the `1` bit and `16` bits are set. It sounds difficult but, as you'll see very shortly, it really isn't!

The number we need to generate is `010001`, where bits `16` and `1` are set. Use your new binary knowledge to calculate the decimal version of this value, and you'll work out that this binary number represents 17.

All we need to do is add the numbers together!

Just to prove the theory, let's imagine that the boxes for `EDIT_JOKE`, `DELETE_JOKE`, `LIST_CATEGORIES` and `EDIT_USER_ACCESS` are ticked. When the form is submitted, we'd get the array `[1, 2, 4, 32]`.

The binary representation of those permissions is `100111`. If you work out what that is in decimal, you'll get `39`. Add together the values from the array `1 + 2 + 4 + 32` and you'll also get 39!

All we need to do to store the numbers in the database is add together each element in the `$_POST['permissions']` array.

PHP even includes a function called `array_sum` that can do exactly that.

The `permissionsSubmit` method in the `Author` controller can be written like this:

```
public function permissionsSubmit($id = null) {
  $author = [
    'id' => $_POST['id'],
    'permissions' => array_sum($_POST['permissions'] ?? [])
  ];

  $this->authorsTable->save($author);

  header('location: /author/list');
}
```

The line `'permissions' => array_sum($_POST['permissions'] ?? [])` is used to add all the values from the `$_POST['permissions']` array. However, if no boxes are ticked, `$_POST['permissions']` won't be set. The `??` operator is used to provide the `array_sum` operator with an empty array if there's nothing in the `$_POST['permissions']` variable.

That's it! The `permissionsSubmit` method is converting the checked boxes into a number, and that number's binary representation is how we're modeling the permissions of each user.

Join Table and Bitwise Approaches: Pros and Cons

When using the `join` table approach for categories, to cater for boxes that were unchecked, we specifically had to delete all the records and reinsert them each time the form was submitted. As the `permissions` column is a single number, if no boxes are checked, `array_sum` will return `0` and the `0` will be inserted into the database, avoiding the need to specifically handle unchecked boxes.

The final piece is the `hasPermission` method in the `Author` entity class. Add the `$permission` class variable. Then, to check if a user has a permission, we need a single line of code.

Example: Relationships-BinaryPermissions[24]

```
public function hasPermission(int $permission) {
  return $this->permissions & $permission;
}
```

There are several advantages to this approach over a `join` table. The first is performance: we don't need to query the database for the user's permissions. Secondly, this is considerably less code for both saving the form and checking whether the permission is set.

There are two downsides to this approach. The first downside is that it can be more difficult to understand if you're not familiar with the bitwise operator. However, bitwise operators are fairly common in PHP. They're used by the `PDO` library and for the various `php.ini` configuration settings, such as `error_reporting`, so it's a good idea to have a basic understanding of what they do. The second downside is that you're limited to 64 bits, because that's all a CPU can process. However, if you find yourself needing more, you can group the permissions into different columns, such as `jokePermissions` and `adminPermissions`.

Whether you choose to implement user roles as bitwise, like I have here, or whether you use a `join` table, is up to you. There are pros and cons to each approach. Personally, I prefer the shorter code and fewer database operations that bitwise operators offer.

Cleaning Up

There's a little tidying up left to do. Firstly, we need to add the permissions to the routes.

---

24. https://github.com/spbooks/phpmysql7/tree/Relationships-BinaryPermissions

Make sure you've granted your user account the permission `EDIT_USER_ACCESS` before making these changes, or you won't be able to change anyone's permissions!

Add the relevant permissions to each page of the website:

```
$restrictedPages = [
    'category/list' => \Ijdb\Entity\Author::LIST_CATEGORIES,
    'category/delete' => \Ijdb\Entity\Author::DELETE_CATEGORY,
    'category/edit' => \Ijdb\Entity\Author::EDIT_CATEGORY,
    'author/permissions' => \Ijdb\Entity\Author::EDIT_USER_ACCESS,
    'author/list' => \Ijdb\Entity\Author::EDIT_USER_ACCESS
   ];
```

This will prevent anyone who doesn't have the `EDIT_USER_ACCESS` permission from changing the permissions of other users.

Editing Other People's Jokes

The final two permissions are `EDIT_JOKE` and `DELETE_JOKE`, which determine whether the logged-in user can edit or delete a joke someone else has posted.

We can't do this with the `$routes` array, because the check isn't done there. The **Edit** link and **Delete** button are hidden in the template, and there are checks inside the `joke` controller.

Firstly, let's make the **Edit** link and **Delete** button appear on the list page for all jokes if you have the `EDIT_JOKE` or `DELETE_JOKE` permissions.

The relevant section of `jokes.html.php` looks like this:

```
<?php if ($userId == $joke->authorId) {
    ?>
  <a href="/joke/edit/<?=$joke->id?>">Edit</a>
  <form action="/joke/delete" method="post">
    <input type="hidden" name="id" value="<?=$joke->id?>">
    <input type="submit" value="Delete">
```

```
  </form>
<?php
} ?>
```

Now that we have different permissions for **Edit** and **Delete**, we'll need two separate `if` statements—one for the **Delete** button and one for the **Edit** link.

However, we can't do this with the `$userId` alone. Instead of passing in the `$userId` variable to the template, change the `list` method in the `Joke` controller to pass in the entire `$author` object that represents the logged-in user:

```
return ['template' => 'jokes.html.php',
  'title' => $title,
  'variables' => [
    'totalJokes' => $totalJokes,
    'jokes' => $jokes,
    'user' => $author, //previously 'userId' => $author->id
    'categories' => $this->categoriesTable->findAll()
  ]
];
```

The check in the template can now be amended so that the button and link are only visible to the person who posted the joke, or someone with the relevant permission:

```
<?php if ($user): ?>
  <?php if (empty($joke) || $user->id == $joke->authorId || $user->
  ↪hasPermission(\Ijdb\Entity\Author::EDIT_JOKE)): ?>
  <a href="/joke/edit/<?=$joke->id?>">Edit</a>
  <?php endif; ?>
  <?php if ($user->id == $joke->authorId || $user->hasPermission
  ↪(\Ijdb\Entity\Author::DELETE_JOKE)): ?>
  <form action="/joke/delete" method="post">
    <input type="hidden" name="id" value="<?=$joke->id?>">
    <input type="submit" value="Delete">
  </form>
  <?php endif; ?>
<?php endif; ?>
```

This is a lot more complicated, as there are now three `if` statements. I've added `if ($user)` around the entire block, because the `$user` variable may be empty if no one is logged in.

The following two use a logical `or` to determine whether the person who's viewing the page is the same person who posted the joke, or if they have the relevant permission.

Editing the `jokes.html.php` template makes the buttons appear, but if you have the `EDIT_JOKE` permission and attempt to edit a joke you didn't post, you'll see the error "You may only edit jokes that you posted". That's because we added a specific check in the `editjoke.html.php` template, and the `delete` method in the `Joke` controller.

Change `delete` to include the permissions check:

```
public function delete() {

  $author = $this->authentication->getUser();

  $joke = $this->jokesTable->find('id', $_POST['id']);

  if ($joke->authorId != $author->id && !$author->
  ↪hasPermission(\Ijdb\Entity\Author::DELETE_JOKE)) {
    return;
  }

  $this->jokesTable->delete('id', $_POST['id']);

  header('location: /joke/list');
}
```

And, as with the `list` method, pass the entire `author` object to the template in the `edit` method, and adjust the template to include the permissions check.

`controllers/joke.php`:

```
return ['template' => 'editjoke.html.php',
  'title' => $title,
  'variables' => [
    'joke' => $joke ?? null,
    'user' => $author,
    'categories' => $categories
  ]
];
```

Now find this code in `editjoke.html.php`:

```
<?php if (empty($joke->id) || $userId == $joke->authorId):  ?>
```

Change it to this:

```
<?php if (empty($joke->id) || $user->id == $joke->authorId || $user->
↪hasPermission(\Ijdb\Entity\Author::EDIT_JOKE)): ?>
```

That's it! All the permission checks are now in place. The code is available in **Relationships-Permissions-Complete**[25]

Phew!

In this chapter, I showed you how to think in a more object-oriented way, and how to handle relationships between objects in an OOP way rather than in a relational way.

You learned how to represent many-to-many relationships using both `join` tables and bitwise operations.

We added permissions to the existing site, but moving forward, you can think about user permissions as you go and create them while you're writing the original code.

That's it! We have a completely functional and working website that does

25. https://github.com/spbooks/phpmysql7/tree/Relationships-Permissions-Complete

almost anything we would want for a real-world project. I have a couple of small things left to show you, but we're basically done! There's little else I can teach you.

In the next chapter, I'll show you how to make a few tweaks to the `DatabaseTable` class to allow sorting and limiting, but you already have almost all the tools you need to build a fully functional website.

Chapter 13

Content Formatting and Pagination

We're almost there! We've designed a database to store jokes, organized them into categories, and tracked their authors. We've learned how to create a web page that displays this library of jokes to site visitors. We've even developed a set of web pages that a site administrator can use to manage the joke library without knowing anything about databases.

In so doing, we've built a site that frees the programmer maintaining the site from continually having to plug new content into tired HTML page templates, and from maintaining an unmanageable mass of HTML files. The HTML is now kept completely separate from the data it displays. If you want to redesign the site, you simply have to make the changes to the HTML contained in the PHP templates that you've constructed. A change to one file (for example, modifying the footer) is immediately reflected in the page layouts of all pages in the site. Only one task still requires knowledge of HTML: **content formatting**.

On any but the simplest of websites, it will be necessary to allow content (in our case, jokes) to include some sort of formatting. In a simple case, this might merely be the ability to break text into paragraphs. Often, however, content providers will expect the facility to create **bold** or *italic* text, hyperlinks, and so on.

As it stands, we've stripped out any formatting from text entered by users using the `htmlspecialchars` function.

If, instead, we just `echo` out the raw content pulled from the database, we can enable administrators to include formatting in the form of HTML code in the joke text:

```
<?php echo $joke->joketext; ?>
```

Following this simple change, a site administrator could include HTML tags that would have their usual effect on the joke text when inserted into a page.

But is this really what we want? Left unchecked, content providers can do a lot

of damage by including HTML code in the content they add to your site's database. Particularly if your system will be enabling nontechnical users to submit content, you'll find that invalid, obsolete, and otherwise inappropriate code will gradually infest the pristine website you set out to build. With one stray tag, a well-meaning user could tear apart the layout of your site.

In this chapter, you'll learn about several PHP functions that you haven't seen before, which are used for finding and replacing patterns of text in your site's content. I'll show you how to use these capabilities to provide a simpler markup language for your users that's better suited to content formatting. By the time we've finished, we'll have completed a content management system that anyone with a web browser can use—without any knowledge of HTML being required.

Regular Expressions

To implement our own markup language, we'll have to write some PHP code to spot our custom tags in the text of jokes and then replace them with their HTML equivalents. For tackling this sort of task, PHP includes extensive support for regular expressions.

A **regular expression** is a short piece of code that describes a pattern of text that may occur in content like our jokes. We use regular expressions to search for and replace patterns of text. They're available in many programming languages and environments, and are especially prevalent in web development languages like PHP.

The popularity of regular expressions has everything to do with how useful they are, and absolutely nothing to do with how easy they are to use—because they're not at all easy. In fact, to most people who encounter them for the first time, regular expressions look like what might eventuate if you fell asleep with your face on the keyboard.

Here, for example, is a relatively simple (yes, really!) regular expression that will match any string that might be a valid email address:

```
/^[\w\.\-]+@([\w\-]+\.)+[a-z]+$/i
```

Scary, huh? By the end of this section, you'll actually be able to make sense of that.

The language of a regular expression is cryptic enough that, once you master it, you may feel as if you're able to weave magical incantations with the code you write. To begin with, let's start with some very simple regular expressions.

This is a regular expression that searches for the text "PHP" (without the quotes):

```
/PHP/
```

Fairly simple, right? It's the text you want to search for, surrounded by a pair of matching **delimiters**. Traditionally, slashes (`/`) are used as regular expression delimiters, but another common choice is the hash character (`#`). You can actually use any character as a delimiter except letters, numbers, or backslashes (`\`). I'll use slashes for all the regular expressions in this chapter.

Escaping Delimiters

To include a forward slash as part of a regular expression that uses forward slashes as delimiters, you must escape it with a preceding backslash (`\/`). Otherwise, it will be interpreted as marking the end of the pattern.

The same goes for other delimiter characters: if you use hash characters as delimiters, you'll need to escape any hashes within the expression with backslashes (`\#`).

To use a regular expression, you must be familiar with the regular expression functions available in PHP. `preg_match` is the most basic, and can be used to determine whether or not a regular expression is matched by a particular text string.

Consider this code:

```
<?php
$text = 'PHP rules!';

if (preg_match('/PHP/', $text)) {
    echo '$text contains the string "PHP".';
} else {
    echo '$text does not contain the string "PHP".';
}
```

In this example, the regular expression finds a match, because the string stored in the variable `$text` contains “PHP”. This example will therefore output the message shown in the image below.

$text contains the string “PHP”.

13-1. The regular expression finds a match

Use of Single Quotes

Notice that the single quotes around the strings in the code above prevent PHP from filling in the value of the variable `$text`.

By default, regular expressions are case-sensitive. That is, lowercase characters in the expression only match lowercase characters in the string, and uppercase characters only match uppercase characters. If you want to perform a case-insensitive search instead, you can use a pattern modifier to make the regular expression ignore case.

Pattern modifiers are single-character flags following the ending delimiter of an expression. The modifier for performing a case-insensitive match is `i`. So while `/PHP/` will only match strings that contain “PHP”, `/PHP/i` will match strings that contain “PHP”, “php”, or even “pHp”.

Here’s an example to illustrate this:

```
<?php
$text = 'What is Php?';

if (preg_match('/PHP/i', $text)) {
    echo '$text contains the string "PHP".';
} else {
    echo '$text does not contain the string "PHP".';
}
```

Again, as shown in the image below, this outputs the same message, despite the string actually containing “Php”.

$text contains the string “PHP”.

13-2. No need to be picky …

Regular expressions are almost a programming language unto themselves. A dazzling variety of characters have a special significance when they appear in a regular expression. Using these special characters, you can describe in great detail the pattern of characters that a PHP function like `preg_match` will search for. To show you what I mean, let’s look at a slightly more complex regular expression:

```
/^PH.*/
```

The caret (`^`) is placed at the beginning of an expression and indicates that the pattern must match the start of the string. The expression above will only match strings that start with `PH` .

The dot (`.`) means “any single character”. The expression `/PH./` would match `PHP` , `PHA` , `PHx` and any other string that started with `PH` and one more letter.

The asterisk (`*`) is a modifier for the dot, and it means “zero or more of the preceding character”. The expression `P*` would match `PPPPPPP` but not `PHP` .

`.*` matches *any character* zero or more times.

Therefore, the pattern `/^PH.*/` matches not only the string “PH”, but “PHP”, “PHX”, “PHP: Hypertext Preprocessor”, and any other string beginning with “PH”.

When you first encounter it, regular expression syntax can be downright confusing and difficult to remember, so if you intend to make extensive use of it, a good reference might come in handy. Regular expressions are a complex and extensive mini-language. I’m not going to try to cover it here. Instead, I’ll introduce the individual characters as we need them. The PHP manual includes a very thorough regular expression reference[1], and interactive tools such as regex101.com[2] are incredibly useful for visual learning.

String Replacement with Regular Expressions

As you may recall, in this chapter we’re aiming to make it easier for non-HTML-savvy users to add formatting to the jokes on our website. For example, if a user puts asterisks around a word in the text of a joke—such as `'Knock *knock*…'` —we’d like to display the joke with HTML emphasis tags around that word: `Knock <em>knock</em>…'` .

We can detect the presence of plain-text formatting such as this in a joke’s text using `preg_match` with the regular expression syntax we’ve just learned. However, what we *need* to do is pinpoint that formatting and *replace* it with appropriate HTML tags. To achieve this, we need to look at another regular expression function offered by PHP: `preg_replace` .

`preg_replace` , like `preg_match` , accepts a regular expression and a string of text, and attempts to match the regular expression in the string. In addition, `preg_replace` takes another string of text and replaces every match of the regular expression with that string.

---

1. http://php.net/manual/en/reference.pcre.pattern.syntax.php
2. https://regex101.com/

The syntax for `preg_replace` is as follows:

```
$newString = preg_replace($regExp, $replaceWith, $oldString);
```

Here, `$regExp` is the regular expression, and `replaceWith` is the string that will replace matches in `$oldString`. The function returns the new string with all the replacements made. In that code, this newly generated string is stored in `$newString`.

We're now ready to build our joke formatting function.

Emphasized Text

We could use a relevant `preg_replace` method everywhere it's required in our templates. However, since this is going to be useful in multiple places, and any website we build, we'll create a class for it and place it in our `Ninja` namespace:

```
namespace Ninja;

class Markdown {
  public function __construct(private string $string) {
  }

  public function toHtml() {
    // convert $this->string to HTML

    return $html;
  }
}
```

The plain-text formatting syntax we'll support is a simplified form of Markdown, a markup language created by John Gruber. Markdown is described like this on its home page[3]:

3. http://daringfireball.net/projects/markdown/

> *Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you to write using an easy-to-read, easy-to-write plain-text format, then convert it to structurally valid XHTML (or HTML).*

Since this class will convert Markdown to HTML, it's named `Markdown`.

This first action is to use the `htmlentities` function to convert any HTML code present in the text into text, by removing any characters that are understood by browsers (`<`, `>`, `&`, `"`). We want to avoid any HTML code appearing in the output, except that which is generated from plain-text formatting.

Not Quite Markdown

Technically, what we're doing here breaks one of the features of Markdown: support for inline HTML. "Real" Markdown can contain HTML code, which will be passed through to the browser untouched. The idea is that you can use HTML to produce any formatting that's too complex to create using Markdown's plain-text formatting syntax. Since we don't want to allow this, it might be more accurate to say we'll support Markdown-*style* formatting.

Let's start with formatting that will create **bold** and *italic* text.

In Markdown, you can emphasize text by surrounding it with a pair of asterisks (`*`), or a pair of underscores (`_`). (For example, *emphasized text* is achieved with `*emphasized text*` or `_emphasized text_`.) We'll replace any such pair with `<em>` and `</em>` tags.

`<em>` and `<strong>` vs `<i>` and `<b>`

You may be more accustomed to using `<i>` and `<b>` tags for bold and italic text. However, I've chosen to respect the most recent HTML standards, which recommend using the more meaningful `<em>` and `<strong>` tags respectively. If bold text doesn't necessarily indicate strong emphasis in your content, and italic text isn't representative of emphasis, you might want to use `<i>` and `<b>` instead.

To achieve this, we'll use two regular expressions: one that handles a pair of asterisks, and one that handles a pair of underscores.

Let's start with the underscores:

```
/_[^_]+_/
```

Breaking this down:

- `/` : we choose our usual slash character to begin (and therefore delimit) our regular expression.

- `_` : there's nothing special about underscores in regular expressions, so this will simply match an underscore character in the text.

- `[^_]` : square brackets are used to match a sequence of one or more characters that are placed between the opening bracket `[` and closing bracket `]`. The caret (`^`), when placed inside square brackets, acts as a logical *not*. The expression `[^_]` will match any character that is not an underscore.

- `+` : the plus character indicates one or more characters that match the preceding expression. `[^_]+` can be read as *one or more characters that are not an underscore*.

- `_` : the second underscore, which marks the end of the italicized text.
- `/` : the end of the regular expression.

In English, the expression `/_[^_]+_/` could be translated as: “Find an underscore, followed by one or more characters that aren’t an underscore, and stop at the following underscore.”

Now, it’s easy enough to feed this regular expression to `preg_replace`, but we have a problem:

```
$text = preg_replace('/_[^_]+_/', '<em>emphasized text</em>', $text);
```

The second argument we pass to `preg_replace` needs to be the text that we want to replace each match with. The problem is, we have no idea what the text that goes between the `<em>` and `</em>` tags will be. It’s part of the text that’s being matched by our regular expression!

Thankfully, another feature of `preg_replace` comes to our rescue. If you surround a portion of the regular expression with parentheses, you can capture the corresponding portion of the matched text and use it in the replacement string. To do this, you’ll use the code `$n`, where `n` is `1` for the first parenthesized portion of the regular expression, `2` for the second, and so on, up to `99` for the 99th. Consider this example:

```
$text = 'banana';
$text = preg_replace('/(.*)(nana)/', '$2$1', $text);
echo $text; // outputs 'nanaba'
```

So `$1` is replaced with the text matched by the first grouped portion of the regular expression (`(.*)` —zero or more non-newline characters), which is `ba` in this case. `$2` is replaced by `nana`, which is the text matched by the second grouped portion of the regular expression (`(nana)`). The replacement string `'$2$1'`, therefore, produces `'nanaba'`.

Double Quotes and Dollar Signs in Regular Expressions

If you think all the way back to Chapter 2, you'll remember that PHP supports both double quotes (`"`) and single quotes (`'`) for strings. One difference between double and single quotes is that variables in double quotes are automatically expanded. For example, take this code:

```
$name = 'Tom';

echo "Welcome back $name";
```

This will print "Welcome back Tom". This presents a problem when using regular expressions, as the dollar sign has a special meaning.

Let's swap out the single quotes for double quotes in the second argument from above:

```
$text = preg_replace('/(.*)(nana)/', "$2$1", $text);
```

Now, PHP will try to expand the variables `$1` and `$2` when the string is created, rather than using them as part of the regular expression replacement. To overcome this, you can escape the `$` in the same way as you would a quote inside the string, by prefixing it with a backslash:

```
$text = preg_replace('/(.*)(nana)/', "\$2\$1", $text);
```

We can use the same principle to create our emphasized text, adding parentheses to our regular expression:

```
/_([^_]+)_/
```

These parentheses have no effect on how the expression works at all, but they create a group of matched characters that we can reuse in our

replacement string. Converting this expression into English would read: “Find any underscore, then store anything that follows it which isn’t an underscore in the first match group”.

The following would replace `_Hello_` with `<em>Hello</em>`:

```
$text = preg_replace('/_([^_]+)_/', '<em>$1</em>', $text);
```

The pattern to match and replace pairs of asterisks looks much the same, except that we need to escape the asterisks with backslashes, since the asterisk character normally has a special meaning in regular expressions:

```
$text = preg_replace('/\*([^\*]+)\*/', '<em>$1</em>', $text);
```

That takes care of emphasized text, but Markdown also supports creating strong emphasis (`<strong>` tags) by surrounding text with a pair of *double* asterisks or underscores (`**strong emphasis**` or `__strong emphasis__`). Here’s the regular expression to match pairs of double underscores:

```
/__(.+?)__/s
```

The double underscores at the start and end are straightforward enough, but what’s going on inside the parentheses?

Previously, in our single-underscore pattern, we used `[^_]+` to match a series of one or more characters, none of which could be underscores. That works fine when the end of the emphasized text is marked by a single underscore. But when the end is a *double* underscore, we want to allow for the emphasized text to contain single underscores (for example, `__text_with_strong_emphasis__`). “No underscores allowed”, therefore, won’t cut it: we must come up with some other way to match the emphasized text.

You might be tempted to use `.+` (one or more characters, any kind), giving us a regular expression like this:

```
/__(.+)__/s
```

The s Pattern Modifier

The `s` pattern modifier at the end of the regular expression ensures that the dot (`.`) will truly match any character, including newlines.

The problem with this pattern is that the `+` is **greedy**: it will cause this portion of the regular expression to gobble up as many characters as it can. Consider this joke, for example:

```
__Knock-knock.__ Who’s there? __Boo.__ Boo who? __Aw, don’t cry about it!__
```

When presented with this text, the regular expression above will see just a single match, beginning with two underscores at the start of the joke and ending with two underscores at the end. The rest of the text in between (including all the other double underscores) will be gobbled up by the greedy `.+` as the text to be emphasized!

To fix this problem, we can ask the `+` to be *non-greedy* by adding a question mark after it. Instead of matching as many characters as possible, `.+?` will match as few characters as possible while still matching the rest of the pattern, ensuring we’ll match each piece of emphasized text (and the double-underscores that surround it) individually. This gets us to our final regular expression:

```
/__(.+?)__/s
```

Using the same technique, we can also come up with a regular expression for double-asterisks. This is how the finished code for applying strong emphasis ends up looking:

```
$text = preg_replace('/__(.+?)__/s', '<strong>$1</strong>', $text);
$text = preg_replace('/\*\*(.+?)\*\*/s', '<strong>$1</strong>', $text);
```

One last point: we must avoid converting pairs of single asterisks and underscores into `<em>` and `</em>` tags until after we've converted the pairs of double asterisks and underscores in the text into `<strong>` and `</strong>` tags. Our `toHtml` function, therefore, will apply strong emphasis first, then regular emphasis:

```
namespace Ninja;

class Markdown {
  public function __construct(private string $string) {
  }

  public function toHtml() {
    // remove any HTML characters and convert to UTF-8
    $text = htmlspecialchars($this->string, ENT_QUOTES, 'UTF-8');

    // strong (bold)
    $text = preg_replace('/__(.+?)__/s', '<strong>$1</strong>', $text);
    $text = preg_replace('/\*\*(.+?)\*\*/s', '<strong>$1</strong>', $text);

    // emphasis (italic)
    $text = preg_replace('/_([^_]+)_/', '<em>$1</em>', $text);
    $text = preg_replace('/\*([^\*]+)\*/', '<em>$1</em>', $text);

    return $text;
  }
}
```

Paragraphs

While we could choose characters to mark the start and end of paragraphs, just as we did for emphasized text, a simpler approach makes more sense. Since your users will type the content into a form field that allows them to create paragraphs using the `Enter` key, we'll take a single newline to indicate a line break (`<br />`) and a double newline to indicate a new paragraph (`</p><p>`).

You can represent a newline character in a regular expression as `\n`. Other whitespace characters you can write this way include a carriage return (`\r`) and a tab space (`\t`).

Exactly which characters are inserted into text when the user hits `Enter` depends on the user's operating system. In general, Windows represents a line break as a carriage return followed by a newline (`\r\n`). macOS used to represent it as a single carriage return character (`\r`). These days, macOS and Linux use a single newline character (`\n`) to indicate a new line.

Line Breaks and Software

In fact, the type of line breaks used can vary between software programs on the same computer. If you've ever opened a text file in Notepad to see all the line breaks missing, you've experienced the frustration this can cause. Advanced text editors used by programmers usually let you specify the type of line breaks to use when saving a text file.

To deal with these different line-break styles, any of which may be submitted by the browser, we must do some conversion:

```
// Convert Windows (\r\n) to Unix (\n)
$text = preg_replace('/\r\n/', "\n", $text);

// Convert Macintosh (\r) to Unix (\n)
$text = preg_replace('/\r/', "\n", $text);
```

Avoid Using Double-quoted Strings with Regular Expressions

All the regular expressions we've seen so far in this chapter have been expressed as single-quoted PHP strings. The automatic variable substitution provided by PHP strings is sometimes more convenient, but they can cause headaches when used with regular expressions.

Double-quoted PHP strings and regular expressions share a number of special character escape codes. `"\n"` is a PHP string containing a newline character. Likewise, `/\n/` is a regular expression that will match any string containing a newline character. We can represent this regular expression as a single-quoted PHP string (`'/\n/'`) and all is well, because the code `\n` has no special meaning in a single-quoted PHP string.

If we were to use a double-quoted string to represent this regular expression, we'd have to write `"/\\n/"`—with a double-backslash. The double-backslash tells PHP to include an actual backslash in the string, rather than combining it with the `n` that follows it to represent a newline character. This string will therefore generate the desired regular expression, `/\n/` .

Because of the added complexity it introduces, it's best to avoid using double-quoted strings when writing regular expressions. Note, however, that I *have* used double quotes for the replacement strings (`"\n"`) passed as the second parameter to `preg_replace` . In this case, we actually do want to create a string containing a newline character, so a double-quoted string does the job perfectly.

With our line breaks all converted to newline characters, we can convert them to paragraph breaks (when they occur in pairs) and line breaks (when they occur alone):

```
// Paragraphs
$text = '<p>' . preg_replace('/\n\n/', '</p><p>', $text) . '</p>';

// Line breaks
$text = preg_replace('/\n/', '<br />', $text);
```

Note the addition of `<p>` and `</p>` tags surrounding the joke text. Because our jokes may contain paragraph breaks, we must make sure the joke text is output within the context of a paragraph to begin with.

This code does the trick: the line breaks in the text will now become the natural line and paragraph breaks expected by the user, removing the requirement to learn anything new to create this simple formatting.

It turns out, however, that there's a simpler way to achieve the same result in this case: there's no need to use regular expressions at all! PHP's `str_replace` function works a lot like `preg_replace`, except that it only searches for strings instead of regular expression patterns:

```
$newString = str_replace($searchFor, $replaceWith, $oldString);
```

We can therefore rewrite our line-breaking code as follows:

```
// Convert Windows (\r\n) to Unix (\n)
$text = str_replace("\r\n", "\n", $text);
// Convert Macintosh (\r) to Unix (\n)
$text = str_replace("\r", "\n", $text);

// Paragraphs
$text = '<p>' . str_replace("\n\n", '</p><p>', $text) . '</p>';
// Line breaks
$text = str_replace("\n", '<br>', $text);
```

`str_replace` is much more efficient than `preg_replace`, because there's no need for it to apply the complex rules that govern regular expressions. Whenever `str_replace` (or `str_ireplace`, if you need a case-insensitive search) can do the job, you should use it instead of `preg_replace`.

One thing you may have spotted here is that, when using `str_replace`, I've put `\r` and `\n` in double quotes rather than single quotes. When PHP encounters `\n` or `\r`, it replaces them with the relevant character, but only when they're in double quotes. If single quotes were used, PHP would look for a backslash followed by the character `n` in the target string rather than the line-break character.

Hyperlinks

While supporting the inclusion of hyperlinks in the text of jokes may seem unnecessary, such a feature makes plenty of sense in other applications.

Here's what a hyperlink looks like in Markdown:

```
[linked text](link URL)
```

Linking in Markdown

Markdown also supports a more advanced link syntax where you put the link URL at the end of the document, as a footnote. But we won't be supporting that kind of link in our simplified Markdown implementation.

Simple, right? You put the text of the link in square brackets, and follow it with the URL for the link in parentheses.

As it turns out, you've already learned everything you need to match and replace links like this with HTML links. If you're feeling up to the challenge, you should stop reading right here and try to tackle the problem yourself!

First, we need a regular expression that will match links in this format. The regular expression is as follows:

```
/\[([^\]]+)]\((.+)\)/i
```

This is a rather complicated regular expression. You can see how regular expressions have gained a reputation for being indecipherable!

Squint at it for a little while and see if you can figure out how it works. Try writing out the expression on regex101.com[4] and it will display the regular expression in its groups with some useful highlighting. You can try typing in various strings to see which match.

Let me break it down for you:

- `/` : as with all our regular expressions, we choose to mark its beginning with a slash.
- `\[` : this matches the opening square bracket (`[`). Since square brackets have a special meaning in regular expressions, we must escape it with a backslash to have it interpreted literally.
- `([^\]]+)` : first of all, this portion of the regular expression is surrounded with parentheses, so the matching text will be available to us as `$1` when we write the replacement string. Inside the parentheses, we're after the linked text. Because the end of the linked text is marked with a closing square bracket (`]`), we can describe it as one or more characters, none of which is a closing square bracket (`[^\]]+`).
- `]\(` : this will match the closing square bracket that ends the linked text, followed by the opening parenthesis that signals the start of the link URL. The parenthesis needs to be escaped with a backslash to prevent it from having its usual grouping effect. (The square bracket doesn't need to be escaped with a backslash, because there's no unescaped opening square bracket currently in play.)
- `(.+)` : as URLs can contain (almost) any character, anything typed inside

4. https://regex101.com/

the Markdown parentheses will be matched by `.+` and stored inside the group `$2` in the replacement string.

- `\)` : this escaped parenthesis matches the closing parenthesis (`)`) at the end of the link URL.
- `/i` : we mark the end of the regular expression with a slash, followed by the case-insensitivity flag, `i` .

We can therefore convert links with the following PHP code:

```
$text = preg_replace(
  '/\[([^\]]+)]\(([-a-z0-9._~:\/?#@!$&\'()*+,;=%]+)\)/i',
  '<a href="$2">$1</a>', $text);
```

As you can see, `$1` is used in the replacement string to substitute the captured link text, and `$2` is used for the captured URL.

Additionally, because we're expressing our regular expression as a single-quoted PHP string, you have to escape the single quote that appears in the list of acceptable characters with a backslash.

Putting It All Together

Here's how our finished class for converting Markdown to HTML looks:

```
<?php
namespace Ninja;

class Markdown {
    public function __construct(private string $string) {
    }

    public function toHtml()
    {
        // convert $this->string to HTML
        $text = htmlspecialchars($this->string, ENT_QUOTES, 'UTF-8');
```

```
        // strong (bold)
        $text = preg_replace('/__(.+?)__/s', '<strong>$1</strong>', $text);
        $text = preg_replace('/\*\*(.+?)\*\*/s', '<strong>$1</strong>', $text);

        // emphasis (italic)
        $text = preg_replace('/_([^_]+)_/', '<em>$1</em>', $text);
        $text = preg_replace('/\*([^\*]+)\*/', '<em>$1</em>', $text);

        // Convert Windows (\r\n) to Unix (\n)
        $text = str_replace("\r\n", "\n", $text);
        // Convert Macintosh (\r) to Unix (\n)
        $text = str_replace("\r", "\n", $text);

        // Paragraphs
        $text = '<p>' . str_replace("\n\n", '</p><p>', $text) . '</p>';
        // Line breaks
        $text = str_replace("\n", '<br>', $text);

        // [linked text](link URL)
        $text = preg_replace('/\[([^\]]+)]\(([-a-z0-9._~:\/?#@!$&\'()*+,;=%]+)\)
        ➥/i', '<a href="$2">$1</a>',  $text);

        return $text;
    }
}
```

We can then use this class in our template that outputs the joke text, *`jokes.html.php`*:

```
<div class="jokelist">

<ul class="categories">
  <?php foreach ($categories as $category): ?>
    <li><a href="/joke/list/<?=$category->id?>"><?=$category->name?></a><li>
  <?php endforeach; ?>
</ul>

<div class="jokes">

<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>
```

```
<?php foreach ($jokes as $joke): ?>
<blockquote>
  <p>
  <?=htmlspecialchars($joke->joketext, ENT_QUOTES, 'UTF-8')?>

    (by <a href="mailto:<?=htmlspecialchars(
    $joke->getAuthor()->email,
    ENT_QUOTES,
      'UTF-8'
); ?>">
        <?=htmlspecialchars(
          $joke->getAuthor()->name,
          ENT_QUOTES,
          'UTF-8'
      ); ?></a> on
<?php
$date = new DateTime($joke->jokedate);

echo $date->format('jS F Y');
?>)

<?php if ($user): ?>
  <?php if ($user->id == $joke->authorId || $user->hasPermission
  ➥(\Ijdb\Entity\Author::EDIT_JOKE)): ?>
  <a href="/joke/edit/<?=$joke->id?>">Edit</a>
  <?php endif; ?>
  <?php if ($user->id == $joke->authorId || $user->hasPermission
  ➥(\Ijdb\Entity\Author::DELETE_JOKE)): ?>
  <form action="/joke/delete" method="post">
    <input type="hidden" name="id" value="<?=$joke->id?>">
    <input type="submit" value="Delete">
  </form>
  <?php endif; ?>
<?php endif; ?>
  </p>
</blockquote>
<?php endforeach; ?>

</div>
```

The line we're interested in is this:

```
<?=htmlspecialchars($joke->joketext, ENT_QUOTES, 'UTF-8')?>
```

However, each joke is already wrapped in a `<p>` tag. This can be removed:

```
<div class="jokelist">

<ul class="categories">
  <?php foreach($categories as $category): ?>
    <li><a href="/joke/list/<?=$category->id?>"><?=$category->name?></a><li>
  <?php endforeach; ?>
</ul>

<div class="jokes">

<p><?=$totalJokes?> jokes have been submitted to the Internet Joke Database.</p>

<?php foreach($jokes as $joke): ?>
<blockquote>
 <!-- Remove the opening tag <p> -->

  <?=htmlspecialchars($joke->joketext, ENT_QUOTES, 'UTF-8')?>

    <!--- … -->
<?php endif; ?>
  <!-- Remove the closing tag </p> -->
</blockquote>
<?php endforeach; ?>

</div>
```

Now, replace the line that shows the joke text with this:

```
<?php
$markdown = new \Ninja\Markdown($joke->joketext);
echo $markdown->toHtml();
?>
```

This will pass the contents of `joketext` to the `markdown` class as a constructor argument and call the `toHtml` method to convert the text to HTML.

This is a lot untidier than the original method, as it requires two lines. As with most things in PHP, there's a way to express this using shorter syntax:

```
<?=(new \Ninja\Markdown($joke->joketext))->toHtml()?>
```

The code above can be found in **Example: Formatting-Markdown**[5].

With these changes made, take your new plain-text formatting for a spin! Edit a few of your jokes to contain Markdown syntax and verify that the formatting is correctly displayed.

Why Using Markdown is Cool

What's nice about adopting a formatting syntax like Markdown for your own website is that there's often plenty of open-source code out there to help you deal with it.

Your newfound regular expression skills will serve you well in your career as a web developer, but if you want to support Markdown formatting on your site, the easiest way to do it would be *not* to write all the code to handle Markdown formatting yourself!

Commonly used Markdown libraries include ParseDown[6] and cebe/markdown[7].

Sorting, Limiting and Offsets

We've spent a lot of time writing PHP code and, thanks to the `DatabaseTable` class, it's been quite some time since you learned about any new SQL.

However, there's a few final MySQL features I'd like to show you before you get your Ninja title.

---

5. https://github.com/spbooks/phpmysql7/tree/Formatting-Markdown
6. https://github.com/erusev/parsedown
7. https://github.com/cebe/markdown

Sorting

MySQL supports asking for retrieved records in a specific order. At the moment, the joke list page displays jokes in the order they were posted. It would be better if it showed the newest first.

A `SELECT` query can contain an `ORDER BY` clause that specifies the column that the data is sorted by.

For our jokes table, the following query would order the jokes by the date they were posted. You can also specify a modifier of `ASC` (ascending, or counting up) or `DESC` (descending, or counting down):

```
SELECT * FROM `joke` ORDER BY `jokedate`
```

```
SELECT * FROM `joke` ORDER BY `jokedate` DESC
```

This query would select all the jokes and order them by date in *descending order*, with the newest first (or at the top).

Let's implement this on the website. All our SQL queries are generated by the `DatabaseTable` class, so we'll need to amend that to include an `ORDER BY` clause.

At the moment, the `findAll` method looks like this:

```
public function findAll($pdo, $table) {
  $stmt = $pdo->prepare('SELECT * FROM `' . $table . '`');
  $stmt->execute();

  return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
  ↪$this->className, $this->constructorArgs);
}
```

Let's add an optional argument for `ORDER BY`:

```
public function findAll($orderBy = null) {
  $query = 'SELECT * FROM `' . $this->table . '`';

  if ($orderBy != null) {
    $query .= ' ORDER BY ' . $orderBy;
  }

  $stmt = $this->pdo->prepare($query);
  $stmt->execute();

  return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
  ↪$this->className, $this->constructorArgs);
}
```

The `SELECT` query is now built up in the same way we built the `INSERT` and `UPDATE` queries. When a value for `$orderBy` is supplied, it's appended to the query along with the `ORDER BY` clause. By making the argument optional, all of our existing code will still work without modification. We can provide a value for the `$orderby` argument only where it's needed.

To sort the joke list page by date descending, amend the `Joke` controller's `list` method to supply the argument to the `findAll` method:

```
public function list($id = null) {

  if (isset($id)) {
    $category = $this->categoriesTable->find('id', $id)[0];
    $jokes = $category->getJokes();
  }
  else {
    $jokes = $this->jokesTable->findAll('jokedate DESC');
  }

  // …
```

At the moment, the main joke list page is sorted newest first. However, if you click on one of the categories, they're listed oldest first.

You might consider adding the same optional argument to the `find` method:

```
public function find(string $column, string $value, string $orderBy = null) {
    $query = 'SELECT * FROM `' . $this->table . '` WHERE `' . $column . '` =
    ↪ :value';

    $values = [
      'value' => $value
    ];

    if ($orderBy != null) {
      $query .= ' ORDER BY ' . $orderBy;
    }

    $stmt = $this->pdo->prepare($query);
    $stmt->execute($values);

    return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
    ↪$this->className, $this->constructorArgs);
}
```

Although this will be useful, it's not going to solve the problem. The list of jokes is generated in the `Category` entity class:

```
public function getJokes() {
  $jokeCategories = $this->jokeCategoriesTable->find('categoryId', $this->id);

  $jokes = [];

  foreach ($jokeCategories as $jokeCategory) {
    $joke =  $this->jokesTable->find('id', $jokeCategory->jokeId)[0] ?? null;
    if ($joke) {
      $jokes[] = $joke;
    }
  }

  return $jokes;
}
```

Because the `find` method is called on the `DatabaseTable` instance that represents the `joke_category` table, we can't easily sort by date.

There are a few ways to solve this. We could add a `date` column to the

`joke_category` table for sorting purposes. We could also use an SQL `JOIN`, but that would be difficult to implement into our OOP `DatabaseTable` class.

Instead, we can do the sort in PHP itself, using the `usort` function. The `usort` function takes two arguments: an array to be sorted, and the name of a function that compares two values.

The example given in the PHP manual[8] is this:

```
<?php
function cmp($a, $b) {
    if ($a == $b) {
        return 0;
    }
    return ($a < $b) ? -1 : 1;
}

$a = [3, 2, 5, 6, 1];

usort($a, "cmp");

foreach ($a as $key => $value) {
    echo "$key: $value\n";
}
```

The code above outputs this:

```
0: 1
1: 2
2: 3
3: 5
4: 6
```

The array has been sorted smallest to largest. The `cmp` function is called with two values from the array, and returns `1` if the first should be placed after the second, and `-1` if the first should be placed before the second. The important part is this line:

---

8. http://php.net/manual/en/function.usort.php

```
return ($a < $b) ? -1 : 1;
```

The syntax here looks strange if you haven't come across it before. You actually know what's happening here, but you haven't seen it expressed in this way. The code here is a shorthand (or ternary) `if` statement, and it's identical in execution to this:

```
if ($a < $b) {
  return -1;
} else {
  return 1;
}
```

The comparison function can take arguments that are objects, and we can build a comparison function into our `Category` class, as shown below.

Example: Formatting-Usort[9]

```
public function getJokes() {
  $jokeCategories = $this->jokeCategoriesTable->find('categoryId', $this->id);

  $jokes = [];

  foreach ($jokeCategories as $jokeCategory) {
    $joke =  $this->jokesTable->find('id', $jokeCategory->jokeId)[0] ?? null;
    if ($joke) {
      $jokes[] = $joke;
    }
  }

  usort($jokes, [$this, 'sortJokes']);

  return $jokes;
}

private function sortJokes($a, $b) {
  $aDate = new \DateTime($a->jokedate);
  $bDate = new \DateTime($b->jokedate);
```

9. https://github.com/spbooks/phpmysql7/tree/Formatting-Usort

```
  if ($aDate->getTimestamp() == $bDate->getTimestamp()) {
    return 0;
  }

  return $aDate->getTimestamp() > $bDate->getTimestamp() ? -1 : 1;
}
```

There's a lot going on here, so I'll go through it line by line. Firstly, the `$jokes` array is sorted using the `usort` function: `usort($jokes, [$this, 'sortJokes']);`. To call a method in a class, rather than just a function, you can use an array containing the object you want to call the method on (in our case, the same instance, `$this`) and the name of the method to be called (`sortJokes`).

The `sortJokes` method starts by converting the dates from each of the `$a` and `$b` objects into `\DateTime` instances for easier comparison. The `getTimestamp` method returns a Unix timestamp—the number of seconds between January 1, 1970, and the date being represented. Using timestamps allows us to compare the dates as integers.

The `if` statement checks to see if the dates have the same timestamp. If so, it returns `0`, indicating that neither should be moved before or after the other in the sorted list.

If the dates are different, either `1` or `-1` is returned to sort the dates. Notice I've used `$a > $b`, which will sort the array in the opposite order to the example, and put the larger timestamps (later dates) first.

There's a slight performance overhead in using `usort` instead of `ORDER BY` and having the database perform the sort, but unless you're dealing with thousands of records, the difference between the two will be milliseconds at worst!

Pagination with `LIMIT` and `OFFSET`

Now that you know how to sort the records, we can think a little about scalability. You've probably got fewer than a dozen jokes in your database at the moment. What will happen after the website has been online a few months and is starting to get popular? You might get users coming on and posting hundreds of jokes a day.

It won't take long before the joke list page takes a very long time to load, because it's displaying hundreds or thousands of jokes. The performance alone will put users off, but nobody is going to sit and read through a page of two thousand jokes.

A common approach is using **pagination** to display a sensible number—for example, ten jokes per page—and allow clicking a link to move between pages.

Before you continue, add at least 21 jokes to your database so we can test this correctly. Alternatively, for testing purposes, change `10` in the following sections to `2` to display two jokes per page.

What If I Don't Know any Jokes?

Don't worry if you can't think of any jokes. Just add some test data like "joke one", "joke two", "joke three", and so on.

Our first task is to display just the first ten jokes. Using SQL, this is incredibly easy. The `LIMIT` clause can be appended to any `SELECT` query to restrict the number of records returned:

```
SELECT * FROM `joke` ORDER BY `jokedate DESC` LIMIT 10
```

We'll need to build this into the `findAll` and `find` methods of the `DatabaseTable` class as optional parameters, as we did with the `$orderBy` variable:

```
public function find(string $column, string $value, string $orderBy = null,
  ➥int $limit = 0) {
    $query = 'SELECT * FROM `' . $this->table . '` WHERE `' . $column . '` =
    ➥:value';

    $values = [
      'value' => $value
    ];

    if ($orderBy != null) {
      $query .= ' ORDER BY ' . $orderBy;
    }

    if ($limit > 0) {
        $query .= ' LIMIT ' . $limit;
    }

    $stmt = $this->pdo->prepare($query);
    $stmt->execute($values);

    return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
    ➥$this->className, $this->constructorArgs);
}

public function findAll(string $orderBy = null, int $limit = 0) {
    $query = 'SELECT * FROM ' . $this->table;

    if ($orderBy != null) {
        $query .= ' ORDER BY ' . $orderBy;
    }

    if ($limit > 0) {
        $query .= ' LIMIT ' . $limit;
    }

    $stmt = $this->pdo->prepare($query);
    $stmt->execute();

    return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
    ➥$this->className, $this->constructorArgs);
}
```

Then, to limit to ten jokes, open up the `Joke` controller class and provide the

value `10` for the new `$limit` argument:

```
$jokes = $this->jokesTable->findAll('jokedate DESC', 10);
```

Also supply the new limit in the `Category` entity class:

```
$jokeCategories = $this->jokeCategoriesTable->find('categoryId',
↪$this->id, null, 10);
```

You'll notice I've supplied `null` for the `$orderBy` argument. Even though the argument is optional, to provide a value for `$limit`, a value for all the earlier arguments must be provided. However, as of PHP 8, named parameters can be used like so:

```
$jokeCategories = $this->jokeCategoriesTable->find(column: 'categoryId',
                                                    value: $this->id,
                                                    limit: 10);
```

With that in place, you'll see only ten jokes on the joke list page. The problem now is how we'll view the rest of the jokes!

The solution is to have different pages that can be accessed by a URL argument: `/joke/list/1/1` would view page `1` in the category with the ID `1`, or `/joke/list/1/2` to select which page 1 of category 2. We already have the first argument set up and working, so we just need to apply the second. However, what if we want to display page 2 of all jokes? To do that, we'll need to use an identifier that isn't a valid category ID, such as `all`, so that `/joke/list/all/3` is page 3 of all jokes.

Before that, let's create the links in the template. We can easily use a `for` loop to display a set of links to different pages. The following code will display links to pages 1–10 in the `all` category:

```
for ($i = 1; $i <= 10; $i++) {
  echo '<a href="/joke/list/all/' . $i . '">' . $i '</a>';
```

```
}
```

The problem is, we need to know how many pages there will be. It's actually very easy to work out. If we're displaying ten jokes per page, the number of pages is the number of jokes in the database divided by ten, and then rounded up.

With 21 jokes in the system, `21/10` is `2.1`, and if we round up, it gives `3` pages (two pages of ten and a final page with just one joke). PHP's `ceil` function can be used to round up any decimal number.

The template already has access to the `$totalJokes` variable, so we can display the pages at the end of `jokes.html.php`:

```
// …
<?php endif; ?>
</blockquote>
<?php endforeach; ?>

Select page:

<?php
// Calculate the number of pages
$numPages = ceil($totalJokes/10);

// Display a link for each page
for ($i = 1; $i <= $numPages; $i++):
?>
  <a href="/joke/list/all/<?=$i?>"><?=$i?></a>
<?php endfor; ?>

</div>
```

If you click the links, the second URL argument will be set. It's now just a matter of using it to display different sets of jokes.

The SQL clause `OFFSET` can be used with `LIMIT` to do exactly what we want:

```
SELECT * FROM `joke` ORDER BY `jokedate` LIMIT 10 OFFSET 10
```

This query will return ten jokes, but instead of returning the *first* ten jokes, it will display ten jokes starting from joke ten.

We'll need to turn page numbers into offsets. Page `1` will be `OFFSET 0`, page `2` will be `OFFSET 10`, page `3` will be `OFFSET 20`, and so on. This is a simple calculation: `$offset = ($page-1)*10`.

As we did with `limit`, let's add `OFFSET` as an optional argument for the `findAll` and `find` methods:

```
public function findAll(string $orderBy = null, int $limit = 0, int
↪$offset = 0) {
  $query = 'SELECT * FROM ' . $this->table;

  if ($orderBy != null) {
    $query .= ' ORDER BY ' . $orderBy;
  }

  if ($limit > 0) {
    $query .= ' LIMIT ' . $limit;
  }

  if ($offset > 0) {
    $query .= ' OFFSET ' . $offset;
  }

  $stmt = $this->pdo->prepare($query);
  $stmt->execute();

  return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
  ↪$this->className, $this->constructorArgs);
}

public function find(string $column, string $value, string $orderBy = null,
↪int $limit = 0, int $offset = 0) {
  $query = 'SELECT * FROM ' . $this->table . ' WHERE ' . $column . ' = :value';

  $parameters = [
```

```
    'value' => $value
  ];

  if ($orderBy != null) {
    $query .= ' ORDER BY ' . $orderBy;
  }

  if ($limit > 0) {
    $query .= ' LIMIT ' . $limit;
  }

  if ($offset > 0) {
    $query .= ' OFFSET ' . $offset;
  }

  $stmt = $this->pdo->prepare($query);
  $stmt->execute();

  return $stmt->fetchAll(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
  ↪$this->className, $this->constructorArgs);
}
```

Then supply the offset in the `list` method in the `Joke` controller:

```
public function list(mixed $categoryId = 0, int $page = 0) {

  if (is_numeric($categoryId)) {
    $category = $this->categoriesTable->find('id', $categoryId)[0];
    $jokes = $category->getJokes();
  }
  else {
    $jokes = $this->jokesTable->findAll('jokedate DESC', 10, $page);
  }

  $title = 'Joke List';

  $totalJokes = $this->jokesTable->total();

  $author = $this->authentication->getUser();

  return ['template' => 'jokes.html.php',
  'title' => $title,
```

```
    'variables' => [
      'totalJokes' => $totalJokes,
      'jokes' => $jokes,
      'user' => $author,
      'categories' => $this->categoriesTable->findAll()
    ]
  ];
}
```

Now, if you click between the different page links, you'll see ten different jokes on each page.

This new pagination doesn't work for the lists within categories, and we'll fix that shortly. But at the moment, the links aren't very user friendly. Let's change the styling of the link that represents the current page.

We can pass the number of the current page and current category to the template:

```
return ['template' => 'jokes.html.php',
  'title' => $title,
  'variables' => [
    'totalJokes' => $totalJokes,
    'jokes' => $jokes,
    'user' => $author,
    'categories' => $this->categoriesTable->findAll(),
    'currentPage' => $page,
    'categoryId' => $categoryId
  ]
];
```

Then add a CSS class to the page link if it's the current page:

```
Select page:

<?php

$numPages = ceil($totalJokes/10);
```

```
for ($i = 1; $i <= $numPages; $i++):
  if ($i == $currentPage):
?>
  <a class="currentpage" href="/joke/list?page=<?=$i?>"><?=$i?></a>
<?php else: ?>
  <a href="/joke/list/<?=$categoryId?>/<?=$i?>"><?=$i?></a>
<?php endif; ?>
<?php endfor; ?>

</div>
```

I've added a `currentpage` CSS class to the link if the link being printed is the current page being viewed. Add some CSS to `jokes.css` to make the link stand out. You could change the color, make it bold, underlined, and so on. I've chosen to surround the number with square brackets.

```
.currentpage:before {
  content: "[";
}
.currentpage:after {
  content: "]";
}
```

Pagination in Categories

We have a small bug in the code at the moment. If you click on one of the categories, it won't supply the correct `offset` value.

To fix this, we can add an `$offset` argument to the `Category` entity's `getJokes` method. While you're there, you may as well supply `$limit` as an argument as well to improve flexibility, instead of hardcoding it in the method:

```
public function getJokes(int $limit = 0; int $offset = 0) {
  $jokeCategories = $this->jokeCategoriesTable->find('categoryId', $this->id,
  ↪null, $limit, $offset);

  $jokes = [];
```

```
  foreach ($jokeCategories as $jokeCategory) {
    $joke =  $this->jokesTable->find('id', $jokeCategory->jokeId)[0] ?? null;
    if ($joke) {
      $jokes[] = $joke;
    }
  }

  usort($jokes, [$this, 'sortJokes']);

  return $jokes;
}
```

Then provide the values when the method is called in the `list` method:

```
if (isset($category)) {
  $category = $this->categoriesTable->find($category)[0];
  $jokes = $category->getJokes(10, $offset);
}
```

With that done, the pagination will work … kind of. You can manually enter the page number in the URL—for example, `https://v.je/joke/list/1/1`. However, the links we created don't work.

If you run the code, you'll notice a problem if you've selected a particular category. The number of page links displayed is based on the total number of jokes in the database, not the number of jokes in the selected category.

At the moment, the `total` method in the `DatabaseTable` class returns the total number of records in a given table. To count a subset of the records, it will need a `WHERE` clause. We can implement it in the same way as the `find` method:

```
public function total(string $field = null, string $value = null) {
  $sql = 'SELECT COUNT(*) FROM `' . $this->table . '`';
  $parameters = [];

  if (!empty($field)) {
```

```
    $sql .= ' WHERE `' . $field . '` = :value';
    $parameters = ['value' => $value];
  }

  $query = $this->query($sql, $parameters);

  $row = $query->fetch();
  return $row[0];
}
```

The `total` method now supports doing something like `echo $this->jokesTable->total('authorId', 4);`, which would give us the total number of jokes by the author with the `id` of `4`.

We can't do the same to count the number of jokes in a category, as there's no `categoryId` column in the `joke` table. We need to call the `total` method on the `jokeCategoriesTable` instance—`$this->jokeCategoriesTable->total('categoryId', 2);`—which would count the number of jokes in the category with the `id` of `2`.

Instead of implementing this in the `list` method, let's add a new method to the `Category` entity class that returns the number of jokes in that particular category—`$totalJokes = $category->getNumJokes();`:

```
public function getNumJokes() {
  return $this->jokeCategoriesTable->total('categoryId', $this->id);
}
```

This can then be called from the `list` method in the `Joke` controller.

Example: Final-Website[10]

```
public function list(mixed $categoryId = 0, int $page = 0) {

  $offset = ($page-1)*10;
```

10. https://github.com/spbooks/phpmysql7/tree/Final-Website

```
if (is_numeric($categoryId)) {
  $category = $this->categoriesTable->find('id', $_GET['category'])[0] ?? null;
  $jokes = $category->getJokes(10, $offset);
  $totalJokes = $category->getNumJokes();
}
else {
  $jokes = $this->jokesTable->findAll('jokedate DESC', 10, $offset);
  $totalJokes = $this->jokesTable->total();
}

$title = 'Joke List';
// …
```

Notice that I've moved the original `$totalJokes` variable into the `else` branch of the `if` statement. When a category is selected, `$totalJokes` is the total number of jokes in the selected category. When no category has been chosen, `$totalJokes` stores the total number of jokes in the database.

Achievement Unlocked: Ninja

That's it! You're done, and you get your PHP black belt.

In this chapter, I showed you some additional tools that will be useful when you develop your next website. You have a basic understanding of regular expressions, along with the SQL features of `LIMIT` and `OFFSET`, and you know how to combine them to paginate data sets.

You now have all the tools you need to build a real website. You know how to think about writing code, and you know how to separate out project-specific code from the code you can use in future projects. You also have an understanding of the concepts behind PHP frameworks. You can jump into Symfony, Zend or Laravel, and although the code will be different, all the concepts you've learned in this book will be familiar.

What Next?

You have all the tools you need to build a fully functional PHP website and put

it live on the Web. Go ahead and publish your first website. It's a great feeling!

With programming, there's always more to learn. There are different techniques and approaches you can try out, and a lot of different tools that will help you develop more efficiently and reduce bugs.

Remember that you're never "done". You never complete the game. You just keep playing. Each time you learn something new—be it a new tool, a new technique, or even a new language—it will extend what you knew before and you'll wonder how you ever coped without it. Things also change constantly, and it's difficult to keep up. Why do you think we're on the seventh edition of this book? Don't be disappointed: learning is fun, and as long as you don't fall into the trap of thinking you know everything, you'll go a long way.

Now that you've finished this book, you do, however, have more than enough knowledge to work on your own projects, or even to get a job as a junior PHP developer!

Before taking the next few steps, I recommend getting at least two or three projects finished to ensure you're comfortable with everything from this book. It won't sink in right away, and as you go forward, you'll find yourself solving different sets of problems. It will take you a few attempts to get everything clear in your mind.

Once you've done that, you can move on to the next few steps. What are those steps?

- Composer[11]. Composer is a package management tool that's used by almost all PHP projects these days. If you want to use someone else's code in your project, you'll need to know how to use Composer.

- Take a look at some PHP frameworks to see how other people do things. In 2022, I'd recommend Laravel[12] and Symfony[13] as starting points. They've

---

11. https://www.sitepoint.com/re-introducing-composer/
12. https://www.sitepoint.com/bootstrapping-laravel-crud-project/

been going strong for years now, and don't look like being replaced for a good while yet.

- PHPUnit[14]. Test-driven development has really taken off in PHP over the last few years, and for good reason. Once you start using TDD, it's difficult to go back. Everything seems so much tidier and easier. Rather than having to load up your website, fill in your form, then check that the record was inserted into a database, you can just run a script that does all that for you!

- Git[15]. Git is a vital tool for software developers. You will likely have come across the website GitHub, which allows sharing code and collaborating with other developers; in fact, the code archive for this book is hosted there. To use the site, you'll need to understand Git. But at its most basic, it's an incredible tool. No more copy/pasting code after making a change, or commenting out large sections. Just delete it, and Git will keep track of any changes you make!

With that said, there's little else I need to add. However you proceed from this point, rest assured you're starting out with a solid grounding in the essentials and a good understanding of the tools and techniques used by modern PHP websites. That's more than can be said for many developers working today. Take that advantage and use it.

Most importantly, go out there and write some code!

---

13. https://www.sitepoint.com/building-a-web-app-with-symfony-2-bootstrapping/
14. https://www.sitepoint.com/re-introducing-phpunit-getting-started-tdd-php/
15. https://www.sitepoint.com/git-for-beginners/