

# HACKING WITH SWIFT



# SERVER-SIDE SWIFT

**COMPLETE TUTORIAL COURSE**

Learn to make web  
apps with real-world  
Swift projects

Paul Hudson

# Contents

<b>Preface</b>	<b>6</b>
About this book	
Why does Kitura exist?	
A note for existing Swift developers	
Configuring your machine	
<b>Introduction: Swift for Complete Beginners</b>	<b>21</b>
How to install Xcode and create a playground	
Variables and constants	
Types of Data	
Operators	
String interpolation	
Arrays	
Dictionaries	
Conditional statements	
Loops	
Switch case	
Functions	
Optionals	
Optional chaining	
Enumerations	
Structs	
Classes	
Properties	
Static properties and methods	
Access control	
Polymorphism and typecasting	
Closures	
Wrap up	
<b>Million Hairs</b>	<b>101</b>
Setting up	
Swift packages explained	
Starting a server	
Routing user requests	
Writing HTML	
Matching custom routes	
Making it look good	

Wrap up

<b>CouchDB Poll</b>	<b>139</b>
Setting up	
CouchDB from scratch	
Designing a JSON API	
Bringing CouchDB and Kitura together	
Creating CouchDB documents	
Casting a vote	
Wrap up	
<b>Routing</b>	<b>178</b>
Setting up	
Chaining routes	
Reading user data	
Routing regular expressions	
Wrap up	
<b>Swift Fan Club</b>	<b>191</b>
Setting up	
Listing forums	
Querying with a key	
Closures within closures	
Users and sessions	
Posting messages and replies	
Wrap up	
<b>Meme Machine</b>	<b>237</b>
Setting up	
Reading files from disk	
Multi-part form encoding	
Wrap up	
<b>Templates</b>	<b>251</b>
Setting up	
Recap on the basics	
Filters and tags	
Wrap up	
<b>Barkr</b>	<b>265</b>
Setting up	

An SQL primer  
Integrating MySQL with Kitura  
Generating tokens  
Reading insert IDs  
Over to you: fuzzy search  
Wrap up

## ASCII art 300

Setting up  
Updating pages interactively  
Reading remote images  
Wrap up

## Databases 316

Setting up  
Indexing for performance  
Normalization for efficiency  
Referential integrity for organization  
Transactions for consistency  
Default values for safety  
Wrap up

## Instant Coder 333

Setting up  
Say hello with GitHub  
Rendering the basic UI  
Creating new accounts  
Routing to success  
Creating new projects  
Finding places to work  
Wrap up

## AppleFanatic 370

Setting up  
Bootstrapping our servers  
Fetching stories from a database  
IDs, slugs, and Markdown  
Stories and errors  
Browsing by category  
Creating an admin section  
Wrap up

## Testing

422

- Setting up
- Bootstrapping XCTest
- Testing our routes
- Wrap up

# Preface

# About this book

The **Server-Side Swift** tutorial series is designed to make it easy for beginners to get started building web apps and websites using the Swift programming language.

My teaching method skips out a lot of theory. It skips out the smart techniques that transform 20 lines of easy-to-understand code into 1 line of near-magic. It ignores coding conventions by the dozen. And perhaps later on, once you've finished, you'll want to go back and learn all the theory I so blithely walked past. But let me tell you this: the problem with learning theory by itself is that your brain doesn't really have any interest in remembering stuff just for the sake of it.

You see, here you'll be learning to code on a Need To Know basis. Nearly everything you learn from me will have a direct, practical application to something we're working on. That way, your brain can see exactly why a certain technique is helpful and you can start using it straight away.

This book has been written on the back of my personal motto: "Programming is an art. Don't spend all your time sharpening your pencil when you should be drawing." We'll be doing some "sharpening" but a heck of a lot more "drawing" – if that doesn't suit your way of learning, you should exit now.

## The three golden rules

The series is crafted around a few basic tenets, and it's important you understand them before continuing:

1. Follow the series: The tutorials are designed to be used in order, starting at the beginning and working through to the end. The reason for this is that concepts are introduced sequentially on a need-to-know basis – you only learn about something when you really have to in order to make the project work.
2. Don't skip the technique projects: The tutorials follow a sequence that places a technique project after every two app projects. That is, you develop two apps then we focus on a particular component to help make your code better. The apps are standalone projects that you can go on to develop as you wish, whereas the technique tutorials will often be used to improve or prepare you for other projects.

3. Get ready to hack: This is not designed to be the one-stop learning solution for all your Swift needs. The goal of each project is to reach the end with as little complication as possible, while learning one or more things along the way.

I can't re-iterate that last point enough. What I have found time and time again is that any tutorial, no matter how carefully written or what audience it's aimed at, will fail to fit the needs of many possible readers. And these people get angry, saying how the tutorial is wrong, how the tutorial is lame, how their tutorial would be much better if only they had the time to write it, and so on.

Over the last 12 years of writing, I have learned to ignore minority whining and move on, because what matters is that this tutorial is useful to *you*.

You'd be surprised by how many people think the path to success is through reading books, attending classes or, well, doing pretty much anything except sitting down in front of a computer and typing. Not me. I believe the best way to learn something is to try to do it yourself and see how it goes.

Sure, going to classes might re-enforce what you've learned, or it might teach you some time-saving techniques, but ultimately I've met too many people with computing degrees who stumble when asked to write simple programs. Don't believe me? Try doing a Google search for "fizz buzz test", and you'll be surprised too.

So, dive in, make things, and please, please, please have fun – because if you're not enjoying yourself, Swift coding probably isn't for you.

If you spot any errors in this book, either typos or technical mistakes, please do let me know so I can correct them as soon as possible. The best way to get in touch is on Twitter [@twostraws](#), but you can also email [paul@hackingwithswift.com](mailto:paul@hackingwithswift.com).

## Xcode, Swift and Kitura

I'm not going to talk much, because I want to get straight into coding. However, there are some points you do need to know:

- I've written these tutorials using Xcode 8.1, which is available for free from the Mac App Store. If you're using an earlier version of Xcode you should upgrade before continuing otherwise you may encounter bugs.
- If you're using Linux or *considering* using Linux, please read the introductory chapter "A note on Linux".
- Swift is a relatively new language, and is evolving quickly. Every new release of Xcode seems to change something or other, and often that means code that used to work now no longer does. At the time of writing, Swift is mature enough that the changes are relatively minor, so hopefully you can make them yourself. If not, check to see if there's an update of the project files on [hackingwithswift.com](http://hackingwithswift.com).
- These projects are designed to work with Kitura 1.0 or later. Please upgrade your system otherwise you'll find this book very confusing indeed!

**Important note:** if any bugs are found in the project files, or if Swift updates come out that force syntax changes, I'm going to be updating this book as needed. You should [follow me on Twitter @twostraws](#) if you want to be notified of updates.

I'm also happy to answer questions on Twitter if you encounter problems, so please feel free to get in touch!

Swift, the Swift logo, Xcode, Instruments, Cocoa Touch, Touch ID, AirDrop, iBeacon, iMessage, iPhone, iPad, Safari, App Store, Mac, and macOS are trademarks of Apple Inc., registered in the U.S. and other countries.

Server-Side Swift is copyright Paul Hudson. All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

## Acknowledgements

I'd like to thank Ankit Aggarwal for his invaluable help and advice with the Swift package manager. It's a fast-developing part of our ecosystem, but Ankit is doing a great job documenting all the changes and supporting everyone who has questions.

I'm also grateful to Kyle Fuller for being so responsive with answers, features, and fixes to his

Stencil template engine. His work made my life a great deal easier, and I look forward to seeing where Stencil goes next!

## Dedication

This book is dedicated to IBM's Kitura team, who have patiently answered my many questions on their Slack channel. Chris Bailey, Shmuel Kallner, Youming Lin, Ian Partridge, and others – thank you!

## Why does Kitura exist?

It doesn't matter whether you've been using iOS, macOS, watchOS, or tvOS – they are all very different to working on the server. Taking iOS as an example, you're already used to the idea that apps are built using multiple layers. Here are some of them:

1. The Swift programming language provides us with basic syntax and functionality like `func`, `var`, as well as closures, classes, and so on.
2. The Swift standard library is a large body of code that provides basic data types and methods: what is an `Int`? How do you sort an `Int` array? What does the `print()` function do?
3. Foundation is Apple's framework for providing extended data types, such as `Date`, `Data`, and `UUID`.
4. Grand Central Dispatch (GCD or “libdispatch”) is a low-level framework for letting us add concurrency without much of the pain. (That is, the ability to run more than one piece of code at a time.)
5. Core Graphics, Core Image, Core Animation, Core Data, Core Text, and many other low-level frameworks provide specific services we can draw on to solve problems.
6. UIKit is Apple's framework for creating interactive user interfaces for iOS.

These layers build on top of each other: if you remove any of the lower layers, UIKit would either not exist or be dramatically worse off. Together, layers 3 through 6 are sometimes (erroneously) called “Cocoa Touch” on iOS or “Cocoa” on macOS, but it's a helpful term because it just means “our fundamental app building blocks.”

That list is far from exhaustive, but it gives us enough for me to describe the most important thing you need to know: server-side Swift is Swift without most of Apple's frameworks. Yes, you still get the Swift language and the standard library, but you get none of the “Core” layer (Core Graphics and so on), you get none of the UIKit, AppKit, or WatchKit frameworks, and Foundation is incomplete.

Yes, you read that right: Foundation is incomplete. This means many methods will fail because they haven't been written yet, other methods might work for some specific code paths, and other methods are implemented fully. On the bright side, you do get GCD, which works just the same on the server as it does on Apple's platforms.

So, with large parts of Foundation missing, plus all “Core” and “Kit” frameworks completely absent, server-side Swift is significantly simpler. If you followed my books Hacking with Swift or Hacking with macOS, you’ll know that most of the learning is about figuring out how UIKit works. Not so with server-side Swift: because it’s significantly simpler, you can spend more time focusing on how to build projects with what you know rather than always having to learn new things.

Because Apple’s framework footprint is significantly smaller, third-party libraries step in to fill the gaps. That’s where Kitura comes in: it’s a framework developed by IBM to provide the tools we need to build web apps and websites. There are four main web frameworks at the time of writing, but I chose Kitura for this book for a number of reasons:

1. It’s extremely fast, coming 1st or 2nd in most benchmarks.
2. It was introduced by Apple at WWDC 2016, as part of their partnership with IBM to bring Swift to the enterprise.
3. It’s being backed with considerable investment by IBM, so it’s picking up pace extremely quickly.
4. It’s similar to Express.js, a hugely popular JavaScript framework, which means many existing web developers can pick up Kitura easily.

Each of those reasons is compelling by itself, but there’s another important reason that really shapes the way you work with server-side Swift: Kitura is highly modular. This means you use the basics of Kitura to handle fundamental functionality, but after that you can pick and choose components from elsewhere to solve whatever problems you have.

In this book, I’ll be using Kitura for as much as possible, because it comes with a lot of the functionality we need to build great web apps. However, I’m not an API zealot: when alternatives such as Vapor do something better, we’ll be using that. Remember, Kitura is modular, so you can plug other components in as needed – this is by design, and means we can all use what works best rather than being constrained to one specific approach.

So, although we lose large parts of Foundation and all the higher frameworks, many gaps are filled by frameworks such as Kitura. And when Kitura alone isn’t enough, we’ll be drawing on other frameworks such as Vapor – as long as it’s Swift, it’s available for us to draw on.

## A note for existing Swift developers

If you've written Swift for one of Apple's four platforms, moving to server-side Swift can be a bit of a shock. As I've said, it's *simpler*, but "simple" and "easy" aren't the same thing.

Sure, server-side Swift doesn't have UIKit, AppKit, Core Graphics, and so on – massive APIs that take months to learn and years to master. But in their place, you'll find a huge range of things that web developers take for granted: sessions, templates, forms, routing, HTTP methods, and more. Databases alone are so big, and so utterly fundamental to any website of note, that there are 1000-page books dedicated to them.

You'll also find that server-side Swift develops significantly faster than iOS, macOS, or other Apple platforms. Not only does Apple create yearly releases, but they also have extremely extended deprecation periods – you can rely on deprecated APIs to carry on working for three to five years before they are finally removed entirely. Not so on on the server: often releases happen every *week*, and it's common to see breaking changes come and go rather than have a gentle deprecation period.

Because many of these things are likely to be new to you, I want to place a message of encouragement early on in this book: if you find yourself thinking that all this web stuff is hard, you're *right* – it *is* hard. But I've done my best to explain everything slowly, carefully, clearly, and thoroughly, so I hope you're able to stick with it. If (when!) you find that APIs have been changed between me writing this book and you reading it, email me at [paul@hackingwithswift.com](mailto:paul@hackingwithswift.com) and I'll do my best to help.

The complexity of server-side Swift is such that this book is a little different from my others. More specifically, although I still try to teach things in a linear fashion, the projects *aren't* strictly graded by difficulty. Even *more* specifically, I've inserted two easier projects part-way through the series to help break up the flow, because without them it can just seem like an overwhelming amount of hard work.

Trust me, by the time you reach project 5 (the first of the easier projects) you'll be more than ready for a break!

# Configuring your machine

Working with server-side Swift is quite different from client-side Swift such as iOS and macOS for one large reason: macOS is very rarely used on servers. It's *possible* to use macOS for your web server, but this usually only happens if someone desperately needs macOS features such as Xcode running on a remote server.

Instead, servers are primarily the domain of Linux, an open-source operating system that has some things – but not many! – in common with macOS.

When Swift was open sourced in December 2015, Apple shipped Swift and the standard library for macOS and Linux. They also shipped a limited version of Foundation, which has slowly been improving over subsequent months.

IBM's Kitura framework, along with the various Kitura-related components that IBM provides, are also designed to work on macOS and Linux, so as long as you stay within the boundaries of the Swift standard library, the open-source Foundation framework and Kitura, your code should work identically on both macOS and Linux.

Now, "should" is a complex word that really means "in theory this does what I think it does, but in practice I bet it doesn't." The problem is two-fold:

1. Foundation on macOS is the original Apple version, which is feature-complete and battle-tested. Foundation on Linux is Apple's open-source reimplementation, which is missing many features.
2. macOS uses a case-insensitive filesystem by default, whereas Linux uses a case-*sensitive* filesystem. This means Linux is less forgiving about small filename mistakes. If you code on macOS and deploy to Linux, you might hit filesystem problems due to case-sensitivity.

Now for the complicated part. If you're already an iOS or macOS developer, you're used to the concept that your entire environment is dictated by your Xcode version number. Which SDKs you have, what Swift compiler version you have, how your storyboard editor looks, and so forth are all decided by your Xcode version.

Not so in server-side Swift. You might code on macOS and deploy to Rackspace Cloud

running Ubuntu 16.04, or you might code on Linux and deploy to IBM Bluemix running Ubuntu 14.04. Each of those environments are very different, and although Swift itself will be the same everywhere the environment it runs in will vary.

No matter what you choose, ultimately your code needs to run on Linux. That in turn means it needs to avoid using the incomplete parts of Foundation, and requires you to be careful about case sensitivity in filenames.

The best way to ensure you're working correctly is to build and run your code on Linux every step of the way. You can code on macOS if you want, it doesn't really matter – but you do need to *build* and *run* on Linux, to ensure what you've written can be deployed to a real server.

If you aren't already running Linux as your desktop operating system, there are four options you can take:

1. Use virtualization software such as VMware Fusion or Parallels. This lets you install a full version of Ubuntu Linux locally, for coding and testing.
2. Set up a cloud server using something like Digital Ocean. This costs about \$5 per month, or less if you pause the instance when you don't need it.
3. Use Docker, which is a bit like an invisible virtual machine running Linux on macOS. (Note: it still reads and writes from your macOS hard disk, so it does *not* solve the case-sensitivity problem.)
4. Pretend Linux doesn't exist, and build and run everything on macOS.

Of the four options, #3 is by far the most popular. Docker is a free program you can install, it lets you create Linux containers that are pre-configured for running Kitura, and it blurs the lines between macOS and Linux because you can write your code on macOS and run it inside the Linux instance.

**Before you can continue with this book, you need to decide which of the four options you want.** You can't not choose – you need to pick one. If you're not sure, go with #3. Later projects require you to install database software, and this is significantly easier if you use Docker. Trust me on this: although it's a little harder to get started, I strongly recommend you choose Docker.

If you chose #1 or #2, please go ahead and install Ubuntu now, either locally or in your preferred cloud provider. When installation has finished run these commands:

```
sudo apt-get update  
sudo apt-get install clang libicu-dev libcurl4-openssl-dev  
libssl-dev
```

Finally, download and extract the correct Swift version for your Ubuntu version from <https://swift.org/download/#releases>.

If you chose #4, please make sure you have Xcode 8.1 installed from the Mac App Store. That's it. Yes, #4 is easy, but trust me: it's extremely easy to create problems for yourself down the line, and you'll need to troubleshoot them yourself.

If you chose #3, go to <https://docs.docker.com/docker-for-mac/> and click “Get Docker for Mac (stable)” to download the latest version of Docker. When the DMG has downloaded, double-click it to open, then drag it to Applications. Once it has finished copying, browse to your Applications folder in Finder and double-click Docker to start it – you’ll be asked to enter your admin password in order to complete setup.

## A brief Docker primer

Rather than repeat instructions for Docker in every project, I want to give them just once so you can refer back here as needed.

To make things simple, I've crafted one Docker command that does everything you need to create, start, and gain access to an environment where you can run your projects. Once you've installed the Docker app and started it, you should see a whale icon in your Mac's status bar - that shows the Docker virtualization system is running.

Open a terminal, then run these commands to create a directory on your desktop where you'll store all your server projects:

```
cd Desktop  
mkdir server  
cd server
```

**Note:** It's possible your terminal starts you at a different location. If you see “~” in there then the commands above are correct, but if you see “~/Desktop” or similar then you're already in your Desktop directory and can skip the first command.

That creates a directory called “server” and changes into it so that it's the current working directory. Now run this command:

```
docker run -itv $(pwd):/projects --name projects -w /projects -p 8089:8089 -p 8090:8090 -p 5984:5984 twostraws/server-side-swift /bin/bash
```

Yes, that's quite a lot, but it's because I've tried to cram everything into one!

What the command does is:

- Tell Docker that we want to run a command in a new container – an isolated virtual machine that will run Linux.
- Request an interactive terminal (“it”), which is something we can type commands using **/bin/bash** – the default Linux terminal.
- Attaches a volume (“v”), which means “/projects” inside the container will refer to the current working directory.
- Tells Docker that we want to map our network port 8089 to the container's port 8089, map our network port 8090 to the container's port 8090, and our network port 5984 to the container's port 5984. More on that in a moment!
- Names the container “projects” (“--name projects”) so we can refer to it more easily.
- Sets the working directory to be “/projects” so we start ready to go.
- Tells Docker to build the container using “twostraws/server-side-swift”, which is a pre-configured Docker container that I've designed to make this book easy to follow. It contains everything we need to get started.

When you run that command Docker will need to download all the components required to make the container work, including Ubuntu itself as well as my customizations. If you're curious, my customized version updates all the software, adds Swift, Curl, CouchDB, and

MySQL, ready for all the projects in this book, and is based on [IBM's own Docker container](#).

After a few minutes you'll find yourself looking at a terminal prompt like this:

```
root@f2429f0045db:/projects#
```

That means installation has now completed: you're now in the "projects" directory inside the container. This is mapped to the current working directory, so if you create any files here you're really creating files in the "server" directory on your Mac's desktop, and if you change any files in macOS those changes will be reflected in the Docker container.

Using one Docker container for all projects isn't the preferred way of working, but it's fine for beginners. When you become more experienced you'll start creating individual containers for each project, and indeed each *part* of a project, but the single-container approach makes life easier for now.

The command you ran bridged the network ports 8089, 8090, and 5984 inside the container to ports 8089, 8090, and 5984 on your Mac. What this means is that if you try access any of those ports on your Mac, you get seamlessly transferred to the container.

We did this because the Kitura server runs on port 8090 by default, so it allows us to test content directly from a web browser on your Mac. Port 5984 isn't used until project two, and 8089 isn't used until project eleven, but we need to configure both them and 8090 now, when creating the container. You'll see this in action soon enough!

Before we're done, I want to show you a few important Docker commands to help you get into and out of your container.

Right now, you see a prompt like this one:

```
root@f2429f0045db:/projects#
```

I say "like" because the string of letters after the @ sign will be unique to you – container IDs are created randomly. The # sign means you're currently root, so you have full control over the virtual system, and we'll be using that to install extra software later on.

Try running the command `exit` now, and after a few seconds you'll be back to your regular macOS terminal prompt. What you just did was quit your Docker terminal, which effectively terminates the container. It still exists on your disk, but because “/bin/bash” was the main process of the machine and we just quit it, the container will cease running.

Now that you're back on the macOS terminal, run this command: `docker ps`. This shows a list of running containers, and you'll see it doesn't show any containers. Instead, you'll see just some column headers where information would be, such as “CONTAINER ID”, “IMAGE”, and so on.

The list is empty because it shows only *running* containers, and exiting our container stopped it from running. We can ask for a list of *all* containers, running or otherwise, with this command: `docker ps -a`. Note the extra “-a” in there. This time you'll see the “projects” container listed, and it will have the status “Exited”.

We gave our container the name “projects” so that it's easy to refer to, so let's start it now: run `docker start projects` then wait a few seconds. You should see “projects” written back to your terminal window, then your regular macOS command prompt. That's Docker's extremely laconic way of saying “your container is now running.”

To verify that “projects” is active, try running the `docker ps` again. This time you'll see your container listed with the status “Up about a minute”, showing that it's active. This time, though, it's active without us having a Bash terminal inside, so we can carry on working inside the macOS terminal as if Docker weren't there.

So, now we have a fully configured Kitura Docker container up and running, and it's effectively invisible – you could quit the Mac's Terminal app altogether and Linux would carry on running in the background. This is the preferred state for containers once you've finished development: you start them up, then forget about them.

Of course, sometimes you're going to want to get back into the container to make changes. Later on, for example, we're going to be installing MySQL inside the container so that we have access to a database. To do that, you need to *attach* to a container that's running – to connect to its existing Bash terminal so you're back at the root prompt.

To attach to the “projects” container, run this command: `docker attach projects`.

You might need to press Return a couple of times, but after a few seconds you should be back at the root prompt inside your container. During development – when you need to run builds frequently – I usually keep the container attached for easy access.

**Tip:** If you want to start your container and attach at the same time, use `docker start -i projects`.

Now that you’re back inside the container, running `exit` will make it terminate just like before. More commonly you’ll want to *detach* from the container, which means “get me back to my macOS terminal, but leave the container running.” This is done using two special keystrokes: Ctrl+p followed by Ctrl+q. These should be pressed one after the other rather than together.

When you press those two, you’ll immediately be returned to your macOS terminal, but the “projects” container will carry on running. You can attach again using `docker attach`, then detach, attach, detach, and so on; it will just carry on running.

If you ever want to destroy the container, use `docker rm projects`. If it’s currently running you’ll be told to stop it first, or add the `-f` parameter to force removal.

One last thing before we’re done with Docker: once you’ve created your Kitura container, Docker automatically saves the image required to recreate it. This means you can create more containers from the same image – “ibmcom/kitura-ubuntu” – in just a few seconds, because Docker doesn’t need to download it again.

Once again, Docker really is the preferred approach to server-side Swift development. With the setup above, you’re able to write your code using your preferred macOS tools, test using your preferred macOS web browser, but build and run using Linux. This means you’re in the perfect position to transfer your code to a real server when you’re ready, without having to constantly transfer files along the way.

# Introduction

If you want to learn the language all at once before you start making apps, this is for you.

## How to install Xcode and create a playground

Xcode is Apple's programming application for developers. It's free from the Mac App Store, and it's required to do iPhone and iPad development. So, your first action is to [click here to install Xcode from the Mac App Store](#) – it's quite a big download, so start downloading it now and carry on reading.

While that's downloading, I can explain a couple of the absolute basics to you:

- **iOS** is the name of the operating system that runs on all iPhones and iPads. It's responsible for all the basic operations of the phone, such as making phone calls, drawing on the screen, and running apps.
- **macOS** is the name for Apple's desktop operating system, which is the technological grandparent of iOS, tvOS, and even watchOS.
- **Swift** is Apple's modern programming language that lets you write apps for iOS, macOS, and other platforms. It contains the functionality for building programs, but doesn't handle anything like user interfaces, audio or networking.
- **Swift 1.2** was the first major update to Swift, tweaking various language features and improving others.
- **Swift 2** was the second major update to Swift, introducing checked exceptions, and many other major improvements.
- **Swift 2.2** was a minor update to Swift 2.0, deprecating some syntax ahead of its removal in Swift 3.
- **Swift 3** is the third major update to Swift, and is the version used throughout Hacking with Swift.
- **UIKit** is Apple's user interface toolkit. It contains things like buttons, text boxes, navigation controls and more, and you drive it using Swift.
- **Cocoa Touch** is the name commonly used for Apple's vast collection of frameworks for iOS. It includes UIKit to do user interfaces, but also SpriteKit for making 2D games, SceneKit for making 3D games, MapKit for maps, Core Graphics for drawing, Core Animation for animating things, and much more.
- **Cocoa** is the name used for Apple's framework collection on macOS. Strictly speaking it's made up of AppKit for user interface, Foundation for basic functionality, and Core Data for object graphs, but like Cocoa Touch it's often used to mean "all of macOS development."

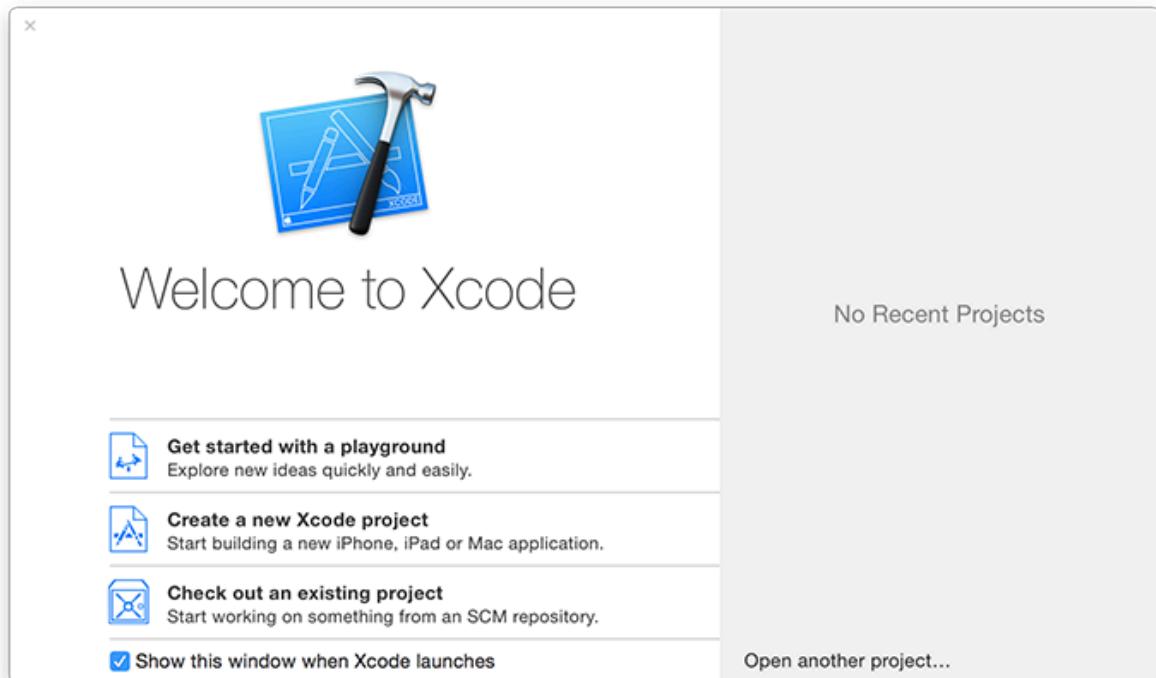
- **NeXTSTEP** is an operating system created by a company that Steve Jobs founded called NeXT. It was bought by Apple, at which point Jobs was placed back in control of the company, and put NeXTSTEP technology right into the core of Apple's development platform.
- **iOS Simulator** is a tool that comes with Xcode that looks and works almost exactly like a real iPhone or iPad. It lets you test iOS apps very quickly without having to use a real device.
- **Playgrounds** are miniature Swift testing environments that let you type code and see the results immediately. You don't build real apps with them, but they are great for learning. We'll be using playgrounds in this introduction.
- **Crashes** are when your code goes disastrously wrong and your app cannot recover. If a user is running your app it will just disappear and they'll be back on the home screen. If you're running in Xcode, you'll see a crash report.
- **Taylor Swift** has nothing to do with the Swift programming language. This is a shame, as you might imagine, but I'll try to make up for this shortfall by using her songs in this tutorial. Deal with it.

That's it for the basics – if Xcode still hasn't finished downloading then why not watch some Taylor Swift videos while you wait? The examples in this tutorial will certainly make a lot more sense...

**Got Xcode installed? OK! Let's do this...**

## **Introduction to Swift playgrounds**

When you launch Xcode, you'll see something like the picture below. Look for the "Get started with a playground" button on the lower left, and click that. Xcode will ask you to name your playground, but "MyPlayground" is fine. When you click Next you'll be asked where to save it, so please choose your desktop and click Create.



Xcode will ask you whether you want to create a playground for iOS or macOS, but it doesn't matter here – this introduction is almost exclusively about the Swift language, with no user interface components. For the avoidance of problems, leave “iOS” selected for the platform.

What you'll see is a window split in two. On the left you'll see this:

```
//: Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"
```

And on the right, you'll see this: "Hello, playground".

This split is important, because it divides code and results. The code is in the left pane, and you will edit this to do your own Swift work as we go. The results are in the right pane, and it shows you what your Swift code has done. In this case, it's telling us that we successfully set the value "Hello, playground."

You will also notice that the very first line of the playground starts with two slashes, `//`. When

Swift sees two slashes like that, it ignores everything after them on a line. This is commonly used for comments: notes that you write into your code to help you understand what it does later.

As you type, the playground will automatically run your code and show the updated results. For example, if you just write **str** by itself, you'll see "Hello, Playground" twice on the right – once because it's being set, and once because you're printing the value.

Playgrounds are a great way to try some code and see the results immediately. They are extremely powerful too, as you'll see over the next hour or so. Let's get started writing Swift!

# Variables and constants

Every useful program needs to store data at some point, and in Swift there are two ways to do it: variables and constants. A variable is a data store that can have its value changed whenever you want, and a constant is a data store that you set once and can never change. So, variables have values that can vary, and constants have values that are constant – easy, right?

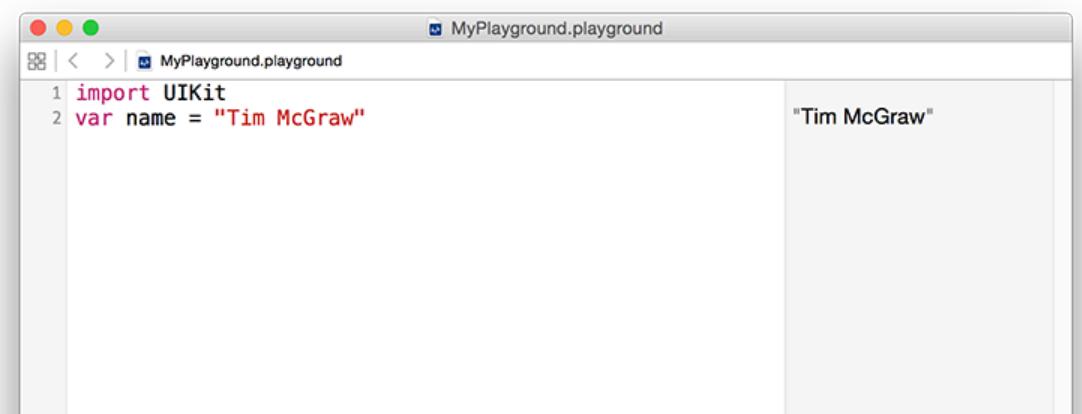
Having both these options might seem pointless, after all you could just create a variable then never change it – why does it need to be made a constant? Well, it turns out that many programmers are – shock! – less than perfect at programming, and we make mistakes. One of the advantages of separating constants and variables is that Xcode will tell us if we've made a mistake. If we say, "make this date a constant, because I know it will never change" then 10 lines later try to change it, Xcode will refuse to build our app.

Constants are also important because they let Xcode make decisions about the way it builds your app. If it knows a value will never change, it is able to apply optimizations to make your code run faster.

In Swift, you make a variable using the **var** keyword, like this:

```
var name = "Tim McGraw"
```

Let's put that into a playground so you can start getting feedback. Delete everything in there apart from the **import UIKit** line (that's the bit that pulls in Apple's core iOS framework and it's needed later on), and add that variable. You should see the picture below.



Because this is a variable, you can change it whenever you want, but you shouldn't use the **var** keyword each time – that's only used when you're declaring new variables. Try writing this:

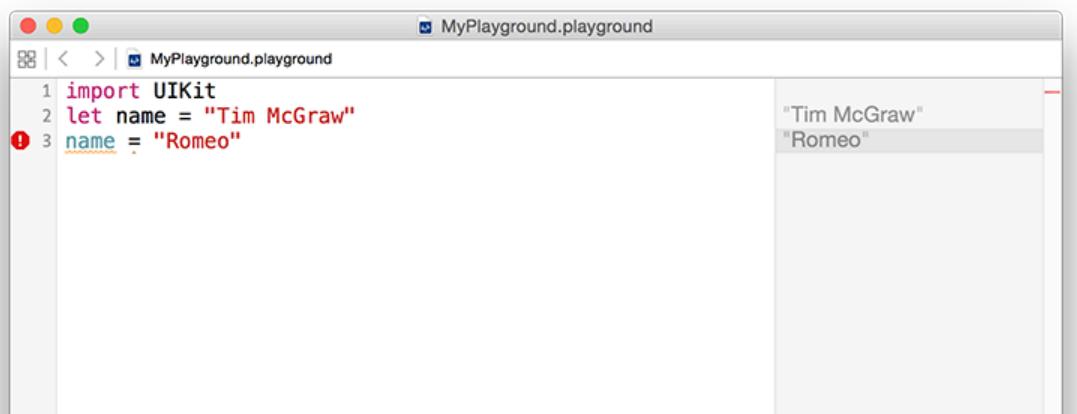
```
var name = "Tim McGraw"  
name = "Romeo"
```

So, the first line creates the **name** variable and gives it an initial value, then the second line updates the **name** variable so that its value is now "Romeo". You'll see both values printed in the results area of the playground.

Now, what if we had made that a constant rather than a variable? Well, constants use the **let** keyword rather than **var**, so you can change your first line of code to say **let name** rather than **var name** like this:

```
import UIKit  
let name = "Tim McGraw"  
name = "Romeo"
```

But now there's a problem: Xcode is showing a red warning symbol next to line three, and it should have drawn a squiggly underline underneath **name**. If you click the red warning symbol, Xcode will tell you the problem: "Cannot assign to 'let' value 'name'" – which is Xcode-speak for "you're trying to change a constant and you can't do that."



So, constants are a great way to make a promise to Swift and to yourself that a value won't

change, because if you do try to change it Xcode will refuse to run. Swift developers have a strong preference to use constants wherever possible because it makes your code easier to understand. In fact, in the very latest versions of Swift, Xcode will actually tell you if you make something a variable then never change it!

**Important note:** variable and constant names must be unique in your code. You'll get an error if you try to use the same variable name twice, like this:

```
var name = "Tim McGraw"  
var name = "Romeo"
```

If the playground finds an error in your code, it will either flag up a warning in a red box, or will just refuse to run. You'll know if the latter has happened because the text in the results pane has gone gray rather than its usual black.

# Types of Data

There are lots of kinds of data, and Swift handles them all individually. You already saw one of the most important types when you assigned some text to a variable, but in Swift this is called a **String** – literally a string of characters.

Strings can be long (e.g. a million letters or more), short (e.g. 10 letters) or even empty (no letters), it doesn't matter: they are all strings in Swift's eyes, and all work the same. Swift knows that **name** should hold a string because you assign a string to it when you create it: "Tim McGraw". If you were to rewrite your code to this it would stop working:

```
var name  
name = "Tim McGraw"
```

This time Xcode will give you an error message that won't make much sense just yet: "Type annotation missing in pattern". What it means is, "I can't figure out what data type **name** is because you aren't giving me enough information."

At this point you have two options: either create your variable and give it an initial value on one line of code, or use what's called a type annotation, which is where you tell Swift what data type the variable will hold later on, even though you aren't giving it a value right now.

You've already seen how the first option looks, so let's look at the second: type annotations. We know that **name** is going to be a string, so we can tell Swift that by writing a colon then **String**, like this:

```
var name: String  
name = "Tim McGraw"
```

You'll have no errors now, because Swift knows what type of data **name** will hold in the future.

**Note:** some people like to put a space before and after the colon, making **var name : String**, but they are *wrong* and you should try to avoid mentioning their wrongness in polite company.

The lesson here is that Swift always wants to know what type of data every variable or

constant will hold. Always. You can't escape it, and that's a good thing because it provides something called type safety – if you say "this will hold a string" then later try and put a rabbit in there, Swift will refuse.

We can try this out now by introducing another important data type, called **Int**, which is short for "integer." Integers are round numbers like 3, 30, 300, or -16777216. For example:

```
var name: String  
name = "Tim McGraw"
```

```
var age: Int  
age = 25
```

That declares one variable to be a string and one to be an integer. Note how both **String** and **Int** have capital letters at the start, whereas **name** and **age** do not – this is the standard coding convention in Swift. A coding convention is something that doesn't matter to Swift (you can write your names how you like!) but does matter to other developers. In this case, data types start with a capital letter, whereas variables and constants do not.

Now that we have variables of two different types, you can see type safety in action. Try writing this:

```
name = 25  
age = "Time McGraw"
```

In that code, you're trying to put an integer into a string variable, and a string into an integer variable – and, thankfully, Xcode will throw up errors. You might think this is pedantic, but it's actually quite helpful: you make a promise that a variable will hold one particular type of data, and Xcode will enforce that throughout your work.

**Before you go on, please delete those two lines of code causing the error, otherwise nothing in your playground will work going forward!**

## Float and Double

Let's look at two more data types, called **Float** and **Double**. This is Swift's way of storing numbers with a fractional component, such as 3.1, 3.141, 3.1415926, and -16777216.5. There are two data types for this because you get to choose how much accuracy you want, but most of the time it doesn't matter so the official Apple recommendation is always to use **Double** because it has the highest accuracy.

Try putting this into your playground:

```
var latitude: Double  
latitude = 36.166667  
  
var longitude: Float  
longitude = -86.783333
```

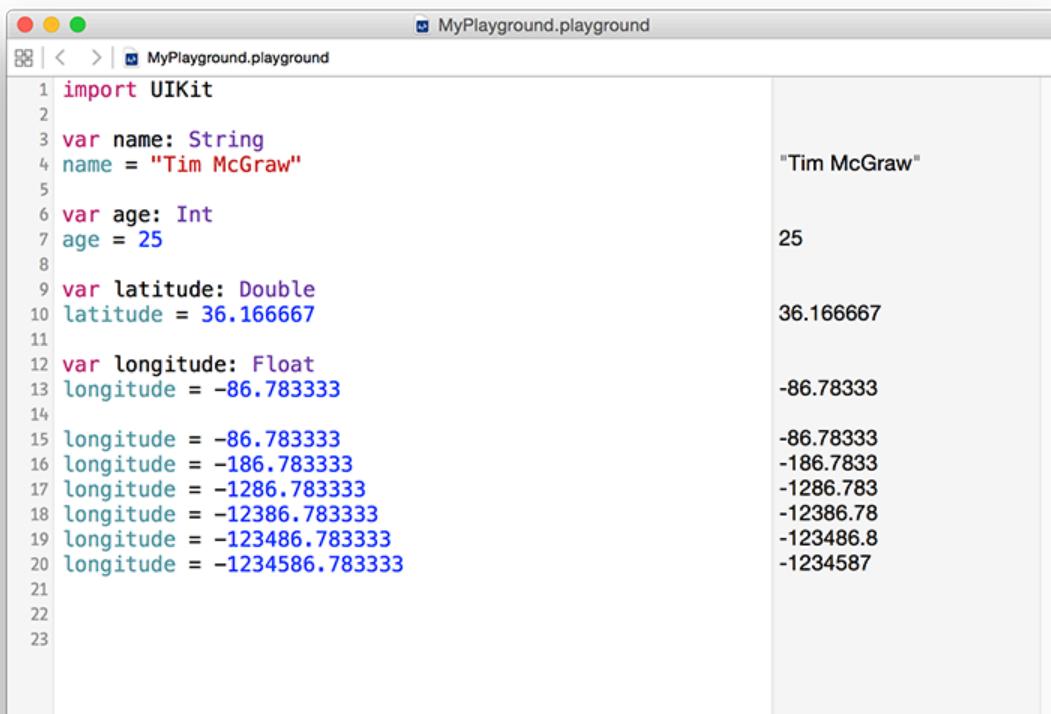
You can see both numbers appear on the right, but look carefully because there's a tiny discrepancy. We said that **longitude** should be equal to -86.78333, but in the results pane you'll see -86.78333 – it's missing one last 3 on the end. Now, you might well say, "what does 0.000003 matter among friends?" but this is ably demonstrating what I was saying about accuracy.

Because these playgrounds update as you type, we can try things out so you can see exactly how **Float** and **Double** differ. Try changing the code to be this:

```
var longitude: Float  
longitude = -86.78333  
longitude = -186.78333  
longitude = -1286.78333  
longitude = -12386.78333  
longitude = -123486.78333  
longitude = -1234586.78333
```

That's adding increasing numbers before the decimal point, while keeping the same amount of numbers after. But if you look in the results pane you'll notice that as you add more numbers before the point, Swift is removing numbers after. This is because it has limited space in which to store your number, so it's storing the most important part first – being off by 1,000,000 is a

big thing, whereas being off by 0.000003 is less so.



The screenshot shows a Xcode playground window titled "MyPlayground.playground". The code in the left pane is as follows:

```
1 import UIKit
2
3 var name: String
4 name = "Tim McGraw"
5
6 var age: Int
7 age = 25
8
9 var latitude: Double
10 latitude = 36.166667
11
12 var longitude: Float
13 longitude = -86.783333
14
15 longitude = -86.783333
16 longitude = -186.783333
17 longitude = -1286.783333
18 longitude = -12386.783333
19 longitude = -123486.783333
20 longitude = -1234586.783333
21
22
23
```

The right pane shows the output of the code. The variables are listed with their values:

Variable	Value
name	"Tim McGraw"
age	25
latitude	36.166667
longitude (Float)	-86.783333
longitude (Double) 15	-86.783333
longitude (Double) 16	-186.783333
longitude (Double) 17	-1286.783333
longitude (Double) 18	-12386.783333
longitude (Double) 19	-123486.783333
longitude (Double) 20	-1234586.783333

Now try changing the **Float** to be a **Double** and you'll see Swift prints the correct number out every time:

```
var longitude: Double
```

This is because, again, **Double** has twice the accuracy of **Float** so it doesn't need to cut your number to fit. Doubles still have limits, though – if you were to try a massive number like 123456789.123456789 you would see it gets cut down to 123456789.1234568.

## Boolean

Swift has a built-in data type that can store whether a value is true or false, called a **Bool**, short for Boolean. Booleans don't have space for "maybe" or "perhaps", only absolutes: true or false. For example:

```
var stayOutTooLate: Bool
stayOutTooLate = true
```

```
var nothingInBrain: Bool  
nothingInBrain = true  
  
var missABeat: Bool  
missABeat = false
```

## Using type annotations wisely

As you've learned, there are two ways to tell Swift what type of data a variable holds: assign a value when you create the variable, or use a type annotation. If you have a choice, the first is always preferable because it's clearer. For example:

```
var name = "Tim McGraw"
```

...is preferred to:

```
var name: String  
name = "Tim McGraw"
```

This applies to all data types. For example:

```
var age = 25  
var longitude = -86.783333  
var nothingInBrain = true
```

This technique is called *type inference*, because Swift can infer what data type should be used for a variable by looking at the type of data you want to put in there. When it comes to numbers like -86.783333, Swift will always infer a **Double** rather than a **Float**.

For the sake of completeness, I should add that it's possible to specify a data type and provide a value at the same time, like this:

```
var name: String = "Tim McGraw"
```

# Operators

Operators are those little symbols you learned in your very first math classes: `+` to add, `-` to subtract, `*` to multiply, `/` to divide, `=` to assign value, and so on. They all exist in Swift, along with a few extras.

Let's try a few basics – please type this into your playground:

```
var a = 10  
a = a + 1  
a = a - 1  
a = a * a
```

In the results pane, you'll see 10, 11, 10 and 100 respectively. Now try this:

```
var b = 10  
b += 10  
b -= 10
```

`+=` is an operator that means "add then assign to." In our case it means "take the current value of `b`, add 10 to it, then put the result back into `b`." As you might imagine, `-=` does the same but subtracts rather than adds. So, that code will show 10, 20, 10 in the results pane.

Some of these operators apply to other data types. As you might imagine, you can add two doubles together like this:

```
var a = 1.1  
var b = 2.2  
var c = a + b
```

When it comes to strings, `+` will join them together. For example:

```
var name1 = "Tim McGraw"  
var name2 = "Romeo"  
var both = name1 + " and " + name2
```

That will write "Tim McGraw and Romeo" into the results pane.

One more common operator you'll see is called modulus, and is written using a percent symbol: %. It means "divide the left hand number evenly by the right, and return the remainder." So, **9 % 3** returns 0 because 9 divides evenly into 3, whereas **10 % 3** returns 1, because 9 divides into 3 three times, with remainder 1.

**Note:** If you bought Hacking with Swift and are using the exclusive guide book accompaniment to the course, you'll find the modulus operator useful later on.

## Comparison operators

Swift has a set of operators that perform comparisons on values. For example:

```
var a = 1.1
var b = 2.2
var c = a + b

c > 3
c >= 3
c > 4
c < 4
```

That shows off greater than (>), greater than or equal (>=), and less than (<). In the results window you'll see true, true, false, true – these are Booleans, because the answer to each of these statements can only ever be true or false.

If you want to check for equality, you can't use = because it already has a meaning: it's used to give a variable a value. So, Swift has an alternative in the form of ==, meaning "is equal to." For example:

```
var name = "Tim McGraw"
name == "Tim McGraw"
```

That will show "true" in the results pane. Now, one thing that might catch you out is that in

Swift strings are case-sensitive, which means "Tim McGraw", "TIM MCGRAW" and "TiM mCgRaW" are all considered different. If you use `==` to compare two strings, you need to make sure they have the same letter case.

There's one more operator I want to introduce you to, and it's called the "not" operator: `!`. Yes, it's just an exclamation mark. This makes your statement mean the opposite of what it did. For example:

```
var stayOutTooLate = true
stayOutTooLate
!stayOutTooLate
```

That will print out true, true, false – with the last value there because it flipped the previous true.

You can also use `!` with `=` to make `!=` or "not equal". For example:

```
var name = "Tim McGraw"
name == "Tim McGraw"
name != "Tim McGraw"
```

# String interpolation

This is a fancy name for what is actually a very simple thing: combining variables and constants inside a string.

Clear out all the code you just wrote and leave only this:

```
var name = "Tim McGraw"
```

If we wanted to print out a message to the user that included their name, string interpolation is what makes that easy: you just write a backslash, then an open parenthesis, then your code, then a close parenthesis, like this:

```
var name = "Tim McGraw"  
"Your name is \(name)"
```

The results pane will now show "Your name is Tim McGraw" all as one string, because string interpolation combined the two for us.

Now, we could have written that using the `+` operator, like this:

```
var name = "Tim McGraw"  
"Your name is " + name
```

...but that's not as efficient, particularly if you're combining multiple variables together. In addition, string interpolation in Swift is smart enough to be able to handle a variety of different data types automatically. For example:

```
var name = "Tim McGraw"  
var age = 25  
var latitude = 36.166667  
  
"Your name is \(name), your age is \(age), and your latitude is  
\(latitude)"
```

Doing that using `+` is much more difficult, because Swift doesn't let you add integers and

doubles to a string.

At this point your result may no longer fit in the results pane, so either resize your window or hover over the result and click the + button that appears to have it shown inline.

One of the powerful features of string interpolation is that everything between `\(` and `)` can actually be a full Swift expression. For example, you can do mathematics in there using operators, like this:

```
var age = 25
"You are \(age) years old. In another \(age) years you will be
\(age * 2)."
```

# Arrays

Arrays let you group lots of values together into a single collection, then access those values by their position in the collection. Swift uses type inference to figure out what type of data your array holds, like so:

```
var evenNumbers = [2, 4, 6, 8]
var songs = ["Shake it Off", "You Belong with Me", "Back to
December"]
```

As you can see, Swift uses brackets to mark the start and end of an array, and each item in the array is separated with a comma.

When it comes to reading items out an array, there's a catch: Swift starts counting at 0. This means the first item is 0, the second item is 1, the third is 2, and so on. Try putting this into your playground:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to
December"]
songs[0]
songs[1]
songs[2]
```

That will print "Shake it Off", "You Belong with Me", and "Back to December" in the results pane.

An item's position in an array is called its index, and you can read any item from the array just by providing its index. However, you do need to be careful: our array has three items in, which means indexes 0, 1 and 2 work great. But if you try and read **songs[3]** your playground will stop working – and if you tried that in a real app it would crash!

Because you've created your array by giving it three strings, Swift knows this is an array of strings. You can confirm this by using a special command in the playground that will print out the data type of any variable, like this:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to
```

```
December"]  
type(of: songs)
```

That will print `Array<String>.Type` into the results pane, telling you that Swift considers `songs` to be an array of strings.

Let's say you made a mistake, and accidentally put a number on the end of the array. Try this now and see what the results pane prints:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to  
December", 3]  
type(of: songs)
```

This time you'll see an error. The error isn't because Swift can't handle mixed arrays like this one – I'll show you how to do that in just a moment! – but instead because Swift is being helpful. The error message you'll see is, “heterogenous collection literal could only be inferred to '[Any]'; add explicit type annotation if this is intentional.” Or, in plain English, “it looks like this array is designed to hold lots of types of data – if you really meant that, please make it explicit.”

Type safety is important, and although it's neat that Swift can make arrays hold any kind of data this particular case was an accident. Fortunately, I've already said that you can use type annotations to specify exactly what type of data you want an array to store. To specify the type of an array, write the data type you want to store with brackets around it, like this:

```
var songs: [String] = ["Shake it Off", "You Belong with Me",  
"Back to December", 3]
```

Now that we've told Swift we want to store only strings in the array, it will always refuse to run the code because 3 is not a string.

If you really want the array to hold any kind of data, use the special `Any` data type, like this:

```
var songs: [Any] = ["Shake it Off", "You Belong with Me", "Back  
to December", 3]
```

## Creating arrays

If you make an array using the syntax shown above, Swift creates the array and fills it with the values we specified. Things aren't quite so straightforward if you want to create the array then fill it later – this syntax doesn't work:

```
var songs: [String]  
songs[0] = "Shake it Off"
```

The reason is one that will seem needlessly pedantic at first, but has deep underlying performance implications so I'm afraid you're just stuck with it. Put simply, writing **var songs: [String]** tells Swift "the **songs** variable will hold an array of strings," but *it doesn't actually create that array*. It doesn't allocate any RAM, or do any of the work to actually create a Swift array. It just says that at some point there will be an array, and it will hold strings.

There are a few ways to express this correctly, and the one that probably makes most sense at this time is this:

```
var songs: [String] = []
```

That uses a type annotation to make it clear we want an array of strings, and it assigns an empty array (that's the **[]** part) to it.

You'll also commonly see this construct:

```
var songs = [String]()
```

That means the same thing: the **()** tells Swift we want to create the array in question, which is then assigned to **songs** using type inference. This option is two characters shorter, so it's no surprise programmers prefer it!

## Array operators

You can use a limited set of operators on arrays. For example, you can merge two arrays by using the `+` operator, like this:

```
var songs = ["Shake it Off", "You Belong with Me", "Love Story"]
var songs2 = ["Today was a Fairytale", "Welcome to New York",
    "Fifteen"]
var both = songs + songs2
```

You can also use `+=` to add and assign, like this:

```
both += ["Everything has Changed"]
```

# Dictionaries

As you've seen, Swift arrays are a collection where you access each item using a numerical index, such as `songs[0]`. Dictionaries are another common type of collection, but they differ from arrays because they let you access values based on a key you specify.

To give you an example, let's imagine how we might store data about a person in an array:

```
var person = [ "Taylor", "Alison", "Swift", "December",
  "taylorswift.com" ]
```

To read out that person's middle name, we'd use `person[1]`, and to read out the month they were born we'd use `person[3]`. This has a few problems, not least that it's difficult to remember what index number is assigned to each value in the array! And what happens if the person has no middle name? Chances are all the other values would move down one place, causing chaos in your code.

With dictionaries we can re-write this to be far more sensible, because rather than arbitrary numbers you get to read and write values using a key you specify. For example:

```
var person = [ "first": "Taylor", "middle": "Alison", "last": "Swift",
  "month": "December", "website": "taylorswift.com" ]
person["middle"]
person["month"]
```

It might help if I use lots of whitespace to break up the dictionary on your screen, like this:

```
var person = [
    "first": "Taylor",
    "middle": "Alison",
    "last": "Swift",
    "month": "December",
    "website": "taylorswift.com"
]

person["middle"]
```

```
person[ "month" ]
```

As you can see, when you make a dictionary you write its key, then a colon, then its value. You can then read any value from the dictionary just by knowing its key, which is much easier to work with.

As with arrays, you can store a wide variety of values inside dictionaries, although the keys are most commonly strings.

# Conditional statements

Sometimes you want code to execute only if a certain condition is true, and in Swift that is represented primarily by the **if** and **else** statements. You give Swift a condition to check, then a block of code to execute if that condition is true.

You can optionally also write **else** and provide a block of code to execute if the condition is false, or even **else if** and have more conditions. A "block" of code is just a chunk of code marked with an open brace – **{** – at its start and a close brace – **}** – at its end.

Here's a basic example:

```
var action: String
var person = "hater"

if person == "hater" {
    action = "hate"
}
```

That uses the **==** (equality) operator introduced previously to check whether the string inside **person** is exactly equivalent to the string "hater". If it is, it sets the **action** variable to "hate". Note that open and close braces, also known by their less technical name of "curly brackets" – that marks the start and end of the code that will be executed if the condition is true.

Let's add **else if** and **else** blocks:

```
var action: String
var person = "hater"

if person == "hater" {
    action = "hate"
} else if person == "player" {
    action = "play"
} else {
    action = "cruise"
```

```
}
```

That will check each condition in order, and only one of the blocks will be executed: a person is either a hater, a player, or anything else.

## Evaluating multiple conditions

You can ask Swift to evaluate as many conditions as you want, but they all need to be true in order for Swift to execute the block of code. To check multiple conditions, use the **&&** operator – it means "and". For example:

```
var action: String
var stayOutTooLate = true
var nothingInBrain = true

if stayOutTooLate && nothingInBrain {
    action = "cruise"
}
```

Because **stayOutTooLate** and **nothingInBrain** are both true, the whole condition is true, and **action** gets set to "cruise." Swift uses something called short-circuit evaluation to boost performance: if it is evaluating multiple things that all need to be true, and the first one is false, it doesn't even bother evaluating the rest.

## Looking for the opposite of truth

This might sound deeply philosophical, but actually this is important: sometimes you care whether a condition is not true, i.e. is false. You can do this with the **!** (not) operator that was introduced earlier. For example:

```
if !stayOutTooLate && !nothingInBrain {
    action = "cruise"
}
```

This time, the **action** variable will only be set if both **stayOutTooLate** and

**nothingInBrain** are false – the `!` has flipped them around.

# Loops

Computers are great at doing boring tasks billions of times in the time it took you to read this sentence. When it comes to repeating tasks in code, you can either copy and paste your code multiple times, or you can use *loops* – simple programming constructs that repeat a block of code for as long as a condition is true.

To demonstrate this, I want to introduce you to a special debugging function called **print()**: you give it some text to print, and it will print it. If you're running in a playground like we are, you'll see your text appear in the results window. If you're running a real app in Xcode, you'll see your text appear in Xcode's log window. Either way, **print()** is a great way to get a sneak peek at the contents of a variable.

Take a look at this code:

```
print("1 x 10 is \u00d7(1 * 10)")  
print("2 x 10 is \u00d7(2 * 10)")  
print("3 x 10 is \u00d7(3 * 10)")  
print("4 x 10 is \u00d7(4 * 10)")  
print("5 x 10 is \u00d7(5 * 10)")  
print("6 x 10 is \u00d7(6 * 10)")  
print("7 x 10 is \u00d7(7 * 10)")  
print("8 x 10 is \u00d7(8 * 10)")  
print("9 x 10 is \u00d7(9 * 10)")  
print("10 x 10 is \u00d7(10 * 10)")
```

When it has finished running, you'll have the 10 times table in your playground results pane. But it's hardly efficient code, and in fact a much cleaner way is to loop over a range of numbers using what's called the closed range operator, which is three periods in a row: **...**

Using the closed range operator, we could re-write that whole thing in three lines:

```
for i in 1...10 {  
    print("\u00d7(i) x 10 is \u00d7(i * 10)")  
}
```

The results pane just shows "(10 times)" for our loop, meaning that the loop was run 10 times. If you want to know what the loop actually did, hover over the "(10 times)" then click the + button that appears on the right. You'll see a box saying "10 x 10 is 100" appear inside your code, and if you right-click on that you should see the option "Value History". Click on that now, and you should see the picture below:

A screenshot of an Xcode playground window titled "MyPlayground.playground". The code in the editor is as follows:

```

1 import UIKit
2
3 print("1 x 10 is \(1 * 10)")
4 print("2 x 10 is \(2 * 10)")
5 print("3 x 10 is \(3 * 10)")
6 print("4 x 10 is \(4 * 10)")
7 print("5 x 10 is \(5 * 10)")
8 print("6 x 10 is \(6 * 10)")
9 print("7 x 10 is \(7 * 10)")
10 print("8 x 10 is \(8 * 10)")
11 print("9 x 10 is \(9 * 10)")
12 print("10 x 10 is \(10 * 10)")
13
14 for i in 1...10 {
15     print("\(i) x 10 is \(i * 10)")
16 }

```

The results pane shows the output of the code:

```

"1 x 10 is 10"
"2 x 10 is 20"
"3 x 10 is 30"
"4 x 10 is 40"
"5 x 10 is 50"
"6 x 10 is 60"
"7 x 10 is 70"
"8 x 10 is 80"
"9 x 10 is 90"
"10 x 10 is 100"
(10 times)

```

A callout box highlights the value history for the variable `i`, showing the values 1 through 10.

What the loop does is count from 1 to 10 (including 1 and 10), assigns that number to the constant `i`, then runs the block of code inside the braces.

If you don't need to know what number you're on, you can use an underscore instead. For example, we could print some Taylor Swift lyrics like this:

```

var str = "Fakers gonna"

for _ in 1 ... 5 {
    str += " fake"
}

print(str)

```

That will print "Fakers gonna fake fake fake fake" by adding to the string each time the

loop goes around.

If Swift doesn't have to assign each number to a variable each time the loop goes around, it can run your code a little faster. As a result, if you write **for i in...** then don't use **i**, Xcode will suggest you change it to \_.

There's a variant of the closed range operator called the half open range operator, and they are easily confused. The half open range operator looks like **..**<**** and counts from one number up to and *excluding* another. For example, **1 ..< 5** will count 1, 2, 3, 4.

## Looping over arrays

Swift provides a very simple way to loop over all the elements in an array. Because Swift already knows what kind of data your array holds, it will go through every element in the array, assign it to a constant you name, then run a block of your code. For example, we could print out a list of great songs like this:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to December"]  
  
for song in songs {  
    print("My favorite song is \(song)")  
}
```

You can also use the **for i in** loop construct to loop through arrays, because you can use that constant to index into an array. We could even use it to index into two arrays, like this:

```
var people = ["players", "haters", "heart-breakers", "fakers"]  
var actions = ["play", "hate", "break", "fake"]  
  
for i in 0 ... 3 {  
    print("\(people[i]) gonna \(actions[i])")  
}
```

You might wonder what use the half open range operator has, but it's particularly useful for

working with arrays because they count from zero. So, rather than counting from 0 up to and including 3, we could count from 0 up to and *excluding* the number of items in an array.

Remember: they count from zero, so if they have 4 items the maximum index is 3, which is why we need to use *excluding* for the loop.

To count how many items are in an array, use **someArray.count**. So, we could rewrite our code like this:

```
var people = ["players", "haters", "heart-breakers", "fakers"]
var actions = ["play", "hate", "break", "fake"]

for i in 0 ...< people.count {
    print("\(people[i]) gonna \(actions[i])")
}
```

## Inner loops

You can put loops inside loops if you want, and even loops inside loops inside loops – although you might suddenly find you're doing something 10 million times, so be careful!

We can combine two of our previous loops to create this:

```
var people = ["players", "haters", "heart-breakers", "fakers"]
var actions = ["play", "hate", "break", "fake"]

for i in 0 ...< people.count {
    var str = "\(people[i]) gonna"

    for _ in 1 ... 5 {
        str += " \(actions[_])"
    }

    print(str)
}
```

That outputs "players gonna play play play play play", then "haters gonna..." Well, you get the idea.

One important note: although programmers conventionally use **i**, **j** and even **k** for loop constants, you can name them whatever you please: **for personNumber in 0 ..< people.count** is perfectly valid.

## While loops

There's a third kind of loop you'll see, which repeats a block of code until you tell it to stop. This is used for things like game loops where you have no idea in advance how long the game will last – you just keep repeating "check for touches, animate robots, draw screen, check for touches..." and so on, until eventually the user taps a button to exit the game and go back to the main menu.

These loops are called **while** loops, and they look like this:

```
var counter = 0

while true {
    print("Counter is now \(counter)")
    counter += 1

    if counter == 556 {
        break
    }
}
```

That code introduces a new keyword, called **break**. It's used to exit a **while** or **for** loop at a point you decide. Without it, the code above would never end because the condition to check is just "true", and true is always true. Without that **break** statement the loop is an infinite loop, which is A Bad Thing.

These **while** loops work best when you're using unknown data, such as downloading things from the internet, reading from a file such as XML, looking through user input, and so on. This

is because you only know when to stop the loop after you've run it a sufficient number of times.

There is a counterpart to **break** called **continue**. Whereas breaking out of a loop stops execution immediately and continues directly after the loop, continuing a loop only exits the current iteration of the loop – it will jump back to the top of the loop and pick up from there.

As an example, consider the code below:

```
var songs = ["Shake it Off", "You Belong with Me", "Back to December"]  
  
for song in songs {  
    if song == "You Belong with Me" {  
        continue  
    }  
  
    print("My favorite song is \(song)")  
}
```

That loops through three Taylor Swift songs, but it will only print the name of two. The reason for this is the **continue** keyword: when the loop tries to use the song "You Belong with Me", **continue** gets called, which means the loop immediately jumps back to the start – the **print()** is call never made, and instead the loop continues straight on to **Back to December**.

## Switch case

You've seen **if** statements and now loops, but Swift has another type of flow control called **switch/case**. It's easiest to think of this as being an advanced form of **if**, because you can have lots of matches and Swift will execute the right one.

In the most basic form of a **switch/case** you tell Swift what variable you want to check, then provide a list of possible cases for that variable. Swift will find the first case that matches your variable, then run its block of code. When that block finishes, Swift exits the whole **switch/case** block.

Here's a basic example:

```
let liveAlbums = 2

switch liveAlbums {
    case 0:
        print("You're just starting out")

    case 1:
        print("You just released iTunes Live From SoHo")

    case 2:
        print("You just released Speak Now World Tour")

    default:
        print("Have you done something new?")
}
```

We could very well have written that using lots of **if** and **else if** blocks, but this way is clearer and that's important.

One advantage to **switch/case** is that Swift will ensure your cases are exhaustive. That is, if there's the possibility of your variable having a value you don't check for, Xcode will refuse to build your app. In situations where the values are effectively open ended, like our **liveAlbums** integer, you need to include a **default** case to catch these potential values.

Yes, even if you "know" your data can only fall within a certain range.

Swift can apply some evaluation to your case statements in order to match against variables. For example, if you wanted to check for a range of possible values, you could use the closed range operator like this:

```
let studioAlbums = 5

switch studioAlbums {
    case 0...1:
        print("You're just starting out")

    case 2...3:
        print("You're a rising star")

    case 4...5:
        print("You're world famous!")

    default:
        print("Have you done something new?")
}
```

One thing you should know is that **switch/case** blocks in Swift don't fall through like they do in some other languages you might have seen. If you're used to writing **break** in your **case** blocks, you should know this isn't needed in Swift. Instead, you use the **fallthrough** keyword to make one case fall into the next – it's effectively the opposite. Of course, if you have no idea what any of this means, that's even better: don't worry about it!

# Functions

Functions let you define re-usable pieces of code that perform specific pieces of functionality. Usually functions are able to receive some values to modify the way they work, but it's not required.

Let's start with a simple function:

```
func favoriteAlbum() {  
    print("My favorite is Fearless")  
}
```

If you put that code into your playground, nothing will be printed. And yes, it is correct. The reason nothing is printed is that we've placed the "My favorite is Fearless" message into a function called **favoriteAlbum()**, and that code won't be called until we ask Swift to run the **favoriteAlbum()** function. To do that, add this line of code:

```
favoriteAlbum()
```

That runs the function (or "calls" it), so now you'll see "My favorite is Fearless" printed out.

As you can see, you define a function by writing **func**, then your function name, then open and close parentheses, then a block of code marked by open and close braces. You then call that function by writing its name followed by an open and close parentheses.

Of course, that's a silly example – that function does the same thing no matter what, so there's no point in it existing. But what if we wanted to print a different album each time? In that case, we could tell Swift we want our function to accept a value when it's called, then use that value inside it.

Let's do that now:

```
func favoriteAlbum(name: String) {  
    print("My favorite is \(name)")  
}
```

That tells Swift we want the function to accept one value (called a "parameter"), named

"name", that should be a string. We then use string interpolation to write that favorite album name directly into our output message. To call the function now, you'd write this:

```
favoriteAlbum(name: "Fearless")
```

You might still be wondering what the point is, given that it's still just one line of code. Well, imagine we used that function in 20 different places around a big app, then your head designer comes along and tells you to change the message to "I love Fearless so much – it's my favorite!" Do you really want to find and change all 20 instances in your code? Probably not. With a function you change it once, and everything updates.

You can make your functions accept as many parameters as you want, so let's make it accept a name and a year:

```
func printAlbumRelease(name: String, year: Int) {
    print("\u{00A9}(name) was released in \u{00A9}(year)")
}

printAlbumRelease(name: "Fearless", year: 2008)
printAlbumRelease(name: "Speak Now", year: 2010)
printAlbumRelease(name: "Red", year: 2012)
```

These function parameter names are important, and actually form part of the function itself. Sometimes you'll see several functions with the same name, e.g. **handle()**, but with different parameter names to distinguish the different actions.

## External and internal parameter names

Sometimes you want parameters to be named one way when a function is called, but another way inside the function itself. This means that when you call a function it uses almost natural English, but inside the function the parameters have sensible names. This technique is employed very frequently in Swift, so it's worth understanding now.

To demonstrate this, let's write a function that prints the number of letters in a string. This is available using the **characters.count** property of strings, so we could write this:

```
func countLettersInString(string: String) {
    print("The string \(string) has \(string.characters.count)
letters.")
}
```

With that function in place, we could call it like this:

```
countLettersInString(string: "Hello")
```

While that certainly works, it's a bit wordy. Plus it's not the kind of thing you would say aloud: "count letters in string string hello".

Swift's solution is to let you specify one name for the parameter when it's being called, and another inside the method. To use this, just write the parameter name twice – once for external, one for internal.

For example, we could name the parameter **myString** when it's being called, and **str** inside the method, like this:

```
func countLettersInString(myString str: String) {
    print("The string \(str) has \(str.characters.count)
letters.")
}

countLettersInString(myString: "Hello")
```

You can also specify an underscore, \_, as the external parameter name, which tells Swift that it shouldn't have any external name at all. For example:

```
func countLettersInString(_ str: String) {
    print("The string \(str) has \(str.characters.count)
letters.")
}

countLettersInString("Hello")
```

As you can see, that makes the line of code read like an English sentence: “count letters in string hello”.

While there are many cases when using `_` is the right choice, Swift programmers generally prefer to name all their parameters. And think about it: why do we need the word “String” in the function – what else would we want to count letters on?

So, what you’ll commonly see is external parameter names like “in”, “for”, and “with”, and more meaningful internal names. So, the “Swifty” way of writing this function is like so:

```
func countLetters(in string: String) {  
    print("The string \(string) has \(string.characters.count)  
letters.")  
}
```

That means you call the function with the parameter name “in”, which would be meaningless inside the function. However, *inside* the function the same parameter is called “string”, which is more useful. So, the function can be called like this:

```
countLetters(in: "Hello")
```

And *that* is truly Swifty code: “count letters in hello” reads like natural English, but the code is also clear and concise.

## Return values

Swift functions can return a value by writing `->` then a data type. Once you do this, Swift will ensure that your function will return a value no matter what, so again this is you making a promise about what your code does.

As an example, let's write a function that returns true if an album is one of Taylor Swift's, or false otherwise. This needs to accept one parameter (the name of the album to check) and will return a Boolean. Here's the code:

```
func albumsIsTaylor(name: String) -> Bool {
```

```

if name == "Taylor Swift" { return true }
if name == "Fearless" { return true }
if name == "Speak Now" { return true }
if name == "Red" { return true }
if name == "1989" { return true }

return false
}

```

If you wanted to try your new **switch/case** knowledge, this function is a place where it would work well.

You can now call that by passing the album name in and acting on the result:

```

if albumsIsTaylor(name: "Red") {
    print("That's one of hers!")
} else {
    print("Who made that?!")
}

if albumsIsTaylor(name: "Blue") {
    print("That's one of hers!")
} else {
    print("Who made that?!")
}

```

# Optionals

Swift is a very safe language, by which I mean it works hard to ensure your code never fails in surprising ways.

One of the most common ways that code fails is when it tries to use data that is bad or missing. For example, imagine a function like this:

```
func getHaterStatus() -> String {  
    return "Hate"  
}
```

That function doesn't accept any parameters, and it returns a string: "Hate". But what if today is a particularly sunny day, and those haters don't feel like hating – what then? Well, maybe we want to return nothing: this hater is doing no hating today.

Now, when it comes to a string you might think an empty string is a great way to communicate nothing, and that might be true sometimes. But how about numbers – is 0 an "empty number"? Or -1?

Before you start trying to create imaginary rules for yourself, Swift has a solution: optionals. An optional value is one that might have a value or might not. Most people find optionals hard to understand, and that's OK – I'm going to try explaining it in several ways, so hopefully one will work.

For now, imagine a survey where you ask someone, "On a scale of 1 to 5 how awesome is Taylor Swift?" – what would someone answer if they had never heard of her? 1 would be unfairly slating her, and 5 would be praising her when they had no idea who Taylor Swift was. The solution is optionals: "I don't want to provide a number at all."

When we used `-> String` it means "this will definitely return a string," which means this function *cannot* return no value, and thus can be called safe in the knowledge that you'll always get a value back that you can use as a string. If we wanted to tell Swift that this function might return a value or it might not, we need to use this instead:

```
func getHaterStatus() -> String? {
```

```
    return "Hate"  
}
```

Note the extra question mark: that means “optional string.” Now, in our case we’re still returning “Hate” no matter what, but let’s go ahead and modify that function further: if the weather is sunny, the haters have turned over a new leaf and have given up their life of hating, so we want to return no value. In Swift, this “no value” has a special name: **nil**.

Change the function to this:

```
func getHaterStatus(weather: String) -> String? {  
    if weather == "sunny" {  
        return nil  
    } else {  
        return "Hate"  
    }  
}
```

That accepts one string parameter (the weather) and returns one string (hating status), but that return value might be there or it might not – it’s nil. In this case, it means we might get a string, or we might get nil.

Now for the important stuff: Swift wants your code to be really safe, and trying to use a nil value is a bad idea. It might crash your code, it might screw up your app logic, or it might make your user interface show the wrong thing. As a result, when you declare a value as being optional, Swift will make sure you handle it safely.

Let’s try this now: add these lines of code to your playground:

```
var status: String  
status = getHaterStatus(weather: "rainy")
```

The first line creates a string variable, and the second assigns to it the value from **getHaterStatus()** – and today the weather is rainy, so those haters are hating for sure.

That code will not run, because we said that **status** is of type **String**, which requires a value, but **getHaterStatus()** might not provide one because it returns an optional string. That is, we said there would be *definitely* be a string in **status**, but **getHaterStatus()** might return nothing at all. Swift simply will not let you make this mistake, which is extremely helpful because it effectively stops dead a whole class of common bugs.

To fix the problem, we need to make the **status** variable a **String?**, or just remove the type annotation entirely and let Swift use type inference. The first option looks like this:

```
var status: String?  
status = getHaterStatus(weather: "rainy")
```

And the second like this:

```
var status = getHaterStatus(weather: "rainy")
```

Regardless of which you choose, that value might be there or might not, and by default Swift won't let you use it dangerously. As an example, imagine a function like this:

```
func takeHaterAction(status: String) {  
    if status == "Hate" {  
        print("Hating")  
    }  
}
```

That takes a string and prints a message depending on its contents. This function takes a **String** value, and *not* a **String?** value – you can't pass in an optional here, it wants a real string, which means we can't call it using the **status** variable.

Swift has two solutions. Both are used, but one is definitely preferred over the other. The first solution is called optional unwrapping, and it's done inside a conditional statement using special syntax. It does two things at the same time: checks whether an optional has a value, and if so unwraps it into a non-optional type then runs a code block.

The syntax looks like this:

```

if let unwrappedStatus = status {
    // unwrappedStatus contains a non-optional value!
} else {
    // in case you want an else block, here you go...
}

```

These **if let** statements check and unwrap in one succinct line of code, which makes them very common. Using this method, we can safely unwrap the return value of **getHaterStatus()** and be sure that we only call **takeHaterAction()** with a valid, non-optional string. Here's the complete code:

```

func getHaterStatus(weather: String) -> String? {
    if weather == "sunny" {
        return nil
    } else {
        return "Hate"
    }
}

func takeHaterAction(status: String) {
    if status == "Hate" {
        print("Hating")
    }
}

if let haterStatus = getHaterStatus(weather: "rainy") {
    takeHaterAction(status: haterStatus)
}

```

If you understand this concept, you're welcome to skip down to the title that says "Force unwrapping optionals". If you're still not quite sure about optionals, carry on reading.

OK, if you're still here it means the explanation above either made no sense, or you sort of understood but could probably use some clarification. Optionals are used extensively in Swift,

so you really do need to understand them. I'm going to try explaining again, in a different way, and hopefully that will help!

Here's a new function:

```
func yearAlbumReleased(name: String) -> Int {  
    if name == "Taylor Swift" { return 2006 }  
    if name == "Fearless" { return 2008 }  
    if name == "Speak Now" { return 2010 }  
    if name == "Red" { return 2012 }  
    if name == "1989" { return 2014 }  
  
    return 0  
}
```

That takes the name of a Taylor Swift album, and returns the year it was released. But if we call it with the album name "Lantern" because we mixed up Taylor Swift with Hudson Mohawke (an easy mistake to make, right?) then it returns 0 because it's not one of Taylor's albums.

But does 0 make sense here? Sure, if the album was released back in 0 AD when Caesar Augustus was emperor of Rome, 0 might make sense, but here it's just confusing – people need to know that 0 means "not recognized."

A much better idea is to re-write that function so that it either returns an integer (when a year was found) or nil (when nothing was found), which is easy thanks to optionals. Here's the new function:

```
func yearAlbumReleased(name: String) -> Int? {  
    if name == "Taylor Swift" { return 2006 }  
    if name == "Fearless" { return 2008 }  
    if name == "Speak Now" { return 2010 }  
    if name == "Red" { return 2012 }  
    if name == "1989" { return 2014 }
```

```
    return nil  
}
```

Now that it returns nil, we need to unwrap the result using **if** **let** because we need to check whether a value exists or not.

**If you understand the concept now, you're welcome to skip down to the title that says “Force unwrapping optionals”.** If you're still not quite sure about optionals, carry on reading.

OK, if you're still here it means you're really struggling with optionals, so I'm going to have one last go at explaining them.

Here's an array of names:

```
var items = ["James", "John", "Sally"]
```

If we wanted to write a function that looked in that array and told us the index of a particular name, we might write something like this:

```
func position(of string: String, in array: [String]) -> Int {  
    for i in 0 ..< array.count {  
        if array[i] == string {  
            return i  
        }  
    }  
  
    return 0  
}
```

That loops through all the items in the array, returning its position if it finds a match, otherwise returning 0.

Now try running these three lines of code:

```
let jamesPosition = position(of: "James", in: items)  
let johnPosition = position(of: "John", in: items)
```

```
let sallyPosition = position(of: "Sally", in: items)
let bobPosition = position(of: "Bob", in: items)
```

That will output 0, 1, 2, 0 – the positions of James and Bob are the same, even though one exists and one doesn't. This is because I used 0 to mean "not found." The easy fix might be to make -1 not found, but whether it's 0 or -1 you still have a problem because you have to remember that specific number means "not found."

The solution is optionals: return an integer if you found the match, or nil otherwise. In fact, this is exactly the approach the built-in "find in array" methods use: **someArray.index(of: someValue)**.

When you work with these "might be there, might not be" values, Swift forces you to unwrap them before using them, thus acknowledging that there might not be a value. That's what **if let** syntax does: if the optional has a value then unwrap it and use it, otherwise don't use it at all. You can't use a possibly-empty value by accident, because Swift won't let you.

If you're *still* not sure how optionals work, then the best thing to do is ask me on Twitter and I'll try to help: you can find me [@twostraws](#).

## Force unwrapping optionals

Swift lets you override its safety by using the exclamation mark character: **!**. If you know that an optional definitely has a value, you can force unwrap it by placing this exclamation mark after it. Please be careful, though: if you try this on a variable that does not have a value, your code will crash.

To put together a working example, here's some foundation code:

```
func yearAlbumReleased(name: String) -> Int? {
    if name == "Taylor Swift" { return 2006 }
    if name == "Fearless" { return 2008 }
    if name == "Speak Now" { return 2010 }
    if name == "Red" { return 2012 }
    if name == "1989" { return 2014 }
```

```

    return nil
}

var year = yearAlbumReleased(name: "Red")

if year == nil {
    print("There was an error")
} else {
    print("It was released in \(year)")
}

```

That gets the year an album was released. If the album couldn't be found, `year` will be set to `nil`, and an error message will be printed. Otherwise, the year will be printed.

Or will it? Well, `yearAlbumReleased()` returns an optional integer, and this code doesn't use `if let` to unwrap that optional. As a result, it will print out the following: "It was released in Optional(2012)" – probably not what we wanted!

At this point in the code, we have already checked that we have a valid value, so it's a bit pointless to have another `if let` in there to safely unwrap the optional. So, Swift provides a solution – change the second `print()` call to this:

```
print("It was released in \(year!)")
```

Note the exclamation mark: it means "I'm certain this contains a value, so force unwrap it now."

## Implicitly unwrapped optionals

You can also use this exclamation mark syntax to create implicitly unwrapped optionals, which is where some people really start to get confused. So, please read this carefully!

- A regular variable must contain a value. Example: `String` must contain a string, even if that string is empty, i.e. `""`. It *cannot* be `nil`.

- An *optional* variable might contain a value, or might not. It must be unwrapped before it is used. Example: **String?** might contain a string, or it might contain nil. The only way to find out is to unwrap it.
- An implicitly unwrapped optional might contain a value, or might not. But it does *not* need to be unwrapped before it is used. Swift won't check for you, so you need to be extra careful. Example: **String!** might contain a string, or it might contain nil – and it's down to you to use it appropriately. It's like a regular optional, but Swift lets you access the value directly without the unwrapping safety. If you try to do it, it means you know there's a value there – but if you're wrong your app will crash.

There are two main times you're going to meet implicitly unwrapped optionals. The first is when you're working with Apple's APIs: these frequently return implicitly unwrapped optionals because their code pre-dates Swift and that was how things were done in Ye Olde Dayes Of Programminge.

The second is when you're working with user interface elements in UIKit on iOS or AppKit on macOS. These need to be declared up front, but you can't use them until they have been created – and Apple likes to create user interface elements at the last possible moment to avoid any unnecessary work. Having to continually unwrap values you definitely know will be there is annoying, so these are made implicitly unwrapped.

Don't worry if you find implicitly unwrapped optionals a bit hard to grasp - it will become clear as you work with the language.

## Optional chaining

Working with optionals can feel a bit clumsy sometimes, and all the unwrapping and checking can become so onerous that you might be tempted to throw some exclamation marks to force unwrap stuff so you can get on with work. Be careful, though: if you force unwrap an optional that doesn't have a value, your code will crash.

Swift has two techniques to help make your code less complicated. The first is called optional chaining, which lets you run code only if your optional has a value. Put the below code into your playground to get us started:

```
func albumReleased(year: Int) -> String? {
    switch year {
        case 2006: return "Taylor Swift"
        case 2008: return "Fearless"
        case 2010: return "Speak Now"
        case 2012: return "Red"
        case 2014: return "1989"
        default: return nil
    }
}

let album = albumReleased(year: 2006)
print("The album is \(album)")
```

That will output "The album is Optional("Taylor Swift")" into the results pane.

If we wanted to convert the return value of `albumReleased()` to be uppercase letters (that is, "TAYLOR SWIFT" rather than "Taylor Swift") we could call the `uppercased()` method of that string. For example:

```
let str = "Hello world"
print(str.uppercased())
```

The problem is, `albumReleased()` returns an optional string: it might return a string or it might return nothing at all. So, what we really mean is, "if we got a string back make it

uppercase, otherwise do nothing." And that's where optional chaining comes in, because it provides exactly that behavior.

Try changing the last two lines of code to this:

```
let album = albumReleased(year: 2006)?.uppercased()  
print("The album is \(album)")
```

Note that there's a question mark in there, which is the optional chaining: everything after the question mark will only be run if everything before the question mark has a value. This doesn't affect the underlying data type of **album**, because that line of code will now either return nil or will return the uppercase album name – it's still an optional string.

Your optional chains can be as long as you need, for example:

```
let album = albumReleased(year:  
2006)?.someOptionalValue?.someOtherOptionalValue?.whatever
```

Swift will check them from left to right until it finds nil, at which point it stops.

## The nil coalescing operator

This simple Swift feature makes your code much simpler and safer, and yet has such a grandiose name that many people are scared of it. This is a shame, because the nil coalescing operator will make your life easier if you take the time to figure it out!

What it does is let you say "use value A if you can, but if value A is nil then use value B." That's it. It's particularly helpful with optionals, because it effectively stops them from being optional because you provide a non-optional value B. So, if A is optional and has a value, it gets used (we have a value.) If A is present and has no value, B gets used (so we still have a value). Either way, we definitely have a value.

To give you a real context, try using this code in your playground:

```
let album = albumReleased(year: 2006) ?? "unknown"  
print("The album is \(album)")
```

That double question mark is the nil coalescing operator, and in this situation it means "if `albumReleased()` returned a value then put it into the `album` variable, but if `albumReleased()` returned nil then use 'unknown' instead."

If you look in the results pane now, you'll see "The album is Taylor Swift" printed in there – no more optionals. This is because Swift can now be sure it will get a real value back, either because the function returned one or because you're providing "unknown". This in turn means you don't need to unwrap anything or risk crashes – you're guaranteed to have real data to work with, which makes your code safer and easier to work with.

# Enumerations

Enumerations – usually just called "enum" and pronounced "ee-num" - are a way for you to define your own kind of value in Swift. In some programming languages they are simple little things, but Swift adds a huge amount of power to them if you want to go beyond the basics.

Let's start with a simple example from earlier:

```
func getHaterStatus(weather: String) -> String? {
    if weather == "sunny" {
        return nil
    } else {
        return "Hate"
    }
}
```

That function accepts a string that defines the current weather. The problem is, a string is a poor choice for that kind of data – is it "rain", "rainy" or "raining"? Or perhaps "showering", "drizzly" or "stormy"? Worse, what if one person writes "Rain" with an uppercase R and someone else writes "Ran" because they weren't looking at what they typed?

Enums solve this problem by letting you define a new data type, then define the possible values it can hold. For example, we might say there are five kinds of weather: sun, cloud, rain, wind and snow. If we make this an enum, it means Swift will accept only those five values – anything else will trigger an error. And behind the scenes enums are usually just simple numbers, which are a lot faster than strings for computers to work with.

Let's put that into code:

```
enum WeatherType {
    case sun, cloud, rain, wind, snow
}

func getHaterStatus(weather: WeatherType) -> String? {
    if weather == WeatherType.sun {
        return nil
    }
}
```

```

    } else {
        return "Hate"
    }
}

getHaterStatus(weather: WeatherType.cloud)

```

Take a look at the first three lines: line 1 gives our type a name, **WeatherType**. This is what you'll use in place of **String** or **Int** in your code. Line 2 defines the five possible cases our enum can be, as I already outlined. Convention has these start with a lowercase letter, so “sun”, “cloud”, etc. And line 3 is just a closing brace, ending the enum.

Now take a look at how it's used: I modified the **getHaterStatus()** so that it takes a **WeatherType** value. The conditional statement is also rewritten to compare against **WeatherType.sun**, which is our value. Remember, this check is just a number behind the scenes, which is lightning fast.

Now, go back and read that code again, because I'm about to rewrite it with two changes that are important. All set?

```

enum WeatherType {
    case sun
    case cloud
    case rain
    case wind
    case snow
}

func getHaterStatus(weather: WeatherType) -> String? {
    if weather == .sun {
        return nil
    } else {
        return "Hate"
    }
}

```

```
getHaterStatus(weather: .cloud)
```

I made two differences there. First, each of the weather types are now on their own line. This might seem like a small change, and indeed in this example it is, but it becomes important soon. The second change was that I wrote `if weather == .sun` – I didn't need to spell out that I meant `WeatherType.sun` because Swift knows I am comparing against a `WeatherType` variable, so it's using type inference.

Note that at this time Xcode is unable to use code completion to suggest enums if you use this short form. If you type them in full, e.g. `WeatherType.sun`, you will get code completion.

Enums are particularly useful inside `switch/case` blocks, particularly because Swift knows all the values your enum can have so it can ensure you cover them all. For example, we might try to rewrite the `getHaterStatus()` method to this:

```
func getHaterStatus(weather: WeatherType) -> String? {
    switch weather {
        case .sun:
            return nil
        case .cloud, .wind:
            return "dislike"
        case .rain:
            return "hate"
    }
}
```

Yes, I realize "haters gonna dislike" is hardly a great lyric, but it's academic anyway because this code won't build: it doesn't handle the `.snow` case, and Swift wants all cases to be covered. You either have to add a case for it or add a default case.

## Enums with additional values

One of the most powerful features of Swift is that enumerations can have values attached to them that you define. To extend our increasingly dubious example a bit further, I'm going to

add a value to the `.wind` case so that we can say how fast the wind is. Modify your code to this:

```
enum WeatherType {
    case sun
    case cloud
    case rain
    case wind(speed: Int)
    case snow
}
```

As you can see, the other cases don't need a speed value – I put that just into wind. Now for the real magic: Swift lets us add extra conditions to the `switch/case` block so that a case will match only if those conditions are true. This uses the `let` keyword to access the value inside a case, then the `where` keyword for pattern matching.

Here's the new function:

```
func getHaterStatus(weather: WeatherType) -> String? {
    switch weather {
        case .sun:
            return nil
        case .wind(let speed) where speed < 10:
            return "meh"
        case .cloud, .wind:
            return "dislike"
        case .rain, .snow:
            return "hate"
    }
}

getHaterStatus(weather: WeatherType.wind(speed: 5))
```

You can see `.wind` appears in there twice, but the first time is true only if the wind is slower than 10 kilometers per hour. If the wind is 10 or above, that won't match. The key is that you

use **let** to get hold of the value inside the enum (i.e. to declare a constant name you can reference) then use a **where** condition to check.

Swift evaluates **switch/case** from top to bottom, and stops as soon as it finds match. This means that if **case .cloud, .wind:** appears before **case .wind(let speed)  
where speed < 10:** then it will be executed instead – and the output changes. So, think carefully about how you order cases!

## Structs

Structs are complex data types, meaning that they are made up of multiple values. You then create an instance of the struct and fill in its values, then you can pass it around as a single value in your code. For example, we could define a **Person** struct type that contains two properties: **clothes** and **shoes**:

```
struct Person {  
    var clothes: String  
    var shoes: String  
}
```

When you define a struct, Swift makes them very easy to create because it automatically generates what's called a memberwise initializer. In plain speak, it means you create the struct by passing in initial values for its two properties, like this:

```
let taylor = Person(clothes: "T-shirts", shoes: "sneakers")  
let other = Person(clothes: "short skirts", shoes: "high  
heels")
```

Once you have created an instance of a struct, you can read its properties just by writing the name of the struct, a period, then the property you want to read:

```
print(taylor.clothes)  
print(other.shoes)
```

If you assign one struct to another, Swift copies it behind the scenes so that it is a complete, standalone duplicate of the original. Well, that's not strictly true: Swift uses a technique called "copy on write" which means it only actually copies your data if you try to change it.

To help you see how struct copies work, put this into your playground:

```
struct Person {  
    var clothes: String  
    var shoes: String  
}
```

```
let taylor = Person(clothes: "T-shirts", shoes: "sneakers")
let other = Person(clothes: "short skirts", shoes: "high
heels")

var taylorCopy = taylor
taylorCopy.shoes = "flip flops"

print(taylor)
print(taylorCopy)
```

That creates two **Person** structs, then creates a third one called **taylorCopy** as a copy of **taylor**. What happens next is the interesting part: the code changes **taylorCopy**, and prints both it and **taylor**. If you look in your results pane (you might need to resize it to fit) you'll see that the copy has a different value to the original: changing one did not change the other.

# Classes

Swift has another way of build complex data types called classes. They look similar to structs, but have a number of important differences, including:

- You don't get an automatic memberwise initializer for your classes; you need to write your own.
- You can define a class as being based off another class, adding any new things you want.
- When you create an instance of a class it's called an object. If you copy that object, both copies point at the same data by default – change one, and the copy changes too.

All three of those are massive differences, so I'm going to cover them in more depth before continuing.

## Initializing an object

If we were to convert our **Person** struct into a **Person** class, Swift wouldn't let us write this:

```
class Person {  
    var clothes: String  
    var shoes: String  
}
```

This is because we're declaring the two properties to be **String**, which if you remember means they absolutely must have a value. This was fine in a struct because Swift automatically produces a memberwise initializer for us that forced us to provide values for the two properties, but this doesn't happen with classes so Swift can't be sure they will be given values.

There are three solutions: make the two values optional strings, give them default values, or write our own initializer. The first option is clumsy because it introduces optionals all over our code where they don't need to be. The second option works, but it's a bit wasteful unless those default values will actually be used. That leaves the third option, and really it's the right one: write our own initializer.

To do this, create a function inside the class called **init()** that takes the two parameters we

care about:

```
class Person {  
    var clothes: String  
    var shoes: String  
  
    init(clothes: String, shoes: String) {  
        self.clothes = clothes  
        self.shoes = shoes  
    }  
}
```

There are two things that might jump out at you in that code. First, you don't write **func** before your **init()** function, because it's special. Second, because the parameter names being passed in are the same as the names of the properties we want to assign, you use **self**. to make your meaning clear – "the **clothes** property of this object should be set to the **clothes** parameter that was passed in." You can give them unique names if you want – it's down to you.

There are two more things you ought to know but can't see in that code. First, when you write a function inside a class, it's called a *method* instead. In Swift you write **func** whether it's a function or a method, but the distinction is preserved when you talk about them.

Second, Swift requires that all non-optional properties have a value by the end of the initializer, or by the time the initializer calls any other method – whichever comes first.

## Class inheritance

The second difference between classes and structs are that classes can build on each other to produce greater things, known as *class inheritance*. This is a technique used extensively in Cocoa Touch, even in the most basic programs, so it's something you should get to grips with.

Let's start with something simple: a **Singer** class that has properties, which is their name and age. As for methods, there will be a simple initializer to handle setting the properties, plus a **sing()** method that outputs some words:

```

class Singer {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }

    func sing() {
        print("La la la la")
    }
}

```

We can now create an instance of that object by calling that initializer, then read out its properties and call its method:

```

var taylor = Singer(name: "Taylor", age: 25)
taylor.name
taylor.age
taylor.sing()

```

That's our basic class, but we're going to build on it: I want to define a **CountrySinger** class that has everything the **Singer** class does, but when I call **sing()** on it I want to print "Trucks, guitars, and liquor" instead.

You could of course just copy and paste the original **Singer** into a new class called **CountrySinger** but that's a lazy way to program and it will come back to haunt if you make later changes to **Singer** and forget to copy them across. Instead, Swift has a smarter solution: we can define **CountrySinger** as being based off **Singer** and it will get all its properties and methods for us to build on:

```

class CountrySinger: Singer {
}

```

That colon is what does the magic: it means "**CountrySinger** extends **Singer**." Now, that new **CountrySinger** class (called a subclass) doesn't add anything to **Singer** (called the parent class, or superclass) yet. We want it to have its own **sing()** method, but in Swift you need to learn a new keyword: **override**. This means "I know this method was implemented by my parent class, but I want to change it for this subclass."

Having the **override** keyword is helpful, because it makes your intent clear. It also allows Swift to check your code: if you don't use **override** Swift won't let you change a method you got from your superclass, or if you use **override** and there wasn't anything to override, Swift will point out your error.

So, we need to use **override func**, like this:

```
class CountrySinger : Singer {  
    override func sing() {  
        print("Trucks, guitars, and liquor")  
    }  
}
```

Now modify the way the **taylor** object is created:

```
var taylor = CountrySinger(name: "Taylor", age: 25)  
taylor.sing()
```

If you change **CountrySinger** to just **Singer** you should be able to see the different messages appearing in the results pane.

Now, to make things more complicated, we're going to define a new class called **HeavyMetalSinger**. But this time we're going to store a new property called **noiseLevel** defining how loud this particular heavy metal singer likes to scream down their microphone.

This causes a problem, and it's one that needs to be solved in a very particular way:

- Swift wants all non-optional properties to have a value.

- Our **Singer** class doesn't have a **noiseLevel** property.
- So, we need to create a custom initializer for **HeavyMetalSinger** that accepts a noise level.
- That new initializer also needs to know the **name** and **age** of the heavy metal singer, so it can pass it onto the superclass **Singer**.
- Passing on data to the superclass is done through a method call, and you can't make method calls in initializers until you have given all your properties values.
- So, we need to set our own property first (**noiseLevel**) then pass on the other parameters for the superclass to use.

That might sound awfully complicated, but in code it's straightforward. Here's the **HeavyMetalSinger** class, complete with its own **sing()** method:

```
class HeavyMetalSinger : Singer {
    var noiseLevel: Int

    init(name: String, age: Int, noiseLevel: Int) {
        self.noiseLevel = noiseLevel
        super.init(name: name, age: age)
    }

    override func sing() {
        print("Grrrrr rargh rargh rarrrrgh!")
    }
}
```

Notice how its initializer takes three parameters, then calls **super.init()** to pass **name** and **age** on to the **Singer** superclass - but only after its own property has been set. You'll see **super** used a lot when working with objects, and it just means "call a method on the class I inherited from. It's usually used to mean "let my parent class do everything it needs to do first, then I'll do my extra bits."

Class inheritance is a big topic so don't fret if it's not clear just yet. However, there is one more thing you need to know: class inheritance often spans many levels. For example, A could

inherit from B, and B could inherit from C, and C could inherit from D, and so on. This lets you build functionality and re-use up over a number of classes, helping to keep your code modular and easy to understand.

## Values vs References

When you copy a struct, the whole thing is duplicated, including all its values. This means that changing one copy of a struct doesn't change the other copies – they are all individual. With classes, each copy of an object points at the same original object, so if you change one they all change. Swift calls structs "value types" because they just point at a value, and classes "reference types" because objects are just shared references to the real value.

This is an important difference, and it means the choice between structs and classes is an important one:

- If you want to have one shared state that gets passed around and modified in place, you're looking for classes. You can pass them into functions or store them in arrays, modify them in there, and have that change reflected in the rest of your program.
- If you want to avoid shared state where one copy can't affect all the others, you're looking for structs. You can pass them into functions or store them in arrays, modify them in there, and they won't change wherever else they are referenced.

If I were to summarize this key difference between structs and classes, I'd say this: classes offer more flexibility, whereas structs offer more safety. As a basic rule, you should always use structs until you have a specific reason to use classes.

# Properties

Structs and classes (collectively: "types") can have their own variables and constants, and these are called properties. These let you attach values to your types to represent them uniquely, but because types can also have methods you can have them behave according to their own data.

Let's take a look at an example now:

```
struct Person {
    var clothes: String
    var shoes: String

    func describe() {
        print("I like wearing \(clothes) with \(shoes)")
    }
}

let taylor = Person(clothes: "T-shirts", shoes: "sneakers")
let other = Person(clothes: "short skirts", shoes: "high
heels")
taylor.describe()
other.describe()
```

As you can see, when you use a property inside a method it will automatically use the value that belongs to the same object.

## Property observers

Swift lets you add code to be run when a property is about to be changed or has been changed. This is frequently a good way to have a user interface update when a value changes, for example.

There are two kinds of property observer: **willSet** and  **didSet**, and they are called before or after a property is changed. In **willSet** Swift provides your code with a special value called **newValue** that contains what the new property value is going to be, and in  **didSet** you are given  **oldValue** to represent the previous value.

Let's attach two property observers to the **clothes** property of a **Person** struct:

```
struct Person {
    var clothes: String {
        willSet {
            updateUI(msg: "I'm changing from \(clothes) to \(newValue)")
        }

        didSet {
            updateUI(msg: "I just changed from \(oldValue) to \(clothes)")
        }
    }
}

func updateUI(msg: String) {
    print(msg)
}

var taylor = Person(clothes: "T-shirts")
taylor.clothes = "short skirts"
```

That will print out the messages "I'm changing from T-shirts to short skirts" and "I just changed from T-shirts to short skirts."

## Computed properties

It's possible to make properties that are actually code behind the scenes. We already used the **uppercase()** method of strings, for example, but there's also a property called **capitalized** that gets calculated as needed, rather than every string always storing a capitalized version of itself.

To make a computed property, place an open brace after your property then use either **get** or

**set** to make an action happen at the appropriate time. For example, if we wanted to add a **ageInDogYears** property that automatically returned a person's age multiplied by seven, we'd do this:

```
struct Person {  
    var age: Int  
  
    var ageInDogYears: Int {  
        get {  
            return age * 7  
        }  
    }  
}  
  
var fan = Person(age: 25)  
print(fan.ageInDogYears)
```

Computed properties are increasingly common in Apple's code, but less common in user code.

## Static properties and methods

Swift lets you create properties and methods that belong to a type, rather than to instances of a type. This is helpful for organizing your data meaningfully by storing shared data.

Swift calls these shared properties "static properties", and you create one just by using the **static** keyword. Once that's done, you access the property by using the full name of the type. Here's a simple example:

```
struct TaylorFan {  
    static var favoriteSong = "Shake it Off"  
  
    var name: String  
    var age: Int  
}  
  
let fan = TaylorFan(name: "James", age: 25)  
print(TaylorFan.favoriteSong)
```

So, a Taylor Swift fan has a name and age that belongs to them, but they all have the same favorite song.

Because static methods belong to the class rather than to instances of a class, you can't use it to access any non-static properties from the class.

## Access control

Access control lets you specify what data inside structs and classes should be exposed to the outside world, and you get to choose three modifiers:

- Public: this means everyone can read and write the property.
- Internal: this means only your Swift code can read and write the property. If you ship your code as a framework for others to use, they won't be able to read the property.
- File Private: this means that only Swift code in the same file as the type can read and write the property.
- Private: this is the most restrictive option, and means the property is available only inside methods that belong to the type.

Most of the time you don't need to specify access control, but sometimes you'll want to explicitly set a property to be private because it stops others from accessing it directly. This is useful because your own methods can work with that property, but others can't, thus forcing them to go through your code to perform certain actions.

To declare a property private, just do this:

```
class TaylorFan {  
    private var name: String!  
}
```

If you want to use “file private” access control, just write it as one word like so:

**fileprivate**.

# Polymorphism and typecasting

Because classes can inherit from each other (e.g. **CountrySinger** can inherit from **Singer**) it means one class is effectively a superset of another: class B has all the things A has, with a few extras. This in turn means that you can treat B as type B or as type A, depending on your needs.

Confused? Let's try some code:

```
class Album {
    var name: String

    init(name: String) {
        self.name = name
    }
}

class StudioAlbum: Album {
    var studio: String

    init(name: String, studio: String) {
        self.studio = studio
        super.init(name: name)
    }
}

class LiveAlbum: Album {
    var location: String

    init(name: String, location: String) {
        self.location = location
        super.init(name: name)
    }
}
```

That defines three classes: albums, studio albums and live albums, with the latter two both inheriting from **Album**. Because any instance of **LiveAlbum** is inherited from **Album** it can be treated just as either **Album** or **LiveAlbum** – it's both at the same time. This is called "polymorphism," but it means you can write code like this:

```
var taylorSwift = StudioAlbum(name: "Taylor Swift", studio:  
    "The Castles Studios")  
var fearless = StudioAlbum(name: "Speak Now", studio:  
    "Aimeeland Studio")  
var iTunesLive = LiveAlbum(name: "iTunes Live from SoHo",  
    location: "New York")  
  
var allAlbums: [Album] = [taylorSwift, fearless, iTunesLive]
```

There we create an array that holds only albums, but put inside it two studio albums and a live album. This is perfectly fine in Swift because they are all descended from the **Album** class, so they share the same basic behavior.

We can push this a step further to really demonstrate how polymorphism works. Let's add a **getPerformance()** method to all three classes:

```
class Album {  
    var name: String  
  
    init(name: String) {  
        self.name = name  
    }  
  
    func getPerformance() -> String {  
        return "The album \(name) sold lots"  
    }  
}  
  
class StudioAlbum: Album {  
    var studio: String
```

```

    init(name: String, studio: String) {
        self.studio = studio
        super.init(name: name)
    }

    override func getPerformance() -> String {
        return "The studio album \(name) sold lots"
    }
}

class LiveAlbum: Album {
    var location: String

    init(name: String, location: String) {
        self.location = location
        super.init(name: name)
    }

    override func getPerformance() -> String {
        return "The live album \(name) sold lots"
    }
}

```

The **getPerformance()** method exists in the **Album** class, but both child classes override it. When we create an array that holds **Albums**, we're actually filling it with subclasses of albums: **LiveAlbum** and **StudioAlbum**. They go into the array just fine because they inherit from the **Album** class, but they never lose their original class. So, we could write code like this:

```

var taylorSwift = StudioAlbum(name: "Taylor Swift", studio:
    "The Castles Studios")
var fearless = StudioAlbum(name: "Speak Now", studio:
    "Aimeeland Studio")

```

```

var iTunesLive = LiveAlbum(name: "iTunes Live from SoHo",
location: "New York")

var allAlbums: [Album] = [taylorSwift, fearless, iTunesLive]

for album in allAlbums {
    print(album.getPerformance())
}

```

That will automatically use the override version of `getPerformance()` depending on the subclass in question. That's polymorphism in action: an object can work as its class and its parent classes, all at the same time.

## Converting types with typecasting

You will often find you have an object of a certain type, but really you know it's a different type. Sadly, if Swift doesn't know what you know, it won't build your code. So, there's a solution, and it's called typecasting: converting an object of one type to another.

Chances are you're struggling to think why this might be necessary, but I can give you a very simple example:

```

for album in allAlbums {
    print(album.getPerformance())
}

```

That was our loop from a few minutes ago. The `allAlbums` array holds the type `Album`, but we know that really it's holding one of the subclasses: `StudioAlbum` or `LiveAlbum`. Swift doesn't know that, so if you try to write something like `print(album.studio)` it will refuse to build because only `StudioAlbum` objects have that property.

Typecasting in Swift comes in three forms, but most of the time you'll only meet two: `as?` and `as!`, known as optional downcasting and forced downcasting. The former means "I think this conversion might be true, but it might fail," and the second means "I know this conversion is true, and I'm happy for my app to crash if I'm wrong." When I say "conversion" I don't mean

that the object literally gets transformed. Instead, it's just converting how Swift treats the object – you're telling Swift that an object it thought was type A is actually type E.

The question and exclamation marks should give you a hint of what's going on, because this is very similar to optional territory. For example, if you write this:

```
for album in allAlbums {  
    let studioAlbum = album as? StudioAlbum  
}
```

Swift will make **studioAlbum** have the data type **StudioAlbum?**. That is, an optional studio album: the conversion might have worked, in which case you have a studio album you can work with, or it might have failed, in which case you have nil. This is most commonly used with **if let** to automatically unwrap the optional result, like this:

```
for album in allAlbums {  
    print(album.getPerformance())  
  
    if let studioAlbum = album as? StudioAlbum {  
        print(studioAlbum.studio)  
    } else if let liveAlbum = album as? LiveAlbum {  
        print(liveAlbum.location)  
    }  
}
```

That will go through every album and print its name, because that's common to the **Album** class and all its subclasses. It then checks whether it can convert the **album** value into a **StudioAlbum**, and if it can it prints out the studio name. The same thing is done for the **LiveAlbum** in the array.

Forced downcasting is when you're really sure an object of one type can be treated like a different type, but if you're wrong your program will just crash. Forced downcasting doesn't need to return an optional value, because you're saying the conversion is definitely going to work – if you're wrong, it means you wrote your code wrong.

To demonstrate this in a non-crashy way, let's strip out the live album so that we just have studio albums in the array:

```
var taylorSwift = StudioAlbum(name: "Taylor Swift", studio:  
    "The Castles Studios")  
var fearless = StudioAlbum(name: "Speak Now", studio:  
    "Aimeeland Studio")  
  
var allAlbums: [Album] = [taylorSwift, fearless]  
  
for album in allAlbums {  
    let studioAlbum = album as! StudioAlbum  
    print(studioAlbum.studio)  
}
```

That's obviously a contrived example, because if that really were your code you would just change `allAlbums` so that it had the data type `[StudioAlbum]`. Still, it shows how forced downcasting works, and the example won't crash because it makes the correct assumptions.

Swift lets you downcast as part of the array loop, which in this case would be more efficient. If you wanted to write that forced downcast at the array level, you would write this:

```
for album in allAlbums as! [StudioAlbum] {  
    print(album.studio)  
}
```

That no longer needs to downcast every item inside the loop, because it happens when the loop begins. Again, you had better be correct that all items in the array are `StudioAlbums`, otherwise your code will crash.

Swift also allows optional downcasting at the array level, although it's a bit more tricksy because you need to use the nil coalescing operator to ensure there's always a value for the loop. Here's an example:

```
for album in allAlbums as? [LiveAlbum] ?? [LiveAlbum]() {
```

```
    print(album.location)
}
```

What that means is, “try to convert **allAlbums** to be an array of **LiveAlbum** objects, but if that fails just create an empty array of live albums and use that instead” – i.e., do nothing. It’s possible to use this, but I’m not sure you’d really want to!

## Converting common types with initializers

Typecasting is useful when you know something that Swift doesn’t, for example when you have an object of type **A** that Swift thinks is actually type **B**. However, typecasting is useful only when those types really are what you say – you can’t force a type **A** into a type **Z** if they aren’t actually related.

For example, if you have an integer called **number**, you couldn’t write code like this to make it a string:

```
let number = 5
let text = number as! String
```

That is, you can’t force an integer into a string, because they are two completely different types. Instead, you need to create a new string by feeding it the integer, and Swift knows how to convert the two. The difference is subtle: this is a *new* value, rather than just a re-interpretation of the same value.

So, that code should be rewritten like this:

```
let number = 5
let text = String(number)
print(text)
```

This only works for some of Swift’s built-in data types: you can convert integers and floats to strings and back again, for example, but if you created two custom structs Swift can’t magically convert one to the other – you need to write that code yourself.

# Closures

You've met integers, strings, doubles, floats, Booleans, arrays, dictionaries, structs and classes so far, but there's another type of data that is used extensively in Swift, and it's called a closure. These are complicated, but they are so powerful and expressive that they are used pervasively in Cocoa Touch, so you won't get very far without understanding them.

A closure can be thought of as a variable that holds code. So, where an integer holds 0 or 500, a closure holds lines of Swift code. It's different to a function, though, because closures are a data type in their own right: you can pass a closure as a parameter or store it as a property. Closures also capture the environment where they are created, which means they take a copy of the values that are used inside them.

You never *need* to design your own closures so don't be afraid if you find the following quite complicated. However, both Cocoa and Cocoa Touch will often ask you to write closures to match their needs, so you at least need to know how they work. Let's take a Cocoa Touch example first:

```
let vw = UIView()  
  
UIView.animate(withDuration: 0.5, animations: {  
    vw.alpha = 0  
})
```

**UIView** is an iOS data type in UIKit that represents the most basic kind of user interface container. Don't worry about what it does for now, all that matters is that it's the basic user interface component. **UIView** has a method called **animate()** and it lets you change the way your interface looks using animation – you describe what's changing and over how many seconds, and Cocoa Touch does the rest.

The **animate()** method takes two parameters in that code: the number of seconds to animate over, and a closure containing the code to be executed as part of the animation. I've specified half a second as the first parameter, and for the second I've asked UIKit to adjust the view's alpha (that's opacity) to 0, which means "completely transparent."

This method needs to use a closure because UIKit has to do all sorts of work to prepare for the

animation to begin, so what happens is that UIKit takes a copy of the code inside the braces (that's our closure), stores it away, does all its prep work, then runs our code when it's ready. This wouldn't be possible if we just run our code directly.

The above code also shows how closures capture their environment: I declared the `vw` constant outside of the closure, then used it inside. Swift detects this, and makes that data available inside the closure too.

Swift's system of automatically capturing a closure's environment is very helpful, but can occasionally trip you up: if object A stores a closure as a property, and that property also references object A, you have something called a strong reference cycle and you'll have unhappy users. This is a substantially more advanced topic than you need to know right now, so don't worry too much about it just yet.

## Trailing closures

As closures are used so frequently, Swift can apply a little syntactic sugar to make your code easier to read. The rule is this: if the last parameter to a method takes a closure, you can eliminate that parameter and instead provide it as a block of code. For example, we could convert the previous code to this:

```
let vw = UIView()  
  
UIView.animate(withDuration: 0.5) {  
    vw.alpha = 0  
}
```

It does make your code shorter and easier to read, so this syntax form – known as trailing closure syntax – is preferred.

## Wrap up

That's the end of our tour around the Swift programming language. I haven't tried to cover everything in the language, but that's OK because you have all the important stuff, all the sometimes-important stuff, and all the nice-to-know stuff – the many other features you'll either come across in a later project or through extended experience with the language.

From here on, we're going to focus primarily on building apps. If you want to learn more about the Swift language itself, you might want to consider my [Pro Swift](#) book.

# **Chapter 1**

## Million Hairs

## Setting up

The first project of any book is always difficult, because there's a steep difficulty ramp as you learn all the basics required just to make something simple. I've tried to keep it as simple as possible while still teaching useful skills, so in this initial project you're going to create a website for the Million Hairs veterinary clinic.

Along the way you're going to learn how to create and configure a project using the Swift package manager, how to route users to different parts of your code depending on the URL they enter, how to log information, and how to separate your code from your presentation – i.e., how to keep Swift separate from HTML.

Now, it's possible you think I'm already patronizing you: surely you already know how to create a new Swift project? You have to remember that server-side Swift is designed to work across platforms, which means that it relies on Xcode far less. In fact, by default Xcode isn't involved at all: you create your project using the Swift package manager, which comes bundled with the open-source Swift distribution.

If you're using macOS with Xcode 8.1 installed, you already have access to the Swift package manager. If you're using Docker with IBM's Kitura container, then that pre-installs the Swift distribution and gets the package manager that way. If you're using a real Linux install, either in a VM or in the cloud, then you installed the package manager when you installed Swift from swift.org.

Regardless of which approach you've taken, you should be able to run **swift package** on the command line to see some information about the Swift package manager.

We're going to create a new project called "project1", then build it and run it. In Swift package manager terms, that's an executable package – a package that is designed to compile into a single executable.

To create a new project, run these commands:

```
cd Desktop/server  
mkdir project1  
cd project1
```

```
swift package init --type executable
```

If you're already in the Desktop/server directory, skip the first command.

**Docker users:** You can run those commands inside the container or directly on your Mac's terminal – it doesn't matter.

That creates a new “project1” directory and changes into it, then instructs the package manager to create a new project. The **init** parameter tells the package manager we want to create a new project. The **--type executable** parameters mean we want to create a standalone binary, rather than a library or something else.

You'll see the following output:

```
Creating executable package: project1
Creating Package.swift
Creating .gitignore
Creating Sources/
Creating Sources/main.swift
Creating Tests/
```

That's a skeleton package set up: just enough to build a basic Swift app. We'll look at what it does shortly, but first let's check that it all works. Run this command:

```
swift build
```

That instructs the Swift compiler to build everything in the “Sources” directory and turn it into a single executable. The package manager creates “Sources/main.swift” with a single **print()** statement, and **swift build** compiled that into an executable located at “.build/debug/project1”.

Before we move on, let's test the executable now. Run this command:

```
.build/debug/project1
```

If everything has worked, you should see “Hello, world!” printed out in your terminal window.

That wasn't so hard, was it?

## Swift packages explained

The Swift package manager is similar in concept to other package managers such as “npm”, although it’s significantly less developed at this time. Its job is to manage your package, which sounds obvious given that it’s a *package manager*, but it’s important.

Your package is your app, and the Swift package manager is responsible for building it, testing it, and most importantly managing its dependencies – third-party software that your code relies on. These dependencies are specified as remote Git repositories, and usually come with their own set of dependencies – again, all handled by the package manager.

Your package is described entirely inside the file `Package.swift`, which is actually Swift source code. Go ahead and open it in a text editor, and you’ll see it contains the following:

```
import PackageDescription

let package = Package(
    name: "project1"
)
```

Every project in this book will require at least two dependencies: the Kitura framework to do most of our heavy lifting, and the Helium logger, which is a super-lightweight logging framework that lets us write debugging information to the terminal so we can spot problems. For this project we’re going to use a third dependency called Stencil, which lets us generate HTML cleanly.

Like I said, these dependencies are specified as Git repositories, but you also get to attach a version number. This means you can deploy the package elsewhere and be assured that it won’t accidentally use newer dependencies than you had intended. The version number can be declared as just a major version – “I want Kitura v1” – or as a major version plus minor version – “I want Kitura v1.1”. Unless you have very specific needs, it’s best to use just a major version, because that guarantees you will get new features as long as they are fully backwards compatible with your existing code.

The Swift package manager doesn’t have a neat way to add dependencies; you literally need to rewrite `Package.swift`’s source code as needed. We need to add dependencies for Kitura,

Helium, and Stencil, so please modify Package.swift to this:

```
import PackageDescription

let package = Package(
    name: "project1",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
        majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
StencilTemplateEngine.git", majorVersion: 1)
    ]
)
```

Make sure you save the file before continuing, otherwise the following steps will be very confusing indeed!

Now that you've modified your package description, I'd like you to run **swift build** again. This time it will take longer, because the package manager detects your dependencies has changed so it will clone the Git repositories – i.e., take a local copy of them. This cloning only happens the first time a dependency is used; after that, each dependency will be stored in a directory called “Packages” so it won't be downloaded again.

The end result of running **swift build** will be the same executable file we had before, but now the dependencies are installed we can start using them.

Before we move on, let's take a brief look at our folder structure:

- “.build” is a hidden directory because it starts with a full stop / period. It's where Swift places our compiled executable.
- “Packages” is where the package manager stores its downloaded dependencies. If you look in there you'll see more than the three dependencies we requested, because each of those dependencies can have their *own* dependencies.

- “Sources” is where you should place your source code. Right now it contains only main.swift, but we’ll add more there soon enough.
- “Tests” is where you place your XCTest-compatible Swift tests. We’ll be covering this in detail towards the end of the book.
- There’s also a hidden file called “.gitignore”, which configures your source control system to ignore the “.build” and “Packages” directories because they are generated dynamically.

You can open Sources/main.swift in any text editor you like; it’s just loose Swift source code. However, if you’re working on a Mac I want to show you how to create an Xcode project so that you can switch to a more familiar environment.

But first, some warnings:

1. Xcode does not exist on Linux. It’s very helpful to use Xcode and I recommend it where possible, but please remember that other team members may not have access to it.
2. Xcode will happily offer code completion for methods that are unimplemented in open-source Foundation.
3. Although you *can* build and run your projects using Xcode, it can be difficult to watch what’s happening in the console logs. Kitura (and Helium) outputs a lot of useful logging information, and having that nice and big in the terminal is definitely best.
4. The Swift package manager automatically configures your package *not* to save the Xcode project into source control. It’s useful to work with, but it’s something you should generate from the package as needed rather than customizing and saving.
5. If you’re using Docker, you might find that Xcode and Docker fight over port 8090. If you’re using Docker you can use Xcode for editing, but I suggest you continue to run inside Docker.

With that in mind, we’re going to ask the Swift package manager to create an Xcode project from our current package so you can try it out. **If you’re using Docker, you should run this command from the macOS terminal rather than the terminal inside your container to avoid any problems.**

Run this command now:

```
swift package generate-xcodeproj
```

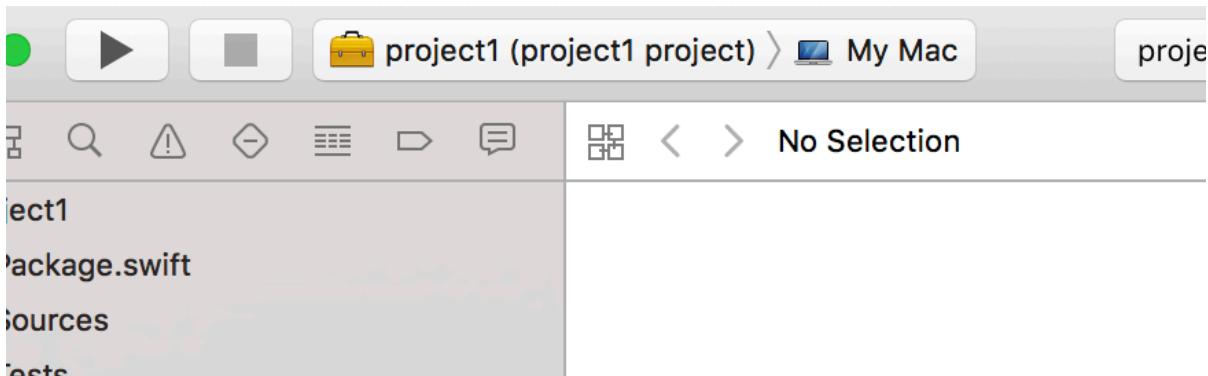
It might take a few seconds to complete, but when it finishes you'll see project1.xcodeproj in your current directory. You can open that in Xcode by running **open project1.xcodeproj** or by double-clicking it in Finder.

Once you're inside Xcode, you'll find main.swift nestled under project1 > Sources > project1. It contains only this so far:

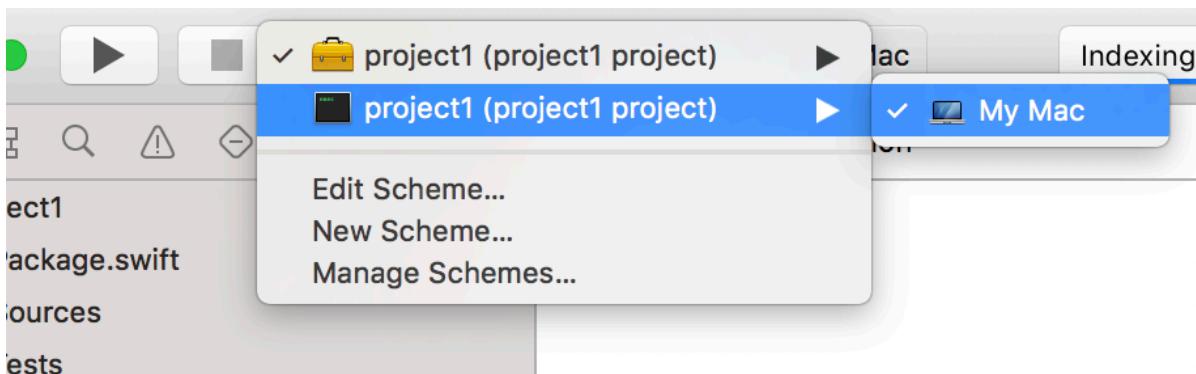
```
print("Hello, world!")
```

Try changing the message to "**Hello, Kitura!**" instead. You'll notice that pressing Cmd +R to run your app does nothing – you'll just hear your Mac beep when you try pressing it. Cmd+B to *build* works, but Cmd+R to run fails because by default Xcode doesn't know what to do.

To fix the problem, look at the active build scheme immediately to the right of the play/stop buttons. You will see "project1 (project1 project)" next to a yellow toolbox icon, then "My Mac", like the screenshot below:



If you leave it with that yellow toolbox selected, Xcode won't build. Instead, you need to click on "project1 (project1 project)", then highlight the option below it and choose "My Mac". Yes, the other option *also* says "project1 (project1 project)", but it has a small black terminal icon next to it.



With that change, Cmd+R will now function as expected: you can build and run your code straight from Xcode.

If you intend solely to develop for macOS, building and running through Xcode is possible – although I do recommend you make the console window nice and big so you can read all the text.

Although Xcode does offer advantages like syntax highlighting and code completion, it *doesn't* automatically detect when you add new files to the Sources folder. Instead, you either need to create them outside Xcode then drag them in by hand, or add them using Xcode and make sure you save them in the Sources folder. Either way, it's down to you to keep Xcode in sync with your package – it's not hard, it just takes careful maintenance.

**Warning:** Right now, Xcode is having trouble with code completion for some Kitura classes. You'll see “error type” appear in various places, which is quite annoying. However, code completion *does* work in your own code, so it's still worth having Xcode around if you have the option.

## Starting a server

When a user enters `yoursite.com` into their web browser, Kitura is responsible for responding to that request with your content. I know that sounds obvious, but it turns out the whole process is complex, so I want to break it down into small components so you can see exactly how Kitura works and, more importantly, *why* it works that way.

Let's start with the absolute basics: Kitura includes a web server that listens on a specific port number. The standard port assigned to HTTP is 80, but many operating systems refuse to let users modify ports numbered below 1024 for safety reasons. As a result, Kitura users work with port 8090 by default – you can use that without an admin password just fine.

As you might imagine, Kitura's web server doesn't know anything about your site structure. You need to tell it what paths you care about, and what Swift code should be attached to each path – a process known as routing. You can specify as many routes as you want, but we're going to start simple and work our way up: we're just going to create a router with no routes, which will cause Kitura to serve its default welcome page.

So, open `Main.swift` in a text editor of your choice (Xcode is fine), and change its contents to this:

```
import Kitura

let router = Router()

Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()
```

That's only four lines of code, but let's walk through them just quickly:

1. All the basic Kitura code is contained inside the “Kitura” framework. There are other components, such as JSON or HTML rendering, that are in separate frameworks, but just using “Kitura” is enough for now.
2. The **Router** class is responsible for matching user requests (“/users/twostraws”) to Swift code that you write. It does a *lot*, but we're starting simple.
3. The **addHTTPServer()** method tells Kitura we want to listen for requests on port

8090, and send any users through to the `router` object we just made.

4. Finally, the `run()` method starts the server. It's a separate call to `addHTTPServer()` because you can listen on multiple ports if you want.

Build and run your code now. If you're using the command line in Docker or directly on your Mac you should run `swift build` followed by `.build/debug/project1`. If you're using Xcode, press the Play button.

**REALLY BIG WARNING:** Some people try to cross the streams by running Docker and running from Xcode at the same time. This is likely to throw up errors that port 8090 is already in use, because Docker likes to grab its ports even though your app isn't running inside there right now. So, if you see errors that port 8090 is in use, you either need to halt Docker or you need to run your code from there.

Our previous code just ran `print("Hello, Kitura!")` then exited. Having imported Kitura then set up and started a web server, the app does... nothing? Well, no, but it certainly *seems* to do nothing: you'll see no output at all, and the app doesn't ever seem to finish.

The app doesn't ever finish because Kitura's `run()` call doesn't return until the server is terminated. That might be because of a crash, because you wrote code to stop the server after a particular event, or because you pressed Ctrl+C on the command line to force termination.

However, Kitura is most definitely up and running – you can see for yourself by opening a web browser and pointing it at the URL <http://localhost:8090>. If everything has worked correctly you should see something like the screenshot below: "Welcome to Kitura".

So, Kitura is definitely working, although it's certainly on the quiet side right now. Behind the scenes, Kitura is outputting log messages whenever interesting things happen, but right now those messages go nowhere. To fix that, we're going to start up Helium Logger in just three lines of code.

First, add these `import` statements below the existing `import Kitura`:

```
import LoggerAPI
import HeliumLogger
```

The LoggerAPI framework provides us with several methods we can call to write log messages at various priorities. These priorities are useful because they are ordered, and you can request to be shown only those messages that are a certain priority level or higher. Helium is a lightweight logging framework that connects to the logger API and prints messages to the terminal, but keeping the two separate means you can switch to a different logging service by changing only one or two lines of code.

To activate Helium, add this just before the `let router = Router()` line:

```
HeliumLogger.use()
```

Rebuild your code then run it again, and this time you'll see Kitura breaks its silence:

```
VERBOSE: init() Router.swift line 55 - Router initialized
VERBOSE: run() Kitura.swift line 71 - Starting Kitura
framework...
VERBOSE: run() Kitura.swift line 73 - Starting an HTTP Server
on port 8090...
INFO: listen(socket:port:) HTTPServer.swift line 128 -
Listening on port 8090
```

Those messages were being logged all along, but without a logging service they were just disappearing into the ether. Notice the “VERBOSE” and “INFO” markers at the beginning – that’s the priority levels I mentioned to you. There are seven in total, ordered very roughly like this:

- “entry” is used to mark a function being entered.
- “exit” is used to mark a function being exited. Along with “entry” this is used to help you follow program flow when debugging.
- “debug” is used for internal developer messages to help you diagnose issues.
- “verbose” is used for messages that are unimportant; it’s the lowest priority message you’ll see by default.
- “info” is used to log informational messages
- “warning” is used to log problems that should be investigated further.
- “error” is used to log serious problems that need to be addressed.

Now, in practice my definition of “info” and “verbose” might not match yours, and that’s OK – realistically they mean whatever you want them to mean. You can adjust the log detail you see by passing a parameter to the `use()` method, like this:

```
HeliumLogger.use(.info)
```

If you try that, you’ll see running the app now writes only one log line at first:

```
INFO: listen(socket:port:) HTTPServer.swift line 128 -
```

## Listening on port 8090

Whenever you load <http://localhost:8090> in your browser, more log lines will be written.

Broadly speaking, you should use the `.debug` level while debugging, and either `.info` or `.warning` when in a live environment.

Whether or not a logger is configured, you can write log messages at any time using one of the priority levels. If a logger has been created then it will be used, otherwise it gets silently ignored. To demonstrate this, add this Taylor Swift quote directly before the call to

`Kitura.run()`:

```
Log.info("Haters gonna hate")
```

This time you'll see two INFO messages when you run the app.

Logging is something quite alien to most iOS and macOS developers, but it's a habit you need to learn if you're serious about server-side Swift. Remember, your code will probably end up running on a headless server in some anonymous Rackspace data center, so the only way you're able to detect problems and find out what happened is by logging. Get into the habit now, and it will pay off – I promise!

**Tip:** By this point you're probably tired of typing `swift build` to build the app, then `.build/debug/project1` to run it. Most terminals let you combine the two using `&&`, and the second part of the command will only be executed if the first command returned successfully. So, try running this command:

```
swift build && .build/debug/project1
```

If the build succeeds, the app will be launched; if the build fails for any reason, you'll see the errors and the app won't be launched. Once you've typed that command once, use the up arrow on your keyboard to repeat it.

## Routing user requests

At this point we have a project package, we have a Kitura server up and running on port 8090, and we're able to log messages freely. The next step is to ditch the Kitura welcome screen and replace it with our own content.

When a request comes in from a user, it has a path associated with it. That path might be “/”, i.e. the root of your website, or it might be something more complex like “/users/twostraws/votes”. There might also be a query string, for example “?start=2016-10&end=2016-12”.

Kitura's job is to read the path that comes in, e.g. “/users/twostraws/votes”, compare that against your list of available routes, and figure out which – if any – should be run. To make that work, you specify the list of available routes up front, then let Kitura handle the rest.

Let's start with a simple route that prints our “Hello, Kitura!” message again. Place this code before the call to **addHTTPServer()**:

```
router.all("/") {
    request, response, next in
    response.send("Hello, Kitura!")
    next()
}
```

I know it's only a handful of lines of code, but in that tiny slice you're getting a huge amount of Kitura's routing power.

First: the **all()** method. The HTTP specification defines a number of methods, which are actions that can be performed by clients. The most common is “GET”, which is used to retrieve information, but also common is “POST”, which is used to submit information. The lines between GET and POST are a bit blurred in practice, but in theory a “GET” request should never cause data to be modified. You can (and should!) bind routes to specific methods, but using **all()** is a good catch-all for now.

Second, “/”. This is the path we want to attach code to, which in this case is the root of our site – a single slash. This is the page that gets served when someone goes directly to <http://localhost:8090>. We'll write more interesting paths soon enough, don't worry.

Third, **request**, **response**, and **next**. Each path has a closure bound to it, which is the code to execute when the route is matched. The **request** parameter tells us about the user's request: what the full URL was, which headers were sent, whether any cookies were attached, what the query string was, and so on. The **response** parameter lets us send data *back* with the result of the request.

Those two are the easy parameters: one is for stuff coming in, and one is for stuff going out, request and response. The **next** parameter is quite different: it's a closure that, when called, tells the router to continue matching paths. This lets you attach multiple pieces of code to a single path, and have them all run in sequence.

Finally, **response.send()**. This lets us deliver content to the user, which might be a filename, it might be data in JSON format, or it might be text as seen in our code above. If you're using **next()** – which almost all of the time you will be – you can send content in multiple handlers, which are executed in the order they appear in your code.

Let's try it now – add this code after the existing route:

```
router.all("/") {
    request, response, next in
    response.send("Here's some more text!")
    next()
}
```

Build and run your server code again, then request “/” using a web browser. This time you'll see “Hello, Kitura!” followed by “Here's some more text!” – Kitura matches both routes, one after the other.

If you *don't* call **next()** then you need to end the request yourself using the **end()** method. This signals that no further content will be added, and causes Kitura to send your full response to the user. Kitura calls this method for you when you use **next()** and it reaches the end of its routes.

Take a look at this code:

```
router.all("/") {
```

```

    request, response, next in
    try response.send("Hello, Kitura!") .end()
// next()
}

router.all("/") {
    request, response, next in
    response.send("Here's some more text!")
    next()
}

```

There are three important things in that code.

First, Kitura makes extensive use of method chaining, which means that many methods you call on an object also return the object. So, calling **response.send()** actually returns the **response** object, which means we can immediately tack on **end()** rather than writing a second line of code.

Second, **end()** is a throwing method, which means it might fail. Normally this would mean adding lots of **do/catch** code every time we needed this simple operation, but helpfully the router's closures are already marked as throwing so we can just write **try** and let any errors bubble upwards to be handled by Kitura.

Third, I commented out **next()** in the first route. This means that the second route will never be executed, even though it matches the same path.

Now, you might be wondering what happens if you call **end()** but also call **next()** in the same route. Well, “nothing” is the answer: once **end()** has been called the content gets packaged up for delivery to the client, and anything else you try to add is effectively ignored.

We'll be looking at routing in more detail in the next two projects, but for this project – Million Hairs Veterinary Clinic – we're going to create three routes: a homepage, a staff page, and a contact page.

Modify main.swift to this:

```

import Kitura
import HeliumLogger
import LoggerAPI

HeliumLogger.use()
let router = Router()

router.get("/") {
    request, response, next in
    response.send("Welcome to Million Hairs")
    next()
}

router.get("/staff") {
    request, response, next in
    response.send("Meet our great team")
    next()
}

router.get("/contact") {
    request, response, next in
    response.send("Get in touch with us")
    next()
}

Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()

```

As a small tweak I've made each of those routes use `get()` rather than `all()`, because it's makes it clear these are routes that are fetched by users rather than submitted with new data.

# Writing HTML

This isn't a book about HTML, JavaScript, and CSS, but at the same time it's basically impossible to create websites without knowing them at least a little. The problem is, "MVC" means splitting our model (data), view (layout) and controller (logic) into distinct parts, so the last thing we want to do is have HTML embedded inside our Swift code.

Consider this code:

```
router.get("/") {
    request, response, next in
    response.send("<html>")
    response.send("<body>")
    response.send("<h1>Welcome to Million Hairs</h1>")
    response.send("</body>")
    response.send("</html>")
    next()
}
```

Think about that for a moment: Swift is a compiled language, which means when you make a small change inside a 1000-line file, that whole file needs to rebuilt. If your designer wants to try **<h2>** rather than **<h1>**, wants to add a **style** attribute to make the text blue, or wants to include a new JavaScript file, you'd need to rebuild all that because it means changing the Swift code.

Even if you're not a fan of MVC, having to rebuild your Swift just to make HTML changes is clearly inefficient. The standard solution for this is to use template engines, and Kitura comes with two built in: Mustache and Stencil. They do similar things so there's no point learning both; as a result, we'll be focusing on the Stencil framework for this book.

Templates let you write all your view code in HTML, JavaScript, and CSS. All of it – you put *no* HTML in your Swift unless you specifically want to generate it dynamically. Your Swift code instead acts as the controller in MVC: it's responsible for fetching data from your model, formatting it, then passing it on to the template for rendering.

Templates aren't just a "nice to have" in the web development world – they are fundamental.

In my own web work, I use templates to render web content, but also RSS feeds, Apple News feeds, Google AMP, Facebook Instant Articles, custom JSON, and more. As long as you ensure you keep your controller and views separate, you should be able to send the same data to a dozen different templates and get a dozen very different results.

In fact, templates are *so* fundamental to web development that Kitura comes with support baked in. All you need to do is choose a template engine, create your templates, then fill them with content.

Let's try it out now. We're going to use the Stencil template engine, which was installed when we added the Kitura dependency in Package.swift. To start using it, add this **import** line to the top of main.swift:

```
import KituraStencil
```

With that framework in place, we need to create a new instance of the Stencil template engine and attach it to the router. That takes only one line of code, so add this after the **let router =** line:

```
router.setDefault(templateEngine: StencilTemplateEngine())
```

The next step is to create some HTML. Like I said, templates allow us to separate our Swift code from our HTML. Kitura comes pre-configured to look for templates in a directory called "Views", so let's start there: create a directory called "Views" in your "project1" directory, next to "Sources" and "Tests".

**Warning:** "Views" has a capital V. Linux uses a case-sensitive filesystem, so if you create a directory caled "views" you will have problems.

This new Views directory is going to contain all our Stencil templates, which are made up of HTML with a few bonus features to customize presentation. For now we don't need anything special, so I'd like you to put this HTML into a file called home.stencil, inside the Views directory:

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

```
<h1>Welcome to Million Hairs</h1>
</body>
</html>
```

When rendered, that will show a big, level-one header saying “Welcome to Million Hairs”, but not much else.

The final step is to render that Stencil template when the “/” route is requested. We’ve already been using the `send()` method of Kitura responses to send custom text, but that same object has a `render()` method that is capable of rendering Stencil templates. You need to pass it the name of the template along with any context – for us the template name is just “home”, and we don’t need any context because we’re just going to render the plain HTML.

Replace your current “/” route with this:

```
router.get("/") {
    request, response, next in
    try response.render("home", context: [:])
    next()
}
```

In fact, because we’re going to be using `next()` in every route, I find it easier to use Swift’s `defer` keyword to ensure that `next()` always gets run. Try this instead:

```
router.get("/") {
    request, response, next in
    defer { next() }
    try response.render("home", context: [:])
}
```

That’s one page down, and it wasn’t hard at all. Let’s look at another easy page: “/contact”. Create a new template file in the Views directory called `contact.stencil`, giving it this content:

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

```
<h1>Get in touch</h1>
<p>Call us on 555-1234.</p>
</body>
</html>
```

That's a large heading and one paragraph of text – nothing special. To attach it to the “/contact” route, change that part of main.swift to this:

```
router.get("/contact") {
    request, response, next in
    defer { next() }
    try response.render("contact", context: [:])
}
```

Go ahead and run the updated code, and you should be able to visit both <http://localhost:8090> and <http://localhost:8090/contact> to see the correct content in both places.

It's usually about now that a manager wanders up to your desk and says what they *really* want is to have a company standard footer on every page. You could go ahead and modify home.stencil and contact.stencil, but deep down you just *know* that in 30 minutes an entirely different manager will wander up and say “that looks good, but it would look even *better* if it were [insert some random color here].”

The principle of Don’t Repeat Yourself (DRY) applies just as much to templates as it does to Swift code, so Stencil has a special “include” tag that embed one template inside another. To use it, just specify the template filename you want to pull in, including its “.stencil” file extension. So, add this line to both home.stencil and contact.stencil, just before the `</body>` line:

```
{% include "footer.stencil" %}
```

Notice the curious syntax: an open brace and percent sign to start, then Stencil instructions, then a percent sign and close brace to end. To finish up, create a footer.stencil file inside the Views subdirectory, giving it this content:

<p>Copyright © 2017 Million Hairs Veterinary Clinic</p>

**&copy;** is a HTML entity that gets replaced with the copyright symbol by web browsers.

Because templates are stored separately from your Swift code, you don't need to re-build your project when you've changed one of them. Instead, just press reload in your web browser, and you should see all the updates immediately – nice!

## Matching custom routes

You've seen how we can match specific routes such as “/” and “/contact”, but Kitura is also able to match route components that we *don't* know. Right now our staff page is loaded using the “/staff” route, but what we're going to do is create pages for “/staff/[some name here]” and load content dynamically.

Helpfully, Kitura makes this process nice and easy: you can write the path “/staff/:name” and it will automatically understand that to mean “/staff/” followed by any sequence of letters.” That means “/staff” and “/staff/” by themselves aren't enough to match, and “/staff/foo/bar” is too much and also won't match.

Inside the route closure, you can read whatever was used in place of `:name` by using the `request.parameters` dictionary. This has keys using whatever name you used in the path, minus the colon. In our case, that means `request.parameters[ "name" ]` should be set to a value.

Once we know the name of the staff member, we can load some information about them. We're not going to use a database just yet because you're already learning lots of other things. Instead, we're going to create a static dictionary with URL names as the key, and a one-line biography as the value, like this:

```
let bios = [
    "kirk": "My name is James Kirk and I love snakes.",
]
```

Finally, the interesting bit: passing data to the template. The previous two templates were mostly just plain HTML, albeit with an `include` tag added in there. This time we want to render custom content, namely the staff member's name and biography.

Any data you want to pass to a template is called its “context”, and so far we've been using `[ : ]` – an empty dictionary. The dictionary must have strings for its keys, because those keys are exposed as variables inside your templates, but they can have any kind of value: it might be a string, an integer, an array, another dictionary, or something else. Stencil templates are significantly laxer with types than Swift is, which means you can mix types freely and it will just figure out what works best.

What we're going to do is create an empty [**String: Any**] dictionary, then attempt to find a bio matching the name from the URL. So, if the user enters the URL “/staff/kirk” we'll find James Kirk, who likes snakes. If they enter the URL “/staff/vzbxks” – a name not in our staff list – we'll send the empty dictionary to be rendered.

OK, that's everything you need to know – replace the existing “/staff” route with this:

```
router.get("/staff/:name") {
    request, response, next in
    defer { next() }

    // pull out the name of the staff member in question
    guard let name = request.parameters["name"] else { return }

    // create some dummy data to work with
    let bios = [
        "kirk": "My name is James Kirk and I love snakes.",
        "picard": "My name is Jean-Luc Picard and I'm mad for
cats.",
        "sisko": "My name is Benjamin Sisko and I'm all about the
budgies.",
        "janeway": "My name is Kathryn Janeway and I want to hug
every hamster.",
        "archer": "My name is Jonathan Archer and beagles are my
thing."
    ]

    // create the context dictionary we'll pass to the template
    var context = [String: Any]()

    // attempt to find a staff member by this name
    if let bio = bios[name] {
        // we found one – set some context details
        context["name"] = name
    }
}
```

```

    context[ "bio" ] = bio
}

// render the template with whatever we have
try response.render( "staff" , context: context)
}

```

That's the Swift code done for now, so let's look at staff.stencil. We could very easily make this two templates, one for staff members and one for invalid names, but using a single template lets me demonstrate conditions.

To write staff.stencil, there are three things you need to know:

1. You can check whether your template context contains a specific key using an **if** condition, for example `{% if name %}` checks whether the “name” variable is set in our context. There are matching **else** and **endif** keywords too.
2. To print a value from the context, write its name inside double braces, like this: `{{ bio }}`. This is different to the `{% %}` used for tags.
3. Stencil comes with built-in filters to modify values, such as **capitalize**, **uppercase**, and **lowercase**. They match their Swift equivalents, but I'll demonstrate how to write your own filter later.

We're going to use all three of those things right now: if we have a **name** variable in our context, we'll send it through the **capitalize** filter so that “kirk” becomes “Kirk”, then print it out along with the **bio** value. Otherwise, we'll write that there's an unknown staff member.

Create staff.stencil in Views, and give it this content:

```

<html>
<body>

{%
  if name %
<h1>{{ name|capitalize }}</h1>
<p>{{ bio }}</p>

```

```

{%- else %}

<h1>Unknown staff member</h1>
<p>We didn't recognize that person.</p>
{%- endif %}

{%- include "footer.stencil" %}

</body>
</html>

```

Notice how Stencil filters are applied by writing the context key, `name`, followed by a pipe symbol, `|`, followed by the filter name, `capitalize`.

At this point you're probably starting to think templates are complicated. When you first see them in serious usage, like above, you start to get "brace blindness" – all the `{{` and `{%` marks start to just look like line noise. Some people like to indent their templates to help clarify structure, like this:

```

<html>
<body>

{%- if name %}

    <h1>{{ name|capitalize }}</h1>
    <p>{{ bio }}</p>

{%- else %}

    <h1>Unknown staff member</h1>
    <p>We didn't recognize that person.</p>
{%- endif %}

{%- include "footer.stencil" %}

</body>
</html>

```

At least with that you can see exactly where the condition starts and ends – do whatever works best for you.

Anyway, back to the template: when the page is loaded, Stencil parses the template, evaluates the **if** tags, then replaces the double-braces with their values from whatever context we pass. That rendered text gets passed back to the router, which gets sent on to users.

As you can see, templates aren't *devoid* of logic, they just focus on *presentation logic*. All the curation of data should happen inside your controller, but the template can still have some logic in order to display data appropriately.

You've already seen an **if** tag, but templates can also do loops when provided with arrays. To try this out, let's extend our staff template so it provides links to other staff members. First, add this to main.swift, directly after the **var context =** line:

```
context[ "people" ] = bios.keys.sorted()
```

That takes the URL names from the dictionary – “kirk”, “picard”, etc – sorts them into alphabetical order, then stores them in the context that gets passed to Stencil for rendering.

Back in staff.stencil, you can loop over an array using the same **for-in** syntax you're used to from Swift. So, we can use the HTML **<ul>** and **<li>** tags to create an unordered list of people, each with links to their staff page. HTML links are created using the **<a>** tag (“anchor”), where the **href** attribute contains a path to the target page.

Add this code to staff.stencil, directly after the **{% endif %}** line:

```
<p>Choose a staff member to view:</p>
<ul>
  {% for person in people %}
    <li><a href="/staff/{{ person }}">{{ person|capitalize }}</a></li>
  {% endfor %}
</ul>
```

Make sure you re-run **swift build** (or build in Xcode) so that the new data is sent into the template context!

One last thing before we move on: Stencil loops have a handy **{% empty %}** tag, which lets

you provide content to be displayed if the loop is empty. For example, we could rewrite the previous loop like this:

```
{% for person in people %}  
<li><a href="/staff/{{ person }}">{{ person|capitalize }}</a></li>  
{% empty %}  
<li>No staff members found</li>  
{% endfor %}
```

Again, these loops are a form of logic inside our templates, but the logic is always about how to present data. That is, your templates don't have the ability to go off and fetch more data themselves, they are just responsible for deciding what to do with the data they were given by Swift.

## Making it look good

So far you've created a Swift package, added a server for port 8090, started a logger, added some paths, and rendered content using a template engine. That's a heck of a lot for a first project, but I have a dilemma. You see, even after all this work the end result looks awful, and the last thing I want is for you to think server-side Swift sucks and give up now.

So, the dilemma: do we dive into some HTML and CSS to make this project look better, or do I wave my hands and say, "imagine you wrote some brilliant HTML here"?

I'm going to take a middle ground, and what's more I'm going to use this as an opportunity to teach you some three useful new things. First, we're going to use Bootstrap, which is a project that came out of Twitter designed to help provide a basic, functional template for websites. It's no substitute for having content on the site, but at least it provides a semblance of realism. Second, we're going to use template inheritance, which lets you build one template on top of another. Third, we're going to use static file serving, which lets us serve up JavaScript, CSS, images, and other non-dynamic content.

Let's start with inheritance, which is easier than it sounds. Right now all three of our templates – home, contact, and staff – share the same start and end:

```
<html>
<body>
[ page content ]
</body>
</html>
```

When we introduce Bootstrap in a few moments, the amount of shared header and footer code is going to massively increase. If we carry on duplicating it in each template, it soon becomes impossible to maintain.

Now, we *could* use the **include** tag to inject a header and footer, but Stencil has a smarter approach called *blocks*. These let us create a master template containing as much content as we want, then mark out certain segments as gaps that can be filled by child templates.

Here's our current `home.stencil` file:

```
<html>
<body>
<h1>Welcome to Million Hairs</h1>
</body>
</html>
```

What we want to do is create a master.stencil file that contains the header and footer, plus a block marker for things we want to provide inside a child template. Let's start with something simple – create a new template called master.stencil, and give it this content:

```
<html>
<body>
{%- block body %}{% endblock %}
{%- include "footer.stencil" %}
</body>
</html>
```

That creates a block called “body”, and includes the footer – it's going onto every page, so putting it in the master makes sense.

You can have as many blocks as you want, and child templates can fill in as many of them as they want. To make things more interesting, let's add a second block – add this just after the `<html>` tag in master.stencil:

```
<head>
<title>{%- block title %}{% endblock %} - Million Hairs</
title>
</head>
```

The finished master.stencil file should look like this:

```
<html>
<head>
<title>{%- block title %}{% endblock %} - Million Hairs</
title>
```

```
</head>
<body>
{ % block body % }{ % endblock %
{ % include "footer.stencil" %
</body>
</html>
```

Notice how the `<title>` tag – the HTML way of showing text in the browser title – uses both a block and some built-in text. This means Stencil will blend the child title and the master title into one single HTML tag.

Just to avoid confusion, you can name your blocks whatever you want. I've named them “title” and “body”, but it could be “fish” and “bookshelf” for all it matters. All that matters is that you use the same names in the parent and child template.

Let's take a look at a child template now. This needs to use the same `block` tag that was used in the parent, as well as a new `extends` tag that makes the inheritance work. Apart from those `block` tags, each page should contain only its own content.

Here's the new `home.stencil` file:

```
{ % extends "master.stencil" %

{ % block body %

<h1>Welcome to Million Hairs</h1>

{ % endblock %

{ % block title %}Welcome{ % endblock %}
```

The first line, `extends`, tells Stencil that we want to base this template on `master.stencil`. There are then two `block` tags, but notice that I've put them in reverse. That's just to prove a point: you can put them in any order you want, because all that matters is using the same block names in the child and parent templates.

When Stencil renders `home.stencil`, it automatically blends its two blocks into the correct

locations in master.stencil to produce a combined result.

You should be able to convert contact.stencil and staff.stencil by yourself, because they follow the same approach as home.stencil: make them extend from master.stencil, then write a **title** block and a **body** block containing their content.

Give it a try now; my version of contact.stencil is below:

```
{% extends "master.stencil" %}

{% block title %}Call us{% endblock %}

{% block body %}
<h1>Get in touch</h1>
<p>Call us on 555-1234.</p>
{% endblock %}
```

And here's staff.stencil – again, it's just a matter of extending master.stencil then placing block markers around the page's unique content:

```
{% extends "master.stencil" %}

{% block title %}Staff{% endblock %}

{% block body %}
{% if name %}
<h1>{{ name|capitalize }}</h1>
<p>{{ bio }}</p>
{% else %}
<h1>Unknown staff member</h1>
<p>We didn't recognize that person.</p>
{% endif %}

<p>Choose a staff member to view:</p>
<ul>
```

```

{%- for person in people %}
<li><a href="/staff/{{ person }}">{{ person|capitalize }}</
a></li>
{%- empty %}
<li>No staff members found</li>
{%- endfor %}
</ul>
{%- endblock %}

```

So, that's template inheritance. I hope you can see why it's better than just using **include** tags – the ability to inject blocks freely, even out of order, lets us create incredibly flexible templates.

The second technique we're going to use is static file serving, which lets Kitura deliver files directly to the user without having to run them through our Swift code to evaluate them further. This is used for JavaScript, for images, for CSS, fonts, movies, and any other assets that you want to serve wholesale to users.

Similarly to templates and their Views directory, Kitura comes preconfigured to load static files from a specific directory in your package. It's called "public", which, cunningly, starts with a *lowercase* "p" in contrast to "Views" uppercase "V". Let me write that again to avoid confusion:

**Warning:** You need to create a directory called "public" in your project directory. If you create one called "Public" – with an uppercase "P" – you *will* have problems when you deploy to a Linux server.

As you saw previously, Kitura's router creates a chain of route handlers that get called in sequence thanks to the **next()** closure. We can take advantage of that to add support for static files in just one line of code. Add this just after the call to **setDefault()** in main.swift:

```
router.all("/static", middleware: StaticFileServer())
```

Even though it's just one line of code, it introduces two new things. First,

**StaticFileServer()** is a Kitura class dedicated to – wait for it! – serving static files. This is separate from serving regular files, because a static server is designed to be as simple and fast as possible; there’s no need for complex logic when you’re just sending the contents of files. I’ve chosen the name “static” for my content, but you can use whatever you want.

Second, middleware. This is a complex beast, but is essentially a layer of code you can inject between the user’s request and your routes to handle it. What our line of code does is provide an ultimate fallback for all paths that delivers the matching filename from the “public” directory. So, if the user requests /static/images/cat.jpg”, Kitura will look for public/images/cat.jpg and serve it if it exists.

You can install lots of middleware for any given request, and indeed you can even register lots of custom static file server directories if you want, but that one line of code is enough for now.

Now that we have template inheritance and static file serving, we can finally serve up Bootstrap. Bootstrap is a collection of CSS and JavaScript that lets you build responsive websites in just a few minutes. It uses a 12-column grid system, where you give each component a size based on how many columns it takes up – 12 being “full width”. This lets your site resize itself automatically when the browser changes width, or is viewed on small devices.

I’m not going to paste vast swathes of Bootstrap code here, because it’s a waste of time and space. Instead, I’ve provided my files for you in the project files you should have received with this book. Take all of my “Views” and “public” folders, and use them to overwrite yours.

Here’s what you’re getting:

- I added a line of text and a button to home.stencil, to make it look a little less deserted.
- My master.stencil file contains the same **title** and **body** blocks, but now has much more Bootstrap content around it to provide a basic layout.
- My “public” directory contains Bootstrap’s CSS, JS, and fonts directories, so you can use all of Bootstrap’s styles in your HTML.
- In the “css” folder you’ll also see site.css, which contains a small amount of custom CSS to make the site work. It forces the page to be 2000px high so you can try scrolling – the navigation bar is fixed.

- In the “images” folder you’ll find logo.png, a small Million Hairs logo image.

Because the new master.stencil file uses the same blocks as before, the contact and staff templates will immediately adopt the new look and feel – no need to change the Swift code.

If you’re curious to learn how Bootstrap works, I encourage you to go to <http://getbootstrap.com> and download the source code for Bootstrap. It includes a “docs” directory, which in turn contains an “examples” subdirectory, where you can find various examples of Bootstrap being used. In particular, look at the Carousel example to see the 12-column grid in action.

Bootstrap is powerful, flexible, and widespread. If you intend to use it in your own projects, I encourage you to visit <http://wrapbootstrap.com>. There you can buy pre-made Bootstrap themes for under \$20 that look great, and come with a range of useful plugins and pre-made designs built in.

## Wrap up

This has been a slow, gentle introduction to server-side Swift. You've used the Swift package manager to create and build app packages, created a Kitura server to serve requests, learned how to define routes and attach code to them, tried your hand at Stencil templates including conditions, loops, and inheritance, and also learned how to serve static content.

That's more than enough for one project, but I hope you're starting to see some parallels between server-side Swift and regular iOS or Mac development. Sure, the setup process is quite different, but once you're going you still need to think about keeping your code organized as MVC. We haven't actually touch the M part yet, the model, because we just injected a dictionary directly – we'll come onto that in the very next project.

## Homework

First, something easy: we made the footer.stencil file to include a standard footer on every page, but now that we have template inheritance that's no longer needed. So, go ahead and merge footer.stencil into master.stencil.

Second, we made routes for individual staff members, but nothing for “/staff”. As a result, going to that URL shows an error page rather than something useful. Your job is to add a new route to match “/staff” by itself and show some useful information. Give careful thought to DRY – Don't Repeat Yourself.

Finally, something optional: play with Bootstrap. I know it's a bit confusing at first, but Bootstrap's approach to flexible layout only takes about an hour to learn to a degree where you can make useful websites.

You'll see lots of code like this:

```
<div class="row">
  <div class="col-lg-4">Stuff</div>
  <div class="col-lg-4">Stuff</div>
  <div class="col-lg-4">Stuff</div>
</div>
```

Remember, Bootstrap uses a 12-column grid, chosen because 12 can be divided by 1, 2, 3, 4, 6, and 12. The above code will create three equally sized columns. You can subdivide columns further by creating a new row inside them.

As you've probably figured out, the "col" part refers to a column, and the "4" part refers to "take up 4 of the 12 columns." This means you could use 10-1-1, 6-2-4, 6-6, or whatever distribution you want.

The "lg" part is more complex, but lets you create different layouts for different-sized screens. "Lg" refers to "large devices", which is screens that are more than 1200px wide. You can also use "md" for medium-sized devices (992px or wider), small devices (768px or wider), or "xs" for extra-small devices (less than 768px). You can use class names like `.col-xs-6 .col-md-4` to size a column differently depending on the user's device.

The easiest way to explore Bootstrap is to look in the docs/examples folder of the Bootstrap source code. Try taking parts of each of the examples and building your own thing, making sure to take the associated CSS at the same time.

# **Chapter 2**

## CouchDB Poll

## Setting up

This project is the polar opposite of project 1. That's not because I enjoy torturing you! Instead, project 1 was there to teach you some fundamental – if a bit boring – techniques, such as the package manager and routing. Now that you understand those, it's time to tackle a more complex project that lets you build on your Kitura foundations.

In this project we're going to build a voting API: a system that lets users create topics to vote on, cast votes, and get results. This is an *API* – an application programming interface – which means that we expose a series of endpoints that someone else can call to control our code. We're *not* going to create a front-end this time, which means we won't be using Stencil templates. Instead, we will vend JSON based on the URL that was requested, and it's down to clients to figure out how to display it.

Alongside introducing JSON for the first time, this project is geared towards letting you get your first experience with CouchDB. This is a NoSQL database, which means you can store and retrieve data without using SQL – an entirely separate language. Learning to use a database isn't easy, but is *important*, so most of the work in this project is going to be getting to grips with CouchDB.

Developing an API is tricky if your experience so far is with client-side development. There's no user interface to view, and in fact if there *were* a user interface it could exist in multiple places: there could a web user interface, an iOS user interface, an Android user interface, and so on, all pointing at the same API.

Go ahead and create a project2 package in your “server” directory. You should have made a shared Docker container, so it will automatically appear inside Docker.

If you've forgotten how to create a new package, I'll repeat the instructions this time. Open a terminal and run these commands:

```
cd Desktop/server  
mkdir project2  
cd project2  
swift package init --type executable
```

If you're already in the Desktop/server directory, skip the first command.

We need to add dependencies for Kitura and HeliumLogger, but this time we need to use Kitura-CouchDB instead of Kitura-StencilTemplateEngine. You can copy the dependencies from your previous project if you want, just switching out the last one.

Here's how your finished Package.swift file should look:

```
import PackageDescription

let package = Package(
    name: "project2",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
CouchDB.git", majorVersion: 1)
    ]
)
```

Go ahead and run **swift build** now so that the dependencies are downloaded and built at least once.

## CouchDB from scratch

**Tip:** In this chapter we're going to be exploring CouchDB without writing any code. You *could* in theory skip this chapter, but I would strongly recommend you don't!

Databases are complex beasts. That doesn't mean they are hard to learn or hard to use, just that there's a huge amount of work going on behind the scenes – if you think to yourself, "wow, this is hard!" just be grateful that database engineers the world over have already done the overwhelming majority of work for you!

Very roughly, databases can be grouped into two types. The first use a dedicated language for creating queries to add, delete, update, and read data. This makes it extremely expressive, because the language lets you create any kind of query you can imagine: you write code to tell the database what you want, and it figures out how to fetch it in the most efficient way possible.

The language used is called Structured Query Language, or SQL for short. If you were curious, it's pronounced as either "Ess Cue Ell" or "Sequel", but if you're one of those people who pronounces it "Sequel" then I'm afraid you won't be getting any invites to *Chez Hudson*. Popular SQL databases include MySQL, PostgreSQL, Microsoft SQL Server, Oracle, and DB2.

The other common database type is in direct opposition to the first type, and is usually called NoSQL. As you might guess, these databases don't use SQL, but still operate in a similar way: you tell it what data you want, and it figures out how to provide that data back to you.

Removing SQL from the equation does remove the ability to create perfectly crafted queries – remarkably enough, SQL is Turing-complete! – but it also makes them dramatically simpler to learn. Popular NoSQL databases include MongoDB, CouchDB, Cassandra, Redis, and Riak.

I'm not interested in holy wars between SQL and NoSQL, so this book is going to teach you both. We're starting with CouchDB here because it's easier to learn, both from a database perspective and from a coding perspective. CouchDB is one of many NoSQL databases you can choose from, but realistically if you learn one you can move to another easily enough.

OK, enough of me drivelng on about databases – it's time to get started with CouchDB. This is a separate piece of software that runs as a server, so you'll need to follow some instructions to get started:

1. If you're using my Docker image, you already have CouchDB installed. Pat yourself on the back! Run the command “/etc/init.d/couchdb start” to start the CouchDB service.
2. If you're using a Linux virtual machine, or Linux on a cloud server, you can run this command: “apt-get install couchdb curl”. This will need a number of dependencies, so when you're asked “Do you want to continue?” please press Return to accept the proposal. Now run “/etc/init.d/couchdb start” to start the CouchDB service.
3. If you're using macOS, go to <http://couchdb.apache.org/#download> and click the download button for macOS, either for v1.6 or v2.0. Extract the zip file, then launch the application it contains. This will place an icon into your status bar so you can see CouchDB is running, and will launch a web browser showing an admin console.

You can verify that CouchDB is installed and working correctly by going to the URL <http://localhost:5984/> in your web browser. All being well you'll see something like this:

```
{ "couchdb": "Welcome", "uuid":  
"25078d5b61f95463eb15c6a257be43d4", "version": "1.6.0",  
"vendor": { "name": "Ubuntu", "version": "15.10" } }
```

Or this, if you installed CouchDB v2:

```
{ "couchdb": "Welcome", "version": "2.0.0", "vendor": { "name":  
"The Apache Software Foundation" } }
```

**Warning:** By default, CouchDB runs in an insecure mode called the Admin Party, presumably because anyone can come to your party, clean out your drinks cabinet, then throw up on your carpet. If you're looking to use CouchDB on a live server, you should either seek professional security advice, use a cloud-based database system such as Cloudant where the work is done for you, or study carefully section 1.6 of the CouchDB documentation, which covers security in detail.

CouchDB responds to HTTP methods like GET and POST, but also PUT, DELETE, and more. Before we write any Swift code to work with the database, we're going to use a command-line tool called Curl – it comes pre-installed on macOS and my Docker image, and on Linux it was installed with the “apt-get” command I gave you to run.

Open a terminal now and run this:

```
curl -X GET http://localhost:5984
```

You should see the same output you received in your web browser. The “-X GET” part means “make this a GET request,” i.e. fetching data. That’s the default type of request, but it’s good to be clear in these early days.

The response we received back from CouchDB is *JSON* – JavaScript Object Notation – which is a simple, efficient way of transferring data. It’s similar to the combination of a Swift dictionary and a Swift array, and can hold nested data too.

When you see braces, `{` and `}`, it signifies dictionary values, and when you see brackets, `[` and `]`, it signifies array values. If I add some spacing to the previous JSON it should be a little clearer:

```
{
    "couchdb": "Welcome",
    "version": "2.0.0",
    "vendor": {
        "name": "The Apache Software Foundation"
    }
}
```

Even though the JS part suggests it’s restricted to JavaScript, JSON has become the de facto data format of the web, even surpassing XML. It does a similar job to XML, but is significantly more concise.

CouchDB works with JSON natively: not only will it serve you data in JSON format when you request it, but it wants to receive data in JSON when you’re making changes.

To help you understand how CouchDB works, we’re going to create some dummy data now, using JSON and Curl. If you’re using Docker, you can run these commands either inside Docker or in your regular macOS terminal – both should work fine.

We're going to be using the address <http://localhost:5984> a lot, so to save some time we can create a terminal variable called \$COUCH. Run this command in your terminal window:

```
COUCH="localhost:5984"
```

That creates a variable called **\$COUCH** to save us some typing. It means these two commands are identical:

```
curl -X GET $COUCH  
curl -X GET localhost:5984
```

To fetch information about a database, use its name as the path in your URL. So, if we had a database called “polls”, we could fetch its information by requesting the URL <http://localhost:5984/polls>. As we have the **\$COUCH** variable in place, we can use that to make the command shorter.

Try running this command now:

```
curl -X GET $COUCH/polls
```

You'll see an error message saying that the database doesn't exist, which is fair enough – we haven't made it yet! To fix that, we're going to use the HTTP PUT method to create the database.

```
curl -X PUT $COUCH/polls
```

You'll get `{"ok": true}` back, signifying that the database has been created successfully. Now run the GET request again, and you'll see a lot more information come back: how big the database is, how many documents it contains, whether it's currently being compacted, and so on. Bluntly, we don't really care about this information, but it's useful because it shows the database is definitely working!

If you run the PUT request a second time, CouchDB will report an error because it already exists. If you genuinely want to recreate it, you'll need to delete it first. Try running these commands:

```
curl -X PUT $COUCH/polls
curl -X DELETE $COUCH/polls
curl -X GET $COUCH/polls
curl -X PUT $COUCH/polls
curl -X GET $COUCH/polls
```

The first command will try to create the database even though it already exists, so you'll get an error. The second one deletes the database, so when the third command tries to fetch its information you'll see the "database does not exist" error. Command four recreates the table, so the final command should put us back to where we were before.

PUT literally means "put this object where I say." There is no "this object" in our request, but CouchDB figures it out.

Now that we have a database, we can create a document. To begin with we're going to use a simple data structure: all polls have a string for their title (e.g. "Which is better: iOS or macOS?"), two strings for the two available vote options ("iOS", and "macOS" in this case), and two integers to track how many votes each option has, starting at 0.

Try running this command:

```
curl -X POST $COUCH/polls -d '{"title": "Which is better: iOS or macOS?", "option1": "iOS", "option2": "macOS", "votes1": 0, "votes2": 0 }'
```

You'll get an error back; don't worry about it for now. Before we look at it, there are three things to note about the command. First, we're using the "POST" method, to create the document. If you POST the same thing multiple times the document will be created multiple times, which is different to the way PUT identified a resource uniquely.

Second, the "-d" parameter to Curl lets us send data to the server. We're sending a JSON structure that creates an example poll, all encapsulated inside single quotes to avoid problems with clashing double quotes in strings.

Third, notice how integers aren't placed in quotes. JSON distinguishes between numbers and

strings, so it's important to use quotes correctly.

Now, that command produced an error:

```
{"error": "bad_content_type", "reason": "Content-Type must be application/json"}
```

Like I said, CouchDB sends and receives JSON, so it's confused that we're sending it plain text when creating a document. Now, we know that it's JSON, but that's not good enough for CouchDB: it needs a special HTTP header that says "you're being given JSON."

Curl has a special parameter that lets us send custom HTTP headers, but it's annoying to type repeatedly so let's create another terminal variable:

```
JSON="Content-Type:application/json"
```

Now run the command again using that content type

```
curl -X POST -H $JSON $COUCH/polls -d '{"name": "Which is better: iOS or macOS?", "option1": "iOS", "option2": "macOS", "votes1": 0, "votes2": 0 }'
```

You'll get back something along the lines of this:

```
{"ok": true, "id": "619d54a4ac9c103d962ba92ac6002fb2", "rev": "1-90130bae5c47c4f582a7577b147fbc62"}
```

The actual string of letters and numbers will be different in both places, but you'll see several things in common:

- The "ok: true" part tells us the operation was successful.
- Each document has a unique "id" value. This is called a universally unique identifier, and CouchDB made one for us.
- Each document has a "rev" value. This starts with a number that tells us how many times the document has been changed, but then there's a unique identifier afterwards. This revision system is used to stop two users updating the recording.

**Note:** Having a revision number might make you think that CouchDB stores older versions of your data. It doesn't, or at least *probably* doesn't – you shouldn't rely on it.

Now that we've inserted some data, we can read it back out. If you remember, we used this command to retrieve information on the “polls” database:

```
curl -X GET $COUCH/polls
```

To pull out one specific document, just append its ID to the URL. My ID started with “619d54”, but yours will be different so you'll need to adjust the command accordingly.

Here's the command I used to fetch the document we inserted:

```
curl -X GET $COUCH/polls/619d54a4ac9c103d962ba92ac6002fb2
```

You'll get back more JSON, which will contain the values we added (name, option1, etc), plus the ID and revision values. Here's what I got back:

```
{"_id": "619d54a4ac9c103d962ba92ac6002fb2", "_rev": "1-90130bae5c47c4f582a7577b147fbc62", "name": "Which is better: iOS or macOS?", "option1": "iOS", "option2": "macOS", "votes1": 0, "votes2": 0}
```

You can request *all* documents in a database using a special URL:

```
curl -X GET "$COUCH/polls/_all_docs?include_docs=true"
```

To change an existing document, just pull out its JSON, make any changes you want, then resubmit. If you leave the current ID and revision values intact, CouchDB will recognize this as an update.

Let's test this out now. I'm going to adjust my previous POST command so that it sends the exact same JSON I just got back from my GET request, including the ID and revision fields. Here's my new command – make sure you replace everything after “-d” with the JSON you got back from your own CouchDB instance:

```
curl -X POST -H $JSON $COUCH/polls -d '{"_id": "619d54a4ac9c103d962ba92ac6002fb2", "_rev": "1-90130bae5c47c4f582a7577b147fbc62", "name": "Which is better: iOS or macOS?", "option1": "iOS", "option2": "macOS", "votes1": 0, "votes2": 0}'
```

Here's what I got back:

```
{"ok":true,"id":"619d54a4ac9c103d962ba92ac6002fb2","rev":"2-09ea94c61331162b500d54a41cf0fa4c"}
```

Again, “ok: true” means the request worked, and the ID value hasn’t changed because all we’ve done is updated our record. However, the revision value *has* changed: it starts with a 2 now because this is our second version of the document.

Press the up cursor key then return to run the command again, but this time you’ll see an error:

```
{"error": "conflict", "reason": "Document update conflict."}
```

This is because our command specifies revision number 1, which is no longer the current revision – it got bumped to 2 after we made the first change. If we want to modify the document again we need to specify the new revision number, otherwise CouchDB thinks we’re trying to modify old data and refuses to comply.

You’ve seen how to create documents, read documents, and update documents, so all that’s left is *deleting* documents. To do that, use the DELETE method with the URL to the document, then write “?rev=” followed by the latest revision number.

**Note:** You should put the whole URL into double quotes to avoid problems with your terminal.

Again, you’ll need to use your own ID and revision values, but the command looks like this:

```
curl -X DELETE "$COUCH/polls/idgoeshere?rev=revisiongoeshere"
```

You'll get another "ok: true" response back from CouchDB, but you'll also see you get a new revision number too. This is because CouchDB distinguishes between "document never existed," and "document used to exist, but has now been deleted."

To demonstrate this, I'm going to fetch my document again:

```
curl -X GET $COUCH/polls/619d54a4ac9c103d962ba92ac6002fb2
```

As it's been deleted, CouchDB will return the following:

```
{"error": "not_found", "reason": "deleted"}
```

If the document had *never* existed the reason would be "missing" rather than "deleted". CouchDB calls this "deleted" record a tombstone: the data is gone, but it still records that it existed at some point.

We're going to come back to CouchDB more in project 4, but I think you're probably keen to get on and start using it now. Don't worry if you think it's complicated – it *is* complicated. To be fair, Core Data on iOS and macOS is equally complicated, and just wait until you meet SQL databases in project 7!

# Designing a JSON API

You've seen how CouchDB works entirely with JSON over HTTP on port 5984. We could, in theory, expose that server directly to clients: someone could code an iOS app so that it talked directly to CouchDB, creating polls and casting votes all from iOS. While that's certainly true, it also scales badly: you'd need to do all the work on iOS, then all the work on Android, on Windows, on the web, and so on. That results in huge duplication of logic, and is guaranteed to cause huge problems when you need to update all your client apps.

This is where server-side Swift comes in: we can create our own API on top of CouchDB, we can put all the logic on the server. This means we'll create API end points to list the available polls, create a new poll, and cast votes. This API will be responsible for checking all the data is correct, talking to CouchDB, and reporting back any errors. If we do our job right, any front-end apps that get built should be trivial – all the hard work is in our API.

Years ago I met Jacob Kaplan-Moss from the Django project, and he gave me some great advice for how to write great API: document how it's suppose to work first, then write the code to match your documentation. So, let's do that now – let's write down how our API ought to work, then write the code to match.

We're going to create three end points:

- “/polls/list” will be responsible for fetching the list of available polls, including their current votes. It won't accept any special parameters, but we'll make it return a JSON array of poll information. This will use GET, because nothing changes when it's called.
- “/polls/create” will create a new poll, so it needs to be given a poll name plus two voting options. This will use POST because it will modify our data.
- “/polls/vote/pollid/option” will cast a vote for a particular poll. The “pollid” should be the ID of the poll in question, and “option” should be either 1 or 2 depending on their choice. This will also use POST, because it modifies data.

I've chosen those because they teach an interesting spread of techniques: the first one does a simple GET, the second one does a POST with data sent as a form rather than over the URL, and the third one does a POST with data sent in the URL so you can get some more practice with routing.

So, that's how we'll *send* data, but we also need to consider how we *receive* data. There are three things we're going to send back:

1. A HTTP status code. In project 1 we just used `response.send()` to deliver HTML back to the user – we didn't actually say whether the request was successful. In this situation, Kitura silently adds a success code, but we're going to do this ourselves now.
2. Our response is going to be JSON, and will always contain a “result” dictionary that will contain a “status” value set to either “ok” or “error”. If it's set to “error” we'll also provide a “message” value with a helpful string.
3. Assuming the request was successful, the JSON from “/polls/list” will also include a “polls” array containing all the polls that are available.

Having the status in both HTTP and JSON isn't required, but it's important to at least know how to do both.

If you were working in a team, you'd almost certainly want to go into more detail: what exactly does the JSON look like? What are the precise names of the values that will be submitted when creating a poll? For now, though, that's enough for us to be getting on with, so let's set up our basic Kitura app to handle those three routes.

First, the easy stuff: importing the basic frameworks, setting up a logger and a router, then starting up a server on port 8090. This is identical to project 1, so please add this code to `main.swift` in the Sources directory:

```
import Kitura
import HeliumLogger
import LoggerAPI

HeliumLogger.use()

let router = Router()

// routes here

Kitura.addHTTPServer(onPort: 8090, with: router)
```

```
Kitura.run( )
```

So far that's just what we had previously, but that changes now: we're going to create the “/polls/list” route so that it outputs some basic JSON.

Kitura uses SwiftyJSON for all its JSON needs, which is awesome – I've been using SwiftyJSON since it was released, and it's such a natural, easy way to read and write JSON. SwiftyJSON is actually bundled inside Kitura so we already have it installed – all we need to do is add an **import** for it. So, add this line now:

```
import SwiftyJSON
```

Creating JSON using SwiftyJSON is trivial: just create dictionaries, arrays, strings, and numbers however you want, then use the result to create a **JSON** object. We can then use **response.send(json:)** to send it to the user, which automatically adds the “Content-Type: application/json” header for us.

I've already said that we're going to send back a result dictionary that contains a status code. The JSON we're looking to create is this:

```
{
    "result" : {
        "status" : "ok"
    }
}
```

We'll be adding more to it soon enough, but that's enough to start with. So, that's a dictionary (“status”: “ok”) inside another dictionary (“result”), which looks like this in Swift:

```
let status = ["status": "ok"]
let result = ["result": status]
let json = JSON(result)
```

That last line highlights just how great SwiftyJSON is: that converts our result/status dictionaries into JSON all in one.

One more thing before we write our first route: as well as sending back JSON, we also want to send a HTTP status code. There are *lots* of these to choose from, but the ones you’re like to want to use are:

- “OK”, for when everything went well: status code 200
- “forbidden”, for when the user doesn’t have permission to make that request: 403.
- “badRequest`, for when the client’s request is incorrect: 400.
- “internalServerError”, for when something went wrong in your code: 500.
- “movedPermanently”, for when you’ve changed where a URL is located: 301.
- “notFound”, for when the URL doesn’t exist: 404.

For information on these codes, as well as the others, see this Wikipedia page: [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes).

In the previous project, I explained that Kitura makes extensive use of method chaining – many methods you call on an object also return the object. It’s common to use this approach when setting an HTTP status code, because you can set the status and send content all in one line, like this:

```
response.status(.OK).send("Hello, world!")
```

If you want to send a 403, a 500, or some other code, just use **.movedPermanently**, **.internalServerError**, and so on.

OK, let’s put in place a first draft of the “/polls/list” route:

```
router.get("/polls/list") {
    request, response, next in
    defer { next() }

    let status = [ "status": "ok" ]
    let result = [ "result": status ]
    let json = JSON(result)

    response.status(.OK).send(json: json)
```

```
}
```

We'll come back to that soon enough, but first we can put in some stubs for the other two routes. Add these two now:

```
router.post("/polls/create") {
    request, response, next in
    defer { next() }
}

router.post("/polls/vote/:pollid/:option") {
    request, response, next in
    defer { next() }
}
```

In project 1 we used “/staff/:name” to match URLs like “/staff/picard”, but here we’re using two parameters: **:pollid** and **:option**.

That’s our basic routing in place, so if you build and run you should be able to get to <http://localhost:8090/polls/list> to see our basic JSON. It’s nothing exciting just yet, but it’s time to combine Kitura and CouchDB to make this project come to life!

## Bringing CouchDB and Kitura together

We've prepared CouchDB and we've prepared Kitura, but the two don't talk to each other – at least not *yet*. We already added the Kitura-CouchDB dependency in `Package.swift`, so we just need to update our code to read and write our data as needed.

We have three routes, and all three need access to CouchDB. So, we can connect to CouchDB once, and share that connection across our routes rather than continually creating new connections.

First, add this new `import` line:

```
import CouchDB
```

Now we need to get access to our “polls” database. This is done in three steps: defining how we want to connect, creating a connection to the CouchDB server, and preparing to use the “polls” database.

Defining how we want to connect is done with `ConnectionProperties`. This takes five parameters in total: the hostname or IP address, a port number, whether to use a secure connection, plus a username and password. Because CouchDB is running in admin party mode we can skip the last two parameters, so we can focus on the first three.

The host parameter should be set to wherever your CouchDB server is. We were using “localhost” with Curl, and we can use that here too. The same applies to the port parameter: we can use the same port number we used with Curl, 5984. The last parameter is new: if you set it to true it means Kitura will connect to CouchDB using HTTPS, and false will use HTTP. This doesn't really matter here because we're using the local server so there's no risk of our data going astray.

Once you've created a `ConnectionProperties` object, you can connect to CouchDB by creating a `CouchDBClient` object. This is initialized using the connection properties we just defined, and lets us access any of the databases on that server. Finally, we can use the `database()` method on `CouchDBClient` to get access to a specific database, which in our case is the “polls” database.

Time to convert that to Swift – put these three lines into main.swift, just after the `HeliumLogger.use()` line:

```
let connectionProperties = ConnectionProperties(host:  
"localhost", port: 5984, secured: false)  
let client = CouchDBClient(connectionProperties:  
connectionProperties)  
let database = client.database("polls")
```

Because that code is placed before our routes, we can use the same database connection no matter what the user is trying to do.

Now that we have access to the “polls” database we can retrieve the list of available polls using the `retrieveAll()` method. This is a bit confusing because it accepts a closure parameter to run when the documents have been fetched, which makes it look asynchronous – i.e., it goes away and runs its code while the rest of our main route closure executes, then calls our closure as a completion handler.

Despite how it looks, I need to warn you in advance that this method is *synchronous*, which means it will execute fully before the rest of our code. That being said, I think it’s dubious to rely on that in the future, because it seems like an implementation detail that may change at any point. To be safe, we’re going to rewrite the “/polls/list” route so that it’s safe to use no matter how our code is executed.

Let’s start with the easy part of the code. When you call `retrieveAll()` you can pass a parameter specifying that you want document contents to be returned. If you don’t specify this you’ll get back all their IDs, but not their detailed data. Like I said, you provide a closure to `retrieveAll()` that gets executed when the documents have been fetched, and you’ll be given two parameters: JSON representing the results of your fetch, and an error that may or may not be set.

If there’s an error, we’re going to send its `localizedDescription` property back to our client so that they can show something meaningful if they want. If there isn’t an error we’ll transform the JSON into our own format and send it on. For now, let’s skip the success case and write the code for failure, because it’s mostly what we have already.

Modify your existing “/polls/list” path to this:

```
router.get("/polls/list") {
    request, response, next in

    database.retrieveAll(includeDocuments: true) { docs, error
        in
            defer { next() }

            if let error = error {
                let errorMessage = error.localizedDescription
                let status = [ "status": "error", "message": errorMessage]
                let result = [ "result": status]
                let json = JSON(result)

                response.status(.OK).send(json: json)
            } else {
                // success!
            }
        }
}
```

Note that I moved the `defer { next() }` code so that it's *inside* the `retrieveAll()` closure, ensuring it's never run before we've finished loading our documents. I'm in two minds about this: the API *is* synchronous, and moving the `defer` call like this doesn't help code readability. I'm hopeful IBM will refactor its Kitura-CouchDB bindings to offer a truly synchronous option in design as well as functionality.

**Tip:** If you see an obscure linker error along the lines of “static (extension in Foundation):Swift.String” then add `import Foundation` to the top of your file.

So, that's the easy part – it's almost identical to what we had previously, with the obvious addition of to the call to `retrieveAll()`. Where things get more complicated is handling

success, because CouchDB delivers data to us in a format that we don't really want to just forward on to clients of our API.

We can replicate the results of `retrieveAll()` using Curl, so you can see exactly what CouchDB is sending. Run this command:

```
curl -X GET "$COUCH/polls/_all_docs?include_docs=true"
```

Assuming you still have a poll in the database, you'll see output like this:

```
{
  "total_rows": 1,
  "offset": 0,
  "rows": [
    {
      "id": "959ef5935064ed90f3f4ad35cf000570",
      "key": "959ef5935064ed90f3f4ad35cf000570",
      "value": {
        "rev": "1-91748a2d8b11a1afcc5c34cac68fe444"
      },
      "doc": {
        "_id": "959ef5935064ed90f3f4ad35cf000570",
        "_rev": "1-91748a2d8b11a1afcc5c34cac68fe444",
        "title": "Which is better: iOS or macOS?",
        "option1": "iOS",
        "option2": "macOS",
        "votes1": 0,
        "votes2": 0
      }
    }
  ]
}
```

If you don't have a poll, go ahead and create one now using Curl so you have something to work with.

As you can see, we get back a dictionary that contains three values: “total\_rows”, “offset”, and “rows”. The “rows” value is itself an array of dictionaries, each one containing one document in our result. However, here’s where things get a bit more confusing: each row in the result contains the ID and key of the result (which are the same), a “value” that contains its revision, then a “doc” dictionary that contains the real information we want.

What this means is that in order to get at the actual meat of our data, we need to loop over the “rows” array, then look inside the “doc” dictionary for each item.

You’ve already seen how easy it is to create JSON from dictionaries using SwiftyJSON, but that’s only part of its appeal. The other awesome thing it does is make it easy to pull values out of JSON, because it intelligently works around the unavoidable ambiguity of JSON.

To explain how this works, consider this small slice of JSON:

```
"doc": {  
    "title": "Which is better: iOS or macOS?",  
    "option1": "iOS",  
    "option2": "macOS",  
    "votes1": 0,  
    "votes2": 0  
}
```

Now, we can see that “doc” is a dictionary, and contains values for “title”, “option1”, “votes2”, and so on. But Swift *doesn’t* know that, and if you read values from a dictionary you’ll end up with optionals because the value might exist or might not. This problem gets worse with JSON because you often have dictionaries inside dictionaries inside dictionaries, so it quickly becomes messy to find a value when it might not exist, or its parent might not exist, or its grandparent might not exist, and so on.

SwiftyJSON provides a brilliant solution: you tell it what kind of data you want to receive, and it will ensure you definitely get that back. So, you can write code like `json["foo"]` `["bar"]` `["baz"]`.`stringValue` and be *guaranteed* to get a string back. If “foo” doesn’t exist, or if “foo” does exist but “bar” doesn’t, or if both “foo” and “bar” exist but “baz” doesn’t,

then you'll be given back an empty string. If all three *do* exist, then the string you get back will be their value. You can also use **intValue**, **arrayValue**, and more – SwiftyJSON will ensure you definitely get something back, with no optionals and no extra checking.

As you've seen, CouchDB sends back a dictionary with a "rows" value. We can loop over all the values inside there using this code:

```
for document in docs["rows"].arrayValue {
```

Again, the use of **arrayValue** means that if "rows" doesn't exist for some reason then an empty array will be used instead. So, the loop won't execute, but neither will it fail because the JSON wasn't what we expected.

It's time to replace the **// success!** comment with real code. This is going to create a new array of dictionaries to hold our polls, because each dictionary will store one poll. It will then convert that to JSON, and send it off.

Replace the **// success!** comment with this:

```
let status = ["status": "ok"]
var polls = [[String: Any]]()

if let docs = docs {
    for document in docs["rows"].arrayValue {
        var poll = [String: Any]()
        poll["id"] = document["id"].stringValue
        poll["title"] = document["doc"]["title"].stringValue
        poll["option1"] = document["doc"]["option1"].stringValue
        poll["option2"] = document["doc"]["option2"].stringValue
        poll["votes1"] = document["doc"]["votes1"].intValue
        poll["votes2"] = document["doc"]["votes2"].intValue

        polls.append(poll)
    }
}
```

```
let result: [String: Any] = ["result": status, "polls": polls]
let json = JSON(result)

response.status(.OK).send(json: json)
```

Build and run the server, then try visiting <http://localhost:8090/polls/list> in your web browser or fetching it with Curl. That's the first route complete!

## Creating CouchDB documents

You'll be pleased to know that it's trivial to create new documents using CouchDB: just pass it some valid JSON, and you'll get back a document ID and revision. What *isn't* trivial is creating that JSON, because Kitura has a slightly roundabout way of delivering submitted data.

Part of the reason for the complexity is entirely justified, and summed up in one simple rule: never trust data from users. It might be there, it might not be there, it might be there but incorrect, it might be there but empty, or worse it might be there but contain malicious content'); DROP TABLE SERVER-SIDE-SWIFT;--

(If that went over your head, you should probably read <https://xkcd.com/327/>.)

Now, clearly we want our clients to do validation on their end so that ideally any requests to our API are already safe. That's absolutely true, but it's *not* a replacement for being secure on the server. Remember: never trust data from users. So, validate data in your client apps, then validate again in your web apps.

We're going to start filling in the "/polls/create" route, but it's done in four steps:

1. Telling Kitura we want to parse any data the user sends us. "Parsing" is the process of converting text into something we can use in our program.
2. Check that some data has been submitted. There's a **body** property on the **request** parameter for our route closure, but it's optional so we need to unwrap it.
3. Attempt to pull out the list of values that was submitted. There are several ways values can be submitted, but we're using "URL encoded".
4. Once we have the submitted form data, we're going to check that we have values for "title", "option1", and "option2", and that all three have at least non-whitespace character.

If all of those succeed, we have everything we need to create a new poll so we can continue to CouchDB. But if any of them fail – if a field is missing, for example – then we're just going to return a code 400, "bad request". This means the client got something wrong and should not attempt to repeat the request without fixing their mistake.

If this were a production API, you should really send back meaningful JSON error messages

explaining what the client got wrong, but for this project we're just going to stick with the HTTP error.

So, four steps: tell Kitura we want to parse user data, check that something was submitted, attempt to pull out URL encoded values, and finally ensure our three fields have meaningful values.

The first step is easy enough thanks to Kitura's middleware system: we need to register some middleware to parse the body of requests posted to “/polls/create”. Kitura uses a blanket **BodyParser** class to handle parsing, which internally figures out how to parse data depending on what it's given. That might sound cleverly flexible, but the end result of each parsing type is quite different so there's no easy way to switch between them.

Add this route *before* the existing “/polls/create” route:

```
router.post("/polls/create", middleware: BodyParser())
```

With that in place, it means by the time our second “/polls/create” route is executed Kitura will have parsed the submitted data for us.

The second step is to check that some data was submitted. The **request** parameter passed to our route closure has a **body** parameter that contains the parsed body. We don't know what *type* of data it is just yet, but we can at least check that it exists.

Keeping in mind that we never trust user data, we're going to check and unwrap the **body** parameter safely, but if it's *missing* we'll immediately send back the **.badRequest** status and exit the closure. Here's how that looks in Swift:

```
guard let values = request.body else {
    try response.status(.badRequest).end()
    return
}
```

That's two steps down, but the remaining two are more complicated. Step three is to pull out the URL encoded values from the user's submission, and it's complicated because Kitura stores the parsed body using an enum with an associated value.

The **values** constant we just set might be URL encoded values with an associated value set to a dictionary, or it might be JSON with an associated value to set a SwiftyJSON object, or it might be something else entirely. Swift has syntax that lets us check for a specific enum value while also unwrapping its associated value, but it's not commonly used so it might confuse you at first. Here it is:

```
guard case .urlEncoded(let body) = values else {
    try response.status(.badRequest).end()
    return
}
```

That means “check that **values** is set to the **.urlEncoded** value, and if so pull out its associated value into a constant called **body**. If that check fails, we again send back that it's a bad request and exit the route.

At this point, **body** is a dictionary contains the keys and values submitted to our route. In theory this should contain a “title” key, an “option1” key, and an “option2” key, but in practice the user might have missed some or all of them, or might have provided no text for them.

To avoid code duplication, we're going to write this final step in a loop. First, we'll create an array of the things we want to look for, like this:

```
let fields = ["title", "option1", "option2"]
```

We can then loop over that array, and check that each value exists in our **body** dictionary as a non-empty string. If any field *doesn't* exist, or exists but is just an empty string, then we'll send the same **.badRequest** status and exit. As each field is validated, we'll store it in a separate dictionary, so that we only insert values once they've had their whitespace removed.

Putting that into Swift, we get this:

```
var poll = [String: Any]()

for field in fields {
    if let value =
```

```

body[field]?.trimmingCharacters(in: .whitespacesAndNewlines) {
    if value.characters.count > 0 {
        poll[field] = value
        continue
    }
}

try response.status(.badRequest).end()
return
}

```

OK, that's the entire route broken down into smaller chunks, so it's time to piece it back together. Replace the existing “/polls/create” route with this:

```

router.post("/polls/create") {
    request, response, next in

    // 2: check we have some data submitted
    guard let values = request.body else {
        try response.status(.badRequest).end()
        return
    }

    // 3: attempt to pull out URL-encoded values from the
    // submission
    guard case .urlEncoded(let body) = values else {
        try response.status(.badRequest).end()
        return
    }

    // 4: create an array of fields to check
    let fields = ["title", "option1", "option2"]

    // this is where we'll store our trimmed values

```

```

var poll = [String: Any]()

for field in fields {
    // check that this field exists, and if it does remove
    // any whitespace
    if let value =
        body[field]?.trimmingCharacters(in: .whitespacesAndNewlines) {
            // make sure it has at least 1 character
            if value.characters.count > 0 {
                // add it to our list of parsed values
                poll[field] = value

                // important: this value exists, so go on to the
                next one
                continue
            }
        }
}

// this value does not exist, so send back an error and
exit
try response.status(.badRequest).end()
return
}

// all set!
}

```

If we get to the **// all set!** comment, it means we've successfully validated the user's input and we can go ahead and create a CouchDB document. Yes, after all that effort we're only half way there – we still need to actually store the data that was sent!

Fortunately, this is substantially less work: we already have a **poll** dictionary that contains the values the user wants to create, so we just need to add default values for the vote counts then convert it to JSON for CouchDB.

Creating a document is done using the `create()` method, which works like `retrieveAll()`: you pass it a closure to run when the operation completes, but it runs synchronously. If the document was created successfully we'll be given back its ID and revision number, plus the new document; otherwise, we'll be given an error. If we *do* get an error, then something has *really* gone wrong because we already validated the user's input. As a result, we'll be using the `.internalServerError` status (500) rather than `.badRequest` (400).

Put this code in place of the `// all set!` comment:

```
// fill in default values for the vote counts
poll["votes1"] = 0
poll["votes2"] = 0

// convert it to JSON, which is what CouchDB ingests
let json = JSON(poll)

database.create(json) { id, revision, doc, error in
    defer { next() }

    if let id = id {
        // document was created successfully; return it back to
        the user
        let status = ["status": "ok", "id": id]
        let result = ["result": status]
        let json = JSON(result)

        response.status(.OK).send(json: json)
    } else {
        // something went wrong – attempt to find out what
        let errorMessage = error?.localizedDescription ??
        "Unknown error"
        let status = ["status": "error", "message": errorMessage]
        let result = ["result": status]
```

```

        let json = JSON(result)

        // mark that this is a problem on our side, not the
        // client's
        response.status(.internalServerError).send(json: json)
    }
}

```

That's our second API endpoint completed: users can now create polls. To try it out, use a Curl command like this one:

```
curl -X POST localhost:8090/polls/create -d "title=iPhones in
gold: great color or greatest color?
&option1=Great&option2=Greatest"
```

Note that we don't need to specify the JSON content type any more because our API expects regular form data.

However, we're not quite done yet because of a cunning Kitura quirk. If you try the Curl example above, then use <http://localhost:8090/polls/list> to see what was created, you'll see your new poll looking good. However, we can also try creating a tiny slice of an example front-end in HTML, and things aren't quite so rosy there.

To see what I mean, create a file test.html on your desktop, giving it this content:

```
<form action="http://localhost:8090/polls/create" method="post"
accept-charset="UTF-8">
<p>Title: <input name="title" /></p>
<p>Option 1: <input name="option1" /></p>
<p>Option 2: <input name="option2" /></p>
<p><input type="submit" /></p>
</form>
```

**Note:** That's just a HTML fragment, so I've had to specify UTF-8 explicitly. Ideally your whole site is written with UTF-8 in mind.

Save that, then double-click it to open it in your web browser. This is a web form that will hit our “/polls/create” endpoint, and you can fill it in with real values to have them stored in CouchDB. It’s not served up through Kitura, but that doesn’t matter – this is just for testing.

What I’d like you to do is enter “What’s your favorite color?” for the title, then “Jet Black” and “Rose Gold” for the options. Finally, hit submit – all being well you should get some JSON back showing that the new document was created successfully.

However, things aren’t quite as happy as they might seem. If you go back to <http://localhost:8090/polls/list> you’ll see JSON similar to this:

```
{  
    "id" : "959ef5935064ed90f3f4ad35cf006f9a",  
    "title" : "What%27s+your+favorite+color%3F",  
    "option1" : "Jet+Black",  
    "option2" : "Rose+Gold",  
    "votes1" : 0,  
    "votes2" : 0  
}
```

Did you type “What%27s+your+favorite+color%3F”? Of course not. But that’s how it got *sent* because character’s like spaces, apostrophes and question marks aren’t safe in URLs and need to be encoded. So, the URL encoded version of a question mark is %3F, the URL encoded version of an apostrophe is %92, and so on. To be fair, we *are* using **.urlEncoded** for parsing the body, so this result is inevitable.

Kitura doesn’t have a built-in way to solve this, and, cunningly, neither does Swift. You see, the use of % numbers to encode reserved characters is common, albeit with some gnarly bits if you don’t use UTF-8, but the use of + in place of spaces is just a quirk of HTML. So, although Swift gives us a **removingPercentEncoding** property on strings, it doesn’t fix the + symbols.

I expect Kitura will add this at some point in the future, but until then we can write a simple function to do the clean up for us. Don’t worry about putting this in a separate file – just put it somewhere in main.swift for now:

```
extension String {
    func removingHTMLEncoding() -> String {
        let result = self.replacingOccurrences(of: "+", with: "")
    }
}
```

We can now modify our data handling code to this:

```
if value.characters.count > 0 {
    poll[field] = value.removingHTMLEncoding()
    continue
}
```

With that small change, our second API endpoint is done for real and the completion of this project is finally in sight.

## Casting a vote

CouchDB has a simple, almost naïve, approach to updating documents: you just read out the existing value, make any changes you want, then store the new data.

Earlier you saw how it uses the “rev” value for each document to automatically reject updates that potentially clash, and this is where it becomes important. Consider the following flow:

1. User A posts a vote to our API.
2. API requests the current document from CouchDB. It is revision 1.
3. User B posts a vote to our API.
4. API requests the current document from CouchDB. It's still revision 1.
5. API updates the vote count for user A and stores the result back in CouchDB. It is now revision 2.
6. API updates the vote count for user B and attempts to store the result back in CouchDB.
7. CouchDB rejects the update because they are trying to modify revision 1 – doing so would overwrite user A’s vote.

This is automatically handled for us, because when updating a document you must always provide both its ID and its latest revision number. If you provide an earlier revision CouchDB will reject the update, so we’ll pass that rejection onto the clients.

Making all this work requires two new CouchDB methods: `retrieve()` pulls out one document by its ID, and `update()` saves changes to a document. It’s quite a lot to digest all at once, so I’m going to split it in two: validating the user’s data then attempting to load the poll they requested, and updating the document.

The first part is mostly what you’ve seen before, with the addition of a call to `retrieve()`. Here’s what it’s going to do:

1. Validate that the user’s URL contained a poll ID and a vote option, otherwise send back `.badRequest` and end the closure.
2. Attempt to load the requested poll using `retrieve()`. This takes a document ID to load, as well as a closure to run when loading has finished.
3. If we got a document, update it. We’ll fill this part in later.

4. If we failed to load the poll in question, we'll send back `.notFound` (HTTP code 404).

The closure you pass to `retrieve()` must accept two parameters: the document that was returned and an error. You'll only receive one of the two depending on whether the request was successful.

Replace the current “/polls/vote/:pollid/:option” route with this:

```
router.post("/polls/vote/:pollid/:option") {  
    request, response, next in  
  
    // ensure both parameters have values  
    guard let poll = request.parameters["pollid"],  
        let option = request.parameters["option"] else {  
        try response.status(.badRequest).end()  
        return  
    }  
  
    // attempt to pull out the poll the user requested  
    database.retrieve(poll) { doc, error in  
        if let error = error {  
            // something went wrong!  
            let errorMessage = error.localizedDescription  
            let status = ["status": "error", "message":  
errorMessage]  
            let result = ["result": status]  
            let json = JSON(result)  
  
            response.status(.notFound).send(json: json)  
            next()  
        } else if let doc = doc {  
            // update the document  
        }  
    }  
}
```

```
    }
}
```

That's not too hard, at least as long as you're feeling OK with CouchDB so far. The second part of our code is trickier: assuming we pulled out a document, we need to update it, create some new JSON, then post it back to CouchDB and evaluate the result.

By the time our `// update the document` is reached, we definitely have a real CouchDB document representing a poll the user wants to vote on. This is a SwiftyJSON object, so we're going to take a copy of it, add one to whichever vote option the user requested, then save it back to CouchDB.

Some parts of that are easy enough. For example, we need to know the ID and revision of the document in order to make the call to `update()`, so we can get them straight from the JSON like this:

```
let id = doc["_id"].stringValue
let rev = doc["_rev"].stringValue
```

Casting the vote is straightforward, because SwiftyJSON lets us modify values directly inside its JSON. The `intValue` property guarantees a number will be returned, because it will be 0 if nothing was found. That same property can be used to write integer values back to the JSON, so we can cast a vote like this:

```
if option == "1" {
    newDocument["votes1"].intValue += 1
} else if option == "2" {
    newDocument["votes2"].intValue += 1
}
```

All that leaves is updating the document in CouchDB using `update()`. This takes an ID and revision number as its first two parameters, which we've already pulled out into `id` and `rev` constants. The third parameter should be a SwiftyJSON object containing the full document that should be put in place of the existing one – and I should emphasize that it's the *full* document, not just the bits you want to change.

As with `retrieveAll()`, `retrieve()`, and `create()`, this method accepts a closure that runs synchronously when the operation completes. You'll be given three values: the new revision number, the updated document JSON, and any error that occurred. Regardless of what comes back we're just going to send some status JSON and a HTTP status code like we've done previously, although this time we'll send back `.conflict` if the update failed. That tells our clients the update failed because someone else got there first, so they can choose to try again or show a message depending on what they want.

OK, that's everything – put this code in place of the `// update the document` comment in your code:

```
var newDocument = doc
let id = doc["_id"].stringValue
let rev = doc["_rev"].stringValue

if option == "1" {
    newDocument["votes1"].intValue += 1
} else if option == "2" {
    newDocument["votes2"].intValue += 1
}

database.update(id, rev: rev, document: newDocument) { rev,
doc, error in
    defer { next() }

    if let error = error {
        let status = ["status": "error"]
        let result = ["result": status]
        let json = JSON(result)

        response.status(.conflict).send(json: json)
    } else {
        let status = ["status": "ok"]
        let result = ["result": status]
```

```
        let json = JSON(result)

        response.status(.OK).send(json: json)
    }
}
```

To test out this route, you need to create a poll and note down the ID you get back from your app, then run a Curl command like this one:

```
curl -X POST localhost:8090/polls/vote/YOURIDHERE/1
```

That's the final route complete, so the project is complete. Of course, in reality this is only part of the problem: you now need to create front-ends for the API, such as an iOS app or a website. These will pull out the list of available polls then let users cast votes, or create new ones as needed.

## Wrap up

That's another massive project completed, but you're already in a place where you can make a variety of server-side projects. With routing, templates, and CouchDB in hand, you're starting to get to grips with three of the most important technologies for any server-side developer. There's still more to come, but you should still feel good that you've already tackled such massive topics head on.

Make no mistake: CouchDB is a complex piece of software. We spent quite some time looking at how it works using Curl – i.e., entirely outside Swift and Kitura – partly because it helps you understand exactly what is happening behind the scenes, but also because it's a highly transferrable skill. That is, if in the future you learn PHP, React, Rails, or another web platform, you'll find CouchDB works just as well there and your skills can be used immediately.

## Homework

There are three things you can try to help practice your Kitura. First, add a new “/polls/delete” API that accept a poll ID and deletes it from CouchDB. If you wanted to be really fancy you could add a simple “password” field that gets auto-generated and returned when a poll is created, so that only users who created a poll can delete it.

Second, see if you can convert the “/polls/vote/:pollid/:option” route so that it’s “/polls/vote/:pollid” where the “option” value is sent as a URL encoded value.

Finally, use the skills you learned in project 1 to make a web front-end for this polling API. You've seen how to create a HTML form to create polls, so now you just need a page to list them and a page to vote. Something like this ought to let users choose one option from the poll:

```
<input type="radio" name="option" value="1"> Jet Black</
input></p>
<input type="radio" name="option" value="2"> Rose Gold</
input></p>
```

# **Chapter 3**

## Routing

## Setting up

Every third project is a *technique* project, where I'm going to discuss a particular approach or API in more detail. In this first technique project we're going to deep dive into one of the most important fundamental parts of Kitura: routing.

We already used the Kitura router in our first two projects, but it really lies at the way we interact with the whole framework. As a result, it should be no surprise that we'll be using the Kitura router in all the other projects too, so it really is worth taking the time to understand how it works and how much work it can do for you.

To try things out, we're going to create a new project using the Swift package manager, add a variety of routes, and test them using Curl. So, use “swift package init” to create a new project called “project3”, then modify its Package.swift dependencies to include Kitura and Helium Logger – we don't need Stencil, CouchDB, or others, just Kitura and Helium Logger.

Finally, add this base content to main.swift:

```
import HeliumLogger
import Kitura

HeliumLogger.use()
let router = Router()

Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()
```

Run “swift build” now so that it fetches and builds all the dependencies, but while that's happening we can continue writing code...

## Chaining routes

Let's start with a nice and simple route. Place this before the call to `addHTTPServer()`:

```
router.get("/hello") {
    request, response, next in
    defer { next() }

    response.send("Hello")
}
```

That prints “Hello” when someone visits “/hello” with a GET request, which we can test using Curl. Something like this ought to do the trick: “curl -vX GET <http://localhost:8090/hello>”. I added the “v” parameter to trigger verbose mode, so you'll see everything Curl sends as well as the full response from Kitura.

In Kitura's list of responses, you'll see “HTTP/1.1 200 OK”. We didn't actually include `response.status(.OK)` in our route closure, but we didn't need to – as long as you send some content and call `next()` or `end()`, that status code will be sent automatically.

When `next()` is called, handling proceeds to the next route in line. So, we could place this route after the previous one to get “Hello, world!” printed:

```
router.get("/hello") {
    request, response, next in
    defer { next() }
    response.send(", world")
}
```

This is all old news, but now imagine some manager comes along and says, “/hello totally the wrong route now that we've added this new ‘world’ feature; make it /helloworld instead.” That means changing both instances of “/hello” to be “/helloworld”, which is hardly an epic amount of work. But what if you have five “/hello” routes? The last thing you want to do is change some but not all, so in some ways Kitura's flexible route handling can cause more harm than good.

Or at least it *would* do if route chaining didn't exist. This lets you split your routing across multiple closures, which each segment in the chain executing only if the previous one called **next()**. Best of all, you specify the path only once, so there's no risk of duplication or missing changes.

Route chaining is possible because router methods like **get()**, **post()** and **all()** are *variadic*. This means they accept one or more closures to run, and Kitura automatically stops the chain if **next()** isn't called.

Using this approach we can rewrite the previous code to this:

```
router.get("/hello", handler: {
    request, response, next in
    defer { next() }
    response.send("Hello")
}, {
    request, response, next in
    defer { next() }
    response.send(", world")
})
```

Thanks to route chaining, both messages are printed without needing to repeat the path.

Route chaining works well when you have several pieces of code you can chain together for the same GET request, but if you want work across multiple HTTP methods Kitura has a different approach: the **route()** method. This lets you attach multiple methods to a single path by chaining them as needed.

To try this out, add this new route to main.swift:

```
router.route("/test")
.get() {
    request, response, next in
    defer { next() }
    response.send("You used GET!")
}.post() {
```

```
request, response, next in
defer { next() }
response.send("You used POST!")
}
```

That attaches two different messages to “/test” – the one you see depends on what HTTP method you use. To see it in action, try these two Curl commands: **curl -vx GET http://localhost:8090/test** and **curl -vx POST http://localhost:8090/test**.

Behind the scenes, **route()** creates what’s called a sub-router – a router responsible for only part of your app’s routing. We’ll be looking at these in more detail in project 11.

So, if you want to group requests for the same path with the same method you use the variadic approach, whereas if you want to group requests for the same path across different methods you use **route()**. But that’s only the start of Kitura’s power...

## Reading user data

There are four primary ways of reading user data using the Kitura router, and it's worth summarizing them in one place so you have an easy reference.

The options are:

1. URL parameters declared in the route, such as “/users/:name”.
2. Reading URL-encoded form parameters that were submitted using POST, using **BodyParser**.
3. Reading URL-encoded parameters that were passed using GET.
4. Reading JSON.

We've used the first two already, so let's just recap them briefly.

First, URL parameters declared in the route. These are created by writing part of your URL after a colon, such as “/games/:name”. This makes each parameter available in the **request.parameters** dictionary, which you can check and unwrap as needed like this:

```
router.get("/games/:name") {  
    request, response, next in  
    defer { next() }  
  
    guard let name = request.parameters["name"] else { return }  
    response.send("Load the \(name) game")  
}
```

Second, URL-encoded form parameters. These are the most common when working with websites, but are complex in Kitura because so many parts may or may not exist. It's done in two steps, starting with adding the **BodyParser()** middleware to your route, like this:

```
router.post("/employees/add", middleware: BodyParser())
```

The second step is reading the value that got submitted, which is done in three smaller steps: ensuring the request has a body, pulling out its URL-encoded parameters, then finding the exact value you're looking for. Here's that in code:

```

router.post("/employees/add") {
    request, response, next in

    guard let values = request.body else {
        try response.status(.badRequest).end()
        return
    }

    guard case .urlEncoded(let body) = values else {
        try response.status(.badRequest).end()
        return
    }

    guard let name = body["name"] else { return }

    response.send("Adding new employee named \(name)")
    next()
}

```

Remember, forms submitted over HTML need to be decoded to remove lots of % and + symbols from your data. This extension does the job nicely:

```

extension String {
    func removingHTMLEncoding() -> String {
        let result = self.replacingOccurrences(of: "+", with: " ")
        return result.removingPercentEncoding ?? result
    }
}

```

That's two of our four options covered, so let's look at two new ones. Option three is reading URL-encoded parameters that were passed using GET, which in a URL looks like this: <http://localhost:8090/platforms?name=iOS>. This is *significantly* easier than working with POST values – here's a complete example:

```

router.get("/platforms") {
    request, response, next in

    guard let name = request.queryParameters["name"] else {
        try response.status(.badRequest).end()
        return
    }

    response.send("Loading the \(name) platform")
    next()
}

```

**Note:** You'll need to use the `removingHTMLEncoding()` extension here too.

Given how much easier GET is compared to POST, you might wonder why we went to so much trouble earlier – why not just use GET for everything?

There are several reasons, not least:

1. When a form is submitted using POST it is designed to modify data. Web browsers understand this, so if the user hits refresh they'll see a warning like “Are you sure you want to submit this form again?” That doesn’t happen with GET, which is designed to *read* data.
2. When a form is submitted using GET, the values are visible in the web browser’s URL bar, making it easy for users to monkey around with them. Using POST the values can still be modified but it takes some knowledge – users can’t just click and type any more.
3. GET requests are limited in length. The number isn’t strictly defined because every browser and server is different, but suffice to say that if you’re intending to submit lots of text – more than about 2000 characters – GET is a bad idea.
4. Some things that GET requests are used for are better expressed using URL parameters. These are called “clean URLs” because they look better to users, are easier to type, and are more easily understood by search engines.

Broadly speaking, URL parameters should be your first choice, POST your second choice, and GET your third.

The final way to submit values is using JSON. This is very similar to using URL-encoded values over POST, although the end result is a SwiftyJSON object rather than a regular dictionary.

So, to get started add some **BodyParser** middleware for your route:

```
router.post("/messages/create", middleware: BodyParser())
```

You then add your regular route closure, this time look for **case .json**, like this:

```
router.post("/messages/create") {
    request, response, next in

    guard let values = request.body else {
        try response.status(.badRequest).end()
        return
    }

    guard case .json(let body) = values else {
        try response.status(.badRequest).end()
        return
    }

    if let title = body["title"].string {
        response.send("Adding new message with the title \
(title)")
    } else {
        response.send("You need to provide a title.")
    }

    next()
}
```

Notice that **body** is a SwiftyJSON object now, so we need to extract values using properties such as **string**. Previously we used **stringValue**, which guarantees a string will be returned by providing an empty string if the requested name couldn't be found. **string** returns an *optional* string because it will return nil if **title** wasn't found, so we need to use **if let** to unwrap it safely.

You can test the JSON route with this Curl command: `curl -vX POST -H "content-type: application/json" localhost:8090/messages/create -d '{"title": "Swift: great language or greatest language?"}'`.

That's a whirlwind tour of four different ways you can read user data. Kitura's **Router** class does a fair amount of work for us, but I suspect over time they'll look for ways to make POST easier to use.

# Routing regular expressions

Regular expressions – or regexes for short – let you use special syntax to look for search patterns rather than specific characters. Although we haven’t taken advantage of this yet, Kitura actually uses regular expressions for all its route paths, which means if you mix regexes into all the other routing features we’ve seen so far.

This isn’t a book on regular expressions – for *that* you should buy my book and video course [Beyond Code](#) – but I can at least give you a brief primer:

- Use **[A-Za-z]** to match any single alphabet character, uppercase or lowercase.
- Use **[a-z]+** to match one or more alphabet characters, lowercase only.
- Use **[0-9]** to match any single digit from 0 to 9.
- Use **[0-9]{3}-[0-9]{4}** to match telephone numbers in the format 555-5555.
- Use **(news|reviews)** to match any either the exact word “news” or the exact word “reviews”.

By default, Kitura will match regex patterns without telling you what they contained. If you care what got matched you should use capture groups by adding parentheses around groups of things.

Here are three examples:

- The regex **[0-9]{3}-[0-9]{4}** will match numbers like 123-4567, but won’t tell you what the matching text was.
- The regex **([0-9]{3})-([0-9]{4})** will also match 123-4567, but will provide you with two variables: “123” and “4567”.
- The regex **([0-9]{3}-[0-9]{4})** will also match 123-4567, but will provide with a single variable: “123-4567”.

If you’re matching parts of a URL, such as “/foo/bar/baz”, you’ll probably want each part in its own capture group.

When capture groups are matched, they are placed into the **request.parameters** dictionary just like other URL parameters. You can access them using numbers referring to their position in the URL.

To demonstrate all that, here's code to match routes such as “/search/2016/twostraws”:

```
router.get("/search/([0-9]+)/([A-Za-z]+)") {
    request, response, next in
    defer { next() }

    guard let year = request.parameters["0"] else { return }
    guard let string = request.parameters["1"] else { return }

    response.send("You searched for \(string) in \(year)")
}
```

That's just a taster of what regular expressions can do, but the important thing is that they are built right into Kitura's router. This means you can combine a little bit router knowledge and a little bit of regex knowledge to make something awesome – all using Swift.

## Wrap up

Technique projects are here to make you think a little bit more closely about how you can approach your code, so with your new-found router knowledge you ought to be able to look back on the first two projects and find opportunities to expand them.

As you've seen, there are several ways you can receive data from users. Although there's clearly a preference order (with GET far from first place!) you'll find they are all used in practice. This means you need to be able to use them, or at least know that they *exist*, if you're serious about server-side Swift.

If there's one thing you remember from this chapter, it's regular expressions. They have always been available to us because they are baked right into Kitura's route handling, which means you start using them in any project – no extra libraries or code required. Regexes take up a large part of my book and video course, Beyond Code. If you're keen to learn more, start there: <https://www.hackingwithswift.com/store/beyond-code>.

# **Chapter 4**

## Swift Fan Club

## Setting up

In this project we're going to return to CouchDB to introduce querying, so you can learn how to find specific documents in your data. More importantly, though, you're going to learn how to authenticate users securely, and how to track logged-in users with sessions.

Of course, there's always a real app involved too, so in this project we're going to build a forum – a site that lets users post messages and replies to each other. This means creating routes to create forums and users, show the list of available forums, show the messages inside a forum, show the replies to a message, and also handle users logging in. There's lots to do, but because all the routes are coded independently it means we can take it all step-by-step.

Create a new Swift executable package called project4, then edit your Package.swift file to this:

```
import PackageDescription

let package = Package(
    name: "project4",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
            majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
CouchDB.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
StencilTemplateEngine.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
Session.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1)
    ]
)
```

As you can see, that adds Kitura-Session into to the mix, which lets us store small amounts of per-user data on the server.

In the project files that accompany this book, I'd like you to look in the project4-files directory for "public", "Views", and "Working" then copy them both into your project directory. The "public" directory contains the same Bootstrap code you've seen before, but Working is new – we'll come onto it later. In "Views" you'll find master.stencil, which is almost identical to the master.stencil file we used in project 1.

Go ahead and run **swift build** so that all the dependencies download fully. If you're using Xcode, generate your project now with **swift package generate-xcodeproj**. Whether you're using Xcode or not, open main.swift for editing and give it this skeleton content:

```
import CouchDB
import Cryptor
import Foundation
import HeliumLogger
import Kitura
import KituraNet
import KituraSession
import KituraStencil
import LoggerAPI
import SwiftyJSON

HeliumLogger.use()

let connectionProperties = ConnectionProperties(host:
"localhost", port: 5984, secured: false)
let client = CouchDBClient(connectionProperties:
connectionProperties)
let database = client.database("forum")

let router = Router()
router.setDefault(templateEngine: StencilTemplateEngine())
router.post("/", middleware: BodyParser())
router.all("/static", middleware: StaticFileServer())
```

```
Kitura.addHTTPServer(onPort: 8090, with: router)  
Kitura.run()
```

That includes a variety of frameworks, sets up a logger, connects to CouchDB, configures a router, and starts Kitura. The only new things in there are some of the imports: Cryptor lets us handle encryption, KituraNet gives us access to the standard collection of HTTP status codes, and KituraSession lets us read and write user sessions.

## Listing forums

So far all we've done is configure an empty Kitura app: it configures itself neatly, but has no routes and so won't serve anything useful. Our first step is to create a homepage that lists available forums, e.g "Announcements", "Help", and so on.

Later on we'll add functionality that lets users create forums, but for now we're going to inject initial content so we can get moving faster. In the Working directory you'll find two files: data.json and views.json. Open data.json in a text editor and you'll see it contains JSON-encoded documents ready for insertion into CouchDB. Here's a slice of what's in there:

```
{
  "docs": [
    {
      "type": "user",
      "_id": "twostraws",
      "salt": "b5c7....",
      "password": "a0e6..."
    },
    {
      "type": "user",
      "_id": "taylorswift13",
      "salt": "76bdf...",
      "password": "677e..."
    },
    {
      "_id": "songs",
      "type": "forum",
      "name": "Taylor's songs"
    },
    {
      "_id": "example_message_1",
      "type": "message",
      "forum": "songs",
      "user": "twostraws",
    }
  ]
}
```

```

        "date": "2016-12-01T14:00:00Z",
        "title": "What's your favorite song?",
        "body": "Mine is Shake it Off.",
        "parent": ""
    }
]
}

```

There are five important things to note in there:

1. Users, forums, and messages are all in the same list of documents.
2. Each document has a "type" property that distinguishes it as a user, forum, or message.
3. I'm forcing document IDs by specifying a precise "\_id" value. This is helpful when injecting content because it lets me dictate a precise structure, e.g. that user A posted message B to forum C.
4. Dates are written as strings using ISO 8601 format: "2016-12-01T14:00:00Z". The "T" separates the time from the date, and the "Z" means Zulu time, aka UTC or GMT.
5. Every user has long, seemingly random values for password and salt. In this test data, the actual password for each user is the same as their username, but storing passwords is both complicated and risky – more on that later.

All that data is just there to help us bootstrap our app – to let us write and test part of the system before another part is finished.

Fire up CouchDB (“/etc/init.d/couchdb start” on Docker, or launch CouchDB on macOS), then run this command in a terminal to create our database:

```
curl -X PUT localhost:5984/forum
```

With the database in place, we can bulk import our data.json documents with a single Curl command. First, use “cd Working” to change into the “Working” directory, then run this:

```
curl -d @data.json -H "Content-type: application/json" -X POST
localhost:5984/forum/_bulk_docs
```

You'll get back a list of IDs and revision numbers for the new documents, but check everything imported successfully run this command:

```
curl "localhost:5984/forum/_all_docs?include_docs=true"
```

We're done with data.json, so next I'd like you to open up views.json. This is where things get complicated, so brace yourself!

In project 2, we used **retrieve()** to pull out a specific document by its ID, **retrieveAll()** to pull out all documents in a database, **create()** to insert new documents, and **update()** to adjust the value of an existing document.

In this project our needs are more complex: we want to show a list of the available forums, so we can't just use **retrieveAll()** because that will return back forums, messages, and users. Later on, we're going to need to pull out a list of replies to a specific message, so again we can't just use **retrieveAll()** – we need a way to filter our data.

CouchDB's solution is views, which are effectively saved query results. You define your own views as code that decides whether or not a specific document should be included in the result, and that code gets run by CouchDB and its results stored so it won't be recomputed next time the view is used. When you create, update, or delete documents, CouchDB recalculates the results of your view for only the parts that have changed.

So far that probably doesn't sound too difficult, but there are two complexities just to make life interesting. First, these functions are written in JavaScript rather than Swift. That's an irritation, but fortunately not a major one – you really don't need much code in there. Second, you query your views using a "key" value, and the content of "key" depends on how you write your view functions.

Rather than try to explain more up front, I think it's easier just to look at some example view code. Open views.json and look for this code:

```
"forums": {
    "map" : "function(doc) { if (doc.type === 'forum')
{ emit(null, doc); } }"
}
```

JavaScript looks sort of like Swift, although it writes **function** rather than **func**, needs parentheses around **if** statements, and uses **==** for absolute comparison of values.

Let's break down what the code does:

1. It creates a new view called "forums". Views are queried by their name, so it's important to give them something useful.
2. It attaches a function to the view's "map" attribute. A mapping function is one that accepts some input and provides some converted output. In our case, it's a function that accepts a document and either adds it to the result or doesn't.
3. Our map function accepts a single parameter, **doc**, which will be one document from CouchDB. It will get called once for every document in the entire database.
4. You can evaluate that document however you want to by reading its properties. If you want to include it, call **emit()** and pass it a value to filter by (a "key"), and the data you want to attach (a "value"). If you don't call **emit()** for a document, it won't be in the result.
5. In our function, we check the **type** property of each document to make sure this view contains only forums.
6. We're emitting **null** for the key because we don't need to filter anything, and emitting **doc** for the value so that CouchDB provides the whole document.

So, when we request the "forums" view, CouchDB effectively runs that JavaScript function on all its documents and returns to you a list of the ones that matched. As well as a map function, you can also provide a reduce function that converts all your matching rows into a single value. For example you might use your map function to find all students in a physics class, then your reduce function to calculate their average grades.

Now that you know how views work, let's take a look at the rest of `views.json`. It contains three views in total, starting with the one we already looked at:

```
"forums": {  
    "map" : "function(doc) { if (doc.type === 'forum')  
    { emit(null, doc); } }"  
}
```

That defines a view called "forums", and adds to it any document with the type "forum".

Second, this one:

```
"forum_posts": {  
    "map" : "function(doc) { if (doc.type === 'message') { if  
(doc.parent === '') { emit(doc.forum, doc); } } }"  
}
```

That defines a view called "forum\_posts", and adds to it any document with the type "message" and an empty "parent" value. We'll be using "parent" to differentiate between top-level posts and their replies. This view uses **doc.forum** as the first parameter to **emit()**, which is where we're going to store the ID of the forum each message belongs to. Using **doc.forum** for the key like this means we can filter for messages that belong to a specific forum.

Finally, this third view:

```
"forum_replies": {  
    "map" : "function(doc) { if (doc.type === 'message') { if  
(doc.parent !== '') { emit(doc.parent, doc); } } }"  
}
```

That defines a view called "forum\_replies". This again filters on for "message" documents, but this time checks for a non-empty **parent** value. It also uses **doc.parent** as the key (the first parameter to **emit()**), which means we can find replies to a specific message by passing in the parent.

If you're new to CouchDB, and particularly if you're experienced with traditional SQL databases, this whole matter of emitting keys and values almost certainly seems bizarre. Give it time – once you see it all hooked up to Swift it will start to make more sense!

We need all three of those views for this project, so we're going to add them all using Curl. Run this command now:

```
curl -d @views.json -H "Content-type: application/json" -X PUT
localhost:5984/forum/_design/forum
```

That injects all of views.json into CouchDB, ready to use. This is what CouchDB calls a "design document" – a special document type that contains executable code rather than simple data. Using "\_design/forum" means "a design document called forum." Anyway, that's it for views and JavaScript, so we can write some code to query that view.

When the user visits our homepage, the "/" route, we need to query the "forums" view so we get back a list of forums the user can choose from. This is done using a method called **queryByView()**: give it the name of a view (we'll use "forums") and the name of your design document where the view is contained (that's "forum" for us) along with any query parameters, and you'll get back all the documents that match.

Let's write the first part of the "/" route now. Add this code before the call to **Kitura.addHTTPServer()**:

```
router.get="/" {
    request, response, next in

    database.queryByView("forums", ofDesign: "forum",
        usingParameters: []) { forums, error in
        defer { next() }

        if let error = error {
            // something went wrong
        } else if let forums = forums {
            // success!
        }
    }
}
```

The real work of that route where the real work needs to take place, but first look at the new method: **queryByView()**. The first time it's used, CouchDB will filter all its database

documents using the "forums" JavaScript map function we wrote, then return the list of matches. The results of the query are saved inside CouchDB, so using `queryByView()` again will just return the previous results – it's extremely efficient.

We have two comments to fill in: one when something went wrong, and one when everything worked. We'll be adding more advanced error handling in later projects, but for now we're just going to write a simple method that sends back a status code and some text. Add this method directly underneath the last `import` line, near the top of `main.swift`:

```
func send(error: String, code: HTTPStatusCode, to response:  
RouterResponse) {  
    _ = try? response.status(code).send(error).end()  
}
```

We're going to use that immediately. Replace the "// something went wrong" comment with this:

```
send(error: error.localizedDescription,  
code: .internalServerError, to: response)
```

So, if the query fails for some reason, that will print out the error and send an "internal server error" code.

The other comment is where we need to take action if the query was successful, which is more complicated. This needs to do three things:

1. Create a Stencil context that contains any basic values we want. This will be empty for now, but later we'll use it to add the user's username.
2. Convert the CouchDB documents into an array we can use in Stencil, and add that to the context object.
3. Pass the finished context to Stencil to render in a template.

First, the template we want to render into. This will build on `master.stencil`, just like we did with project 1. It uses the same `title` and `body` blocks, but this time the body is more complex because we need an conditional statement and a loop. If the user is logged in, we'll

print a welcome message then print the list of forums as web links; if not, we'll show a link telling them to log in.

Create a new template in Views called "home.stencil" and give it this content:

```
{% extends "master.stencil" %}

{% block title %}Home{% endblock %}

{% block body %}
<h1>Taylor Swift Fan Club</h1>
{% if username %}
<p>Welcome back, {{ username }}!</p>

<ul>
  {% for forum in forums %}
    <li><a href="/forum/{{ forum.value._id }}">{{ forum.value.name }}</a></li>
  {% endfor %}
</ul>

{% else %}
<a href="/users/login">Please log in.</a>
{% endif %}
{% endblock %}
```

Notice how the values for each forum – their ID and name – are contained inside the **value** property. If you try to read **forum.name** directly you'll see nothing at all, so be careful!

Next, the basic context for the template. When you build a site that requires users to log in, as ours will soon enough, it's common to save some information about them for quick access. For example, you might store their user ID or display name for quick access, rather than re-query CouchDB every time the page loads. If you use these values in your templates – for example, adding their username to the navigation bar on every page so they know they are logged in – then it's useful to write a function that creates this basic context in one central location.

Add this method below the `send(error:)` method we just wrote:

```
func context(for request: RouterRequest) -> [String: Any] {
    var result = [String: String]()
    result["username"] = "testing"
    return result
}
```

Yes, that just creates and returns a dictionary containing a dummy value, but it's enough to get us moving.

So, we have a template to render into, plus some basic context that we can send in. The last step is to merge that context with the results from CouchDB and send it off to Stencil.

Inside our `queryByView()` closure we have a `forum` value that will be set to the data we got back from CouchDB, and just like in project 2 the documents themselves will be stored inside the "rows" value of that data. However, that document data will all be SwiftlyJSON objects, which is great for Swift but not so much for Stencil. Fortunately, we can convert all those rows into an array of Swift dictionaries with a tiny amount of code:

`forums["rows"].arrayObject` is all it takes.

That's everything: we have a basic context, we can add the CouchDB documents to it, and we have a template to render the whole thing. Replace the `// success!` comment with this:

```
var forumContext = context(for: request)
forumContext["forums"] = forums["rows"].arrayObject

_ = try? response.render("home", context: forumContext)
```

If you build and run the code now, you should be able to navigate to <http://localhost:8090/> in your browser to see the home page. Right now, this will say "Welcome back, testing!" because we're forcing a username in there, but it will also list the two forums we added to CouchDB by hand.

Now, you might look at that and feel discouraged – after all, you just had to work through over

2000 words just to see a heading and two links. Chin up: you've learned how views work, which is by far the hardest thing in this project!

## Querying with a key

Our "/" route uses the "forums" view to read all the forum documents in our database, but it *doesn't* do any filtering – we just show all the forums that exist. The next step in this project is to create the "/forum/:forumid" path, which needs to show all top-level posts from a specific forum. This will use the "forum\_posts" view, which *does* need to be filtered because we only want to show messages that belong to the forum that was chosen.

If you were confused by the `emit()` function and its key/value approach to filtering, it should all become clear now. When handling the "/forum/:forumid" route, we can read the `:forumid` request parameter to figure out what forum to show. That will be a CouchDB ID, which usually is a long sequence of letters and numbers, but for our sample data is "songs" and "videos".

What we want to do is take that forum ID and ask CouchDB for all the top-level posts that belong to that forum. A "top-level" post is an original message, rather than a reply to an existing message. We have a view just for this job, "forum\_posts", which contains this mapping function:

```
function(doc) {  
    // only match message documents  
    if (doc.type === 'message') {  
        // only match messages without a parent  
        if (doc.parent === '') {  
            // add this document to the result, using its forum ID  
            // as its key  
            emit(doc.forum, doc);  
        }  
    }  
}
```

I spaced it out and added comments to make it easier to read.

And here's an example message, taken from data.json:

```
{
```

```

    "_id": "example_message_1",
    "type": "message",
    "forum": "songs",
    "user": "twostraws",
    "date": "2016-12-01T14:00:00Z",
    "title": "What's your favorite song?",
    "body": "Mine is Shake it Off.",
    "parent": ""
}

}

```

You can see the "forum" value right there – this message belongs to the "songs" forum. When the "forum\_posts" view is being built by CouchDB, it groups all the "message" documents by their key, which we've set to be **doc.forum** – it will group messages by their forum. All the messages are still there, mind you, but they are neatly organized by forum ID so we can find what we're looking for.

When we used **queryByView()** previously, we passed an empty array to the **usingParameters** parameter. That results in all matching documents being returned, which was fine before but not enough here: we need only messages that belong to the forum in question. This is done using keys: the view has organized all the messages by their "forum" property, so we can now pass in a specific forum ID to have only that subset of documents returned.

This is done a little clumsily in Kitura-CouchDB, so bear with me. **usingParameters** is an array of options that customize how you want the results. One of the options is **.keys**, which is an enum with an array associated value containing the keys you want returned. Cunningly, the definition of the array inside **.keys** is different on macOS on Linux, and the only way to make code compile cleanly is to typecast each key as a **Database.KeyType**.

So:

1. We pass an array of parameters.
2. We'll be using **.keys** to pull out a specific key.
3. You need to put an array inside **.keys** that contains the keys you want.
4. Each element inside that array can be any object, but it needs to be typecasted as

## **Database.KeyType**

5. Yes, it was actually designed to work this way.

To make things more interesting, we're going to add a second element to the **usingParameters** array: **.descending(true)**. This will return documents in reverse order, although in practice the only way to be *really* sure you're getting reverse order is to do it in your own code.

Anyway, we're going to get the current forum ID into a constant called **forumID**, and once that's done here's how the call to **queryByView()** is going to look:

```
database.queryByView("forum_posts", ofType: "forum",
    usingParameters: [.keys([forumID as
        Database.KeyType]), .descending(true)]) { messages, error in
    // code here
}
```

Putting it all together, the "forum\_posts" view contains a map function that groups all the messages by the forum they belong to, and our call to **queryByView()** says "give us the ones that belong to forum (X)."

There's one more thing to talk about before we dive into code. It's not complicated, but it does make our life difficult. As you've seen, Kitura-CouchDB is all closure-based: you run a method, then pass it a closure to run when the method completes. This works well when you have simple queries, but as your complexity mounts these closures can really get in the way. Later in the book I'll be helping you write wrappers around these closures to make your code easier, but while you're still learning it's best to do it the "official" way, so be prepared for closures inside closures inside closures!

OK, time for the new route. Here's what it needs to do:

1. Ensure that the ":forumid" value was set in the route.
2. Fetch the forum with that ID from CouchDB so we have its name.
3. If that failed, print an error and stop.
4. Otherwise, fetch the top-level messages in that forum.

5. Again, if that failed print an error and stop.
6. Otherwise, convert the CouchDB JSON to a Swift array ready for Stencil, then add it to the default context along with the name and ID of the current forum.
7. Render a new "forum.stencil" template with that context.

Here's the code – put this below the "/" route we added previously:

```
router.get("/forum/:forumid") {
    request, response, next in

    guard let forumID = request.parameters["forumid"] else {
        send(error: "Missing forum ID", code: .badRequest, to:
response)
        return
    }

    database.retrieve(forumID) { forum, error in
        if let error = error {
            send(error: error.localizedDescription,
code: .notFound, to: response)
        } else if let forum = forum {
            database.queryByView("forum_posts", ofDesign: "forum",
usingParameters: [.keys([forumID as
Database.KeyType]), .descending(true)]) { messages, error in
                defer { next() }

                if let error = error {
                    send(error: error.localizedDescription,
code: .internalServerError, to: response)
                } else if let messages = messages {
                    var pageContext = context(for: request)
                    pageContext["forum_id"] =
forum["_id"].stringValue
                    pageContext["forum_name"] =

```

```
forum[ "name" ].stringValue  
        pageContext[ "messages" ] =  
messages[ "rows" ].arrayObject  
  
        _ = try? response.render( "forum" , context:  
pageContext)  
    }  
}  
}  
}  
}  
}
```

That won't work yet because we haven't created the `forum.stencil` template, so let's do that now. This needs to inherit from `master.stencil`, and provide a **body** block that a) shows the forum name, b) loops over all the messages and prints them out, and c) adds a form to the end to let users write new posts.

To make things a bit neater, we're going to use an unordered list for all the messages. This is a **<ul>** element for the whole list, with individual **<li>** elements each list item. Forms are created using a **<form>** tag to mark the start and end of all the fields in the form, then whatever content you want inside. We'll be using one **input** to capture the message title, and a **textarea** to provide free text entry for the message body. We're going to attach the CSS class "form-control" to both of those form fields, which adds Bootstrap styling to them.

In order to submit the form – to send it somewhere for processing – we need to add a button for the user to click. This is done using HTML's `<button>` element along with the `type="submit"` attribute, but we're going to add some styling here too – the CSS class names `btn` and `btn-lg` will make a large button for us thanks to Bootstrap.

That's enough code for now, so create a new template file in Views called `forum.stencil`, and give it this content:

```
{% extends "master.stencil" %}

{% block title %}{% forum name %}{% endblock %}
```

```

{%
    block body %
}

<h1>{{ forum_name }}</h1>

{%
    if messages %
}
<ul>
{%
    for message in messages %
}
<li><a href="/forum/{{ forum_id }}/{{ message.id }}"/>{{ message.value.title }}</a> – posted by
{{ message.value.user}} on {{ message.value.date }}</li>
{%
    endfor %
}
</ul>
{%
    endif %
}

<form method="post">
<h3>Add a new post</h3>
<p><input name="title" type="text" class="form-control"
placeholder="Enter a title" /></p>
<p><textarea name="body" class="form-control" rows="5"></
textarea></p>
<p><button type="submit" class="btn btn-lg">Post</button></p>

{%
    endblock %
}
```

Build and run your project, then go to your web browser and click one of the forums. You should see messages lined up like this:

- What are the lyrics to Blank Space? – posted by adele on 2016-12-02T11:00:00Z
- What's your favorite song? – posted by twostraws on 2016-12-01T14:00:00Z

So, it definitely *works*, but I think the output is... well, let's be generous and call it *suboptimal*. Unless you're a Vulcan, you're probably not going to enjoy reading dates as "2016-12-01T14:00:00Z", so we want to do a little cleaning of our data.

There are two ways we can do our cleaning: we can either manipulate the array we got back from CouchDB *before* we pass it to Stencil, or we can manipulate it *inside* Stencil. The latter option is preferable here, because after all converting the display format of dates is presentation logic – it's the kind of thing you want to keep in your template files.

However, at the same time we don't want complex logic in our templates, and date parsing certainly *is* complex. So, what we're going to do is create our own Stencil filter. If you remember, we used the **capitalize** filter in project 1 to turn "kirk" into "Kirk", like this:

```
<h1>{{ name|capitalize }}</h1>
```

We're going to write a new filter called **format\_date**, that will convert any ISO 8601-style date into something nicer.

You've already seen that we attach a template engine directly to the Kitura router, like this:

```
router.setDefault(templateEngine: StencilTemplateEngine())
```

Well, for the first time we're going to pass a parameter to the **StencilTemplateEngine** initializer: a namespace containing our customer filter. In Stencil, a “namespace” is responsible for storing all the filters and tags that are available to a template while it's being rendered. The default namespace is what gives us things like **capitalize** and **{% for %}**, but we can create a new one to add our our custom features onto the basic set.

To start, add an **import** line for Stencil so we can use its **Namespace** class:

```
import Stencil
```

Now look for the **router.setDefault()** line in main.swift, and change it to this:

```
let namespace = Namespace()

// custom filters here!

router.setDefault(templateEngine:
StencilTemplateEngine(namespace: namespace))
```

That creates an empty namespace, and passes it into the **StencilTemplateEngine** initializer so it gets used.

The real work is to replace the "// custom filters here!" comment with our date formatting filter. To make this work we're going to use a new data type called **DateFormatter**, which is able to convert strings to dates and dates to strings. There are two ways of using it, and we'll need both: you can specify a precise date format, or you can use one of the built-in format types. We'll use a precise date format coming in because we need to match dates in the format 2016-12-01T14:00:00Z, but we'll use built-in formats for output because it's easier and more flexible.

Here's an example before we dive in to the real code:

```
let formatter = DateFormatter()
formatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZ"
let date = formatter.date(from: someString)
```

That creates a formatter, tells it to work with the date format we need, then converts **someString** into a date. There are two complexities in there, of which the most obvious one is the date format: "yyyy-MM-dd'T'HH:mm:ssZ". That mixes uppercase and lowercase letters, dashes and colons, and more. Each of those letter segments has meaning, and its case matters:

- "yyyy" looks for a four-digit date.
- "MM" looks for a month in numerical format. ("MMM" looks for "Jan", and "MMMM" looks for "January")
- "dd" looks for "day of the month" ("DD" looks for day of the year)
- "T" looks for the exact character "T"
- "HH" looks for hours with zero padding, e.g. 09 ("H" looks for hours without zero padding, e.g. "9")
- "mm" looks for minutes with zero padding, e.g. 05 ("m" looks for minutes without zero padding)
- "ss" looks for seconds with zero padding, e.g. 07 ("s" looks for seconds without zero padding)

- "Z" looks for a timezone, which in our case is "Zulu time" – also Z

So, that's the first complexity: specifying how your dates are formatted is hard. And no, it never really gets any easier. As I said, the **DateFormatter** type has a more convenient option, but first I want to mention the second complexity of our previous code. It's here:

```
let date = formatter.date(from: someString)
```

When that code runs you might think that **date** contains a valid date, but it actually contains **Date?** – an optional date. This is because it's possible **someString** contained some text that doesn't match the format string we specified, such as "wombat". So, **date(from:)** will return a valid date if it can, otherwise it will return nil. This means we need to check and unwrap the result using **if let**.

OK, onto the convenient way to use **DateFormatter**: its **dateStyle** and **timeStyle** properties. These use built-in ways to format date and time based on your region and how much detail you want. For example, we'll be using code like this:

```
formatter.dateStyle = .long
formatter.timeStyle = .medium
let string = formatter.string(from: someDate)
```

That requests a long date ("December 2, 2016") with a medium time ("2:00:00 PM"), then places the result into a string. Unlike **date(from:)**, **string(from:)** always returns a value – it won't return nil, because conversion will always succeed. It's important to remember you have *no* control over the value that comes out: it's decided by **DateFormatter**, which could in theory change at any point in the future, and also takes your region into account. So, if your server is configured for French you'll get different results from a server configured for Italian.

That's all you need to know about date formatting, so go ahead and add this code after the **let namespace = Namespace()** line:

```
namespace.registerFilter("format_date") { (value: Any?) in
    if let value = value as? String {
```

```

let formatter = DateFormatter()
formatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZ"

if let date = formatter.date(from: value) {
    formatter.dateStyle = .long
    formatter.timeStyle = .medium
    return formatter.string(from: date)
}

return value
}

```

Notice how it starts with an **if let**? This is because the data type coming in is **Any?** – literally any kind of data, or nothing at all. The "nothing at all" part is because the value passed into the filter might not have been set in the template context; the "any kind of data" part is there because you can use filters on arrays, dictionaries, integers, and more.

In the case of our date formatting filter, we obviously can't handle formatting anything other than strings, so the first thing our filter does is attempt to typecast the input to a string. If that fails – or if the string to date conversion fails – then we always return the original input value.

Save your Swift, modify `forum.stencil` so that it uses `{ { message.value.date | format_date }` for the date value, then give it a try. All being well you should see messages listed like this:

- What are the lyrics to Blank Space? – posted by adele on December 2, 2016 at 11:00:00 AM
- What's your favorite song? – posted by twostraws on December 1, 2016 at 2:00:00 PM

Much better!

## Closures within closures

The next route we're going to implement is "/forum/:forumid/:messageid", which is used when a user clicks a message to read. This one is more complicated than you might think, but only because of the way Kitura-CouchDB is structured – all those closures really start to get onerous about now.

This route needs to:

1. Retrieve the forum ID contained in the `:forumid` request parameter. This tells us the forum a message was posted to.
2. Retrieve the message contained in the `:messageid` request parameter. This gives us the text of the message in query.
3. Use `queryByView()` to pull out all the replies to the message in question.

The problem is that all three of those steps use closures, so you end up with a pyramid of closures. We'll look at ways around this later on in the book, but for now we're just going to stick with the pyramid.

Let's start with the easy stuff:

1. If the "forumid" and "messageid" request parameters don't exist, we can exit immediately because this is an invalid request.
2. We can use the value from "forumid" to load the current forum. Again, if that doesn't exist it's a fatal error.
3. We can use the value from "messageid" to load the selected message, and once more it's a fatal error if that doesn't exist.
4. Only if the previous three steps are executed successfully do we continue.

We can turn that into the start of our code – add this route to main.swift:

```
router.get("/forum/:forumid/:messageid") {  
    request, response, next in  
  
    guard let forumID = request.parameters["forumid"],  
        let messageID = request.parameters["messageid"] else {
```

```

        try response.status(.badRequest).end()
        return
    }

database.retrieve(forumID) { forum, error in
    if let error = error {
        send(error: error.localizedDescription,
code: .notFound, to: response)
    } else if let forum = forum {
        database.retrieve(messageID) { message, error in
            if let error = error {
                send(error: error.localizedDescription,
code: .notFound, to: response)
            } else if let message = message {
                // success!
            }
        }
    }
}
}

```

When we finally hit the `// success!` comment we can do the real work: find any replies, then create output context and render the page. This means using our third and final CouchDB view, "forum\_replies", which looks like this:

```

function(doc) {
    if (doc.type === 'message') {
        if (doc.parent !== '') {
            emit(doc.parent, doc);
        }
    }
}

```

That a) only returns documents of the "message" type, b) that have a "parent" value set, c)

grouped by the "parent" value. This means we can query it using the ID of our selected message, and get back all the replies to that message. This means repeating the ugly **Database.KeyType** dance from before, like this:

```
[.keys([messageID as Database.KeyType])]
```

If the query fails for any reason it's yet another error condition so exit. Otherwise, we're going to combine the forum information, the message information, and the replies array into a single Stencil context, and pass it off to a new template for rendering. The replies will be an array of messages so we need to use **arrayObject** again to get it into a format that's useful for Stencil, but the top-level message is just one dictionary of data so we'll use **dictionaryObject**.

Add this code in place of the **// success!** comment:

```
database.queryByView("forum_replies", ofDesign: "forum",
usingParameters: [.keys([messageID as Database.KeyType])])
{ replies, error in
    defer { next() }

    if let error = error {
        send(error: error.localizedDescription,
code: .internalServerError, to: response)
    } else if let replies = replies {
        var pageContext = context(for: request)
        pageContext["forum_id"] = forum["_id"].stringValue
        pageContext["forum_name"] = forum["name"].stringValue
        pageContext["message"] = message.dictionaryObject!
        pageContext["replies"] = replies["rows"].arrayObject

        _ = try? response.render("message", context: pageContext)
    }
}
```

Hopefully all that code will make sense to you at this point in your Kitura career, but I have to

admit: the use of closures for simple results is quite tedious. I'm hopeful this will go away if and when the Kitura-CouchDB code gets refactored.

The message.stencil template is easy enough: print out the message, then loop over the **replies** value and print out each one. We'll need to re-use the **format\_date** filter again in order to make the dates look good, and we're also going to add another HTML form to the end so people can post replies. Create message.stencil now, giving it this content:

```
{% extends "master.stencil" %}

{%
    block title
    {{ message.title }} - {{ forum_name }}
%}
endblock

{%
    block body
%}
<h1>{{ message.title }}</h1>
<h2>Posted by {{ message.user }} on {{ message.date | format_date }}</h2>

{{ message.body }}

{%
    if replies
%}
{%
    for reply in replies
%}
<h4>Reply from {{ reply.value.user }} on {{ reply.value.date | format_date }}</h4>
<p>{{ reply.value.body }}</p>
{%
    endfor
%}
{%
    endif
%}

<form method="post">
<h3>Add a new reply</h3>
<input type="hidden" name="title" value="Reply" />
<p><textarea name="body" class="form-control" rows="5"></textarea></p>
<p><input type="submit" class="btn btn-lg" /></p>
```

```
{% endblock %}
```

Notice that form has an input called "title" with the fixed value of "Reply". This uses the "hidden" type, which means it won't be visible to users, but it makes our form parsing a lot easier as you'll see later!

You should now be able to browse to <http://localhost:8090/>, click a forum, then click a message to read it and its replies. You can't *post* anything yet, but we'll be getting to that soon enough...

## Users and sessions

Before we're able to post messages and replies, we need something else first: we need to know *who* is posting. Right now we're injecting the "testing" username directly into the Stencil context, but that's just a stop-gap measure – we need to replace that with real user credentials, which in turn means learning about sessions and encryption.

First, sessions: these are temporary, individual data stores allocated for every user who comes to your site. When a session is started, a small identifier is sent back to your user's web browser and stored – known as a "cookie". This cookie contains a unique identifier that automatically gets sent back to the server with every subsequent page request. When Kitura receives the session cookie, it looks up the identifier in its session database and loads any data you've stored there. This approach not only means that you can store sensitive data in your sessions because they never leave your server, but also that every user is guaranteed to have their own data.

**Note:** Session cookies are temporary. When your user closes their web browser they are automatically deleted.

In this project, we're going to be using sessions to track logged in users. This means creating a new "/users/login" route that will display a login form then process the results. When a user has been authenticated successfully, we're going to add their username to their session, so we know exactly who is logged in and who isn't.

In order to make that work, you also need to learn a little about encryption. Storing user data is always a sensitive affair, particularly when storing *passwords*. Despite the best efforts of websites around the world, most users continue to rely on passwords that are shared across many sites. As a result, if one website has its user database hacked, it's often possible to use the hacked database to get into accounts on other sites with predictably catastrophic results.

As web developers, it's our job to store user passwords securely, and that's done using three techniques: hashing, salting, and rounds. You need to understand all three of these, but it only takes a few minutes so bear with me.

First, hashing. A *hash* is a bit like one-way encryption: you can calculate the hash value for any input, but you can't convert the has hash value into its original input. Hashes have a fixed

length no matter how much data you input, which means a hash of the text "Hello, world" will have the same length as the hash of a 25GB Blu-ray. Hashes are designed to be as unique as possible, which means the hash for "Hello, world" is very different to the hash of "Hello, World" even though the only difference is one capital letter.

Second, salting. If you calculate a hash value of "Hello, world" 100 times, you will get the same result 100 times – hash functions always produce the same output from the same input. This causes a problem, because a hacker could simply calculate the hash values for the 100,000 most popular passwords and do a simple string comparison to find which hashes match. A "salt" is random data that gets added to the input string in order to make the output unique. So rather than hashing "Hello, world" you hash "19229Hello, world". Salts aren't secret; they are just there to stop people from guessing hashed passwords easily.

Finally, rounds. A computer can calculate the hash of an input string in an infinitesimally small amount of time, so if your user database gets leaked it wouldn't take a hacker long to figure out many of the passwords just by hashing common passwords. To work around this, it's common to use rounds, which are effectively hashing loops. You calculate the hash of a user's password plus their salt, then you hash that hash, then hash *that* hash, and so on. It's common to use at least 100,000 rounds, and in fact we'll be using 250,000 rounds – our code will literally keep generating new values a quarter of a million times until it reaches the final result.

So, hashing lets us convert specific input into specific, fixed-length output, salting lets us avoid making passwords easily guessed, and rounds let us add a massive computational roadblock to the whole thing. Even with 250,000 rounds, a modern computer will still take less than a second to calculate the final hash for a user's password, but that's exponentially longer than before and that's what matters. It won't stop committed hackers, but it will massively slow them down.

Given the huge complexity, you might think it takes a lot of code to hash and salt passwords safely. Fortunately, we can wrap the whole thing up in a relatively small function, thanks to IBM's BlueCryptor framework. This comes bundled with Kitura-Session, so we can use it immediately.

Most of work is done using the **PBKDF.deriveKey()** method, which takes a string and salt, along with a hashing function, the number of rounds to perform, and how long we want

the result to be. We're going to be using the SHA-512 algorithm across 250,000 rounds, which was designed by the US National Security Agency and is currently the recommended standard hashing function. The output of `deriveKey()` is an array of integers, but BlueCryptor provides a utility method for converting that to a string so we can store it.

Add this method to main.swift:

```
func password(from str: String, salt: String) -> String {
    let key = PBKDF.deriveKey(fromPassword: str, salt: salt,
prf: .sha512, rounds: 250_000, derivedKeyLength: 64)
    return CryptoUtils.hexString(from: key)
}
```

With that in place, we're now able to create users and log them in. We already have a couple of test users that we injected straight into CouchDB, so let's start by creating a login screen.

There are three parts to this, two of which are trivial. First, we need to create a users-login.stencil template in the Views directory. This is just going to show a login form that submits a username and password using POST. Create that file now, and give it this content:

```
{% extends "master.stencil" %}

{% block title %}Login{% endblock %}

{% block body %}
<h1>Login</h1>
<form method="post">
<p>Username: <input type="text" name="username" /></p>
<p>Password: <input type="password" name="password" /></p>
<p><input type="submit" class="btn btn-lg" />
</form>
{% endblock %}
```

The second trivial addition is a "/users/login" route that shows the above form. Add this route to main.swift:

```

router.get("/users/login") {
    request, response, next in
    defer { next() }

    try response.render("users-login", context: [:])
}

```

The third part is more complex. It's also the "/users/login" route, but this time using the POST method – triggered when the user submits the form we just created. This needs to:

1. Extract the values that were submitted in the form, and ensure they are all present.
2. Fetch the user document from CouchDB that matches the user name they entered.
3. Use the **password(from:salt:)** method to convert the password they entered to the hashed version we saved.
4. Compare that result against the hash saved in the database.
5. If it succeeds, save their username in the session and redirect them to the homepage. Otherwise, bail out.

Now, if you remember from project 2, extracting form values and checking all exist takes a fair amount of code. Worse, this is a common thing to want to do, and we'll be doing it multiple times in this project as well as repeatedly in later projects.

To make our life a little easier we're going to create a method to do it for us. This method, **getPost()**, will accept a request along with a list of fields we expect to have, and will return those fields in a dictionary if they all exist, or nil if at least one of them does not exist or is empty. It will also fix up the HTML encoding that is present – all those percent symbols and + signs.

First, add the **removingHTMLEncoding()** string extension we used back in project 2, placing it before the **HeliumLogger.use()** line:

```

extension String {
    func removingHTMLEncoding() -> String {
        let result = self.replacingOccurrences(of: "+", with: " ")
        return result
    }
}

```

```

        ")
    return result.removingPercentEncoding ?? result
}
}
}

```

Second, add this method to main.swift, just below the **removingHTMLEncoding()** method:

```

func getPost(for request: RouterRequest, fields: [String]) ->
[String: String]? {
    guard let values = request.body else { return nil }
    guard case .urlEncoded(let body) = values else { return
nil }

    var result = [String: String]()

    for field in fields {
        if let value =
body[field]?.trimmingCharacters(in: .whitespacesAndNewlines) {
            if value.characters.count > 0 {
                result[field] = value.removingHTMLEncoding()
                continue
            }
        }
    }

    return nil
}

return result
}

```

That is almost identical to the code we used in project 2, although now it returns the finished dictionary.

With those two new methods in place, we can finally write the second "/users/login" route, this time to handle POST requests. I've already explained what it needs to do, but there are two new pieces of code you need to learn before we dive in to the full thing.

First, user session data belongs to the **request** object we get passed to our route closures. This is an optional SwiftyJSON object, so we can inject values directly into it. For example:

```
request.session?["username"].string = "some_username"
```

The session might not exist, because Kitura only starts its session handler if you ask for it. To do that, you need to create a new **Session** object with a secret key, and attach it to whatever routes need access to your session data. For this project – and indeed most projects – that's *all* routes, so please add this line of code to main.swift, directly after the line that attaches

**StaticFileServer**:

```
router.all(middleware: Session(secret: "The rain in Spain falls mainly on the Spaniards"))
```

The session secret is there to encrypt the session ID on the user's machine – it can be any string you want. Once you've started your session you can read and write any data you want.

The second new code deals with redirecting users, which is when you want to move them to a different URL from the one they requested. Our users log in through the "/users/login" route, but if login works we want to redirect them to the homepage so they see the list of forums.

In Kitura, redirecting is done through the **redirect()** method. This takes the new path as its parameter, and is a throwing function because it's possible (but hopefully unlikely!) the redirect might fail.

That's everything, so go ahead and add this to main.swift:

```
router.post("/users/login") {
    request, response, next in

    // ensure all the correct fields are present
    if let fields = getPost(for: request, fields: ["username",
        "password"]) {
        if validateUser(fields) {
            let session = Session(secret: "The rain in Spain falls mainly on the Spaniards")
            session.set("username", value: fields["username"])
            session.set("password", value: fields["password"])
            response.redirect(to: "/")
        } else {
            response.status(.badRequest)
            response.end()
        }
    } else {
        response.status(.badRequest)
        response.end()
    }
}
```

```

"password"]) {
    // load the user from CouchDB
    database.retrieve(fields["username"]!) { doc, error in
        defer { next() }

        if let error = error {
            // this user doesn't exist!
            send(error: "Unable to load user.",
            code: .badRequest, to: response)
        } else if let doc = doc {
            // load the salt and password from the dicument
            let savedSalt = doc["salt"].stringValue
            let savedPassword = doc["password"].stringValue

            // hash the user's input password with the saved
            salt; this should produce the same password we have saved
            let testPassword = password(from:
                fields["password"]!, salt: savedSalt)

            if testPassword == savedPassword {
                // the password was correct - save the username
                in the session and redirect to the homepage
                request.session!["username"].string =
                doc["_id"].string
                _ = try? response.redirect("/")
            } else {
                // wrong password!
                print("No match")
            }
        }
    }
} else {
    // the form was not filled in fully
    send(error: "Missing required fields", code: .badRequest,

```

```
        to: response)
    }
}
```

Before you try running the new code, we need to make one small change: we need to stop forcing a username in our base context. Find this code:

```
func context(for request: RouterRequest) -> [String: Any] {
    var result = [String: String]()
    result["username"] = "testing"
    return result
}
```

Replace the "testing" line with this:

```
result["username"] = request.session?["username"].string
```

That loads the user's *actual* username into the context based on reading their session. If they haven't logged in yet that will set `result["username"]` to be nil.

Go ahead and run the code now – you should see a log in message on the homepage, which takes you to the login form we made. If you use the username "twostraws" and password "twostraws" you should be able to log in successfully, at which point you'll be redirected back to the homepage and see our forum links – good job!

Now that we have code to log in users, it's only a short hop further to let people create accounts themselves. This has a similar three steps to logging in: there's a simple template to present some HTML, a GET "/users/create" route to handle loading that template, and a POST "/users/create" route to do all the hard work.

First, the template. Add this code to a `users-create.stencil` file in Views:

```
{% extends "master.stencil" %}

{% block title %}Create account{% endblock %}
```

```

{%
    block body %
}
<h1>Create account</h1>
<form method="post">
<p>Username: <input type="text" name="username" /></p>
<p>Password: <input type="password" name="password" /></p>
<p><input type="submit" class="btn btn-lg" />
</form>
{%
    endblock %
}
```

Now add this route to main.swift:

```

router.get("/users/create") {
    request, response, next in
    defer { next() }

    try response.render("users-create", context: [:])
}
```

So far, so boring. As before, it's the POST method that's more complex, because it has to create a new user document based on the form that just got submitted. This route will:

1. Use **getPost()** to ensure that "username" and "password" have valid values from the user's form.
2. Attempt to retrieve a user with the same ID as the one that was entered. If this succeeds it means the username is taken already so we can't continue.
3. If the username fetch fails it means the username is unique, so we can create a new user document.
4. If the new username couldn't be created successfully we'll send back an error.

Creating the new user document isn't as easy as it was in project 2. First, we need to force an ID rather than let CouchDB create one, because we're using the username itself for our IDs. This is done by specifying an "`_id`" key in the JSON we send to CouchDB. Second, we need to attach a "type" key with the value "user" so our views work correctly. Third, and most

important, we need to generate a random salt, use it to generate a hashed password for the user, then store both salt and password hash in our user document.

BlueCryptor comes with a method to generate random data suitable for a salt, but it's a throwing method meaning that it might fail. So, we're going to have a fallback: if we can use the BlueCryptor randomizer we will, but if that fails we'll create a hash by combining their username and password with a secret string. In code, it looks like this:

```
if let salt = try? Random.generate(byteCount: 64) {
    saltString = CryptoUtils.hexString(from: salt)
} else {
    saltString = (fields["username"]! + fields["password"]! +
"project4").digest(using: .sha512)
}
```

In theory the fallback should never be needed, but it's smart to cover all eventualities.

To make this a bit easier to follow, let's write the route in two parts. First, the easy part: if some fields are missing or the username exists already, we can bail out. Add this route to main.swift:

```
router.post("/users/create") {
    request, response, next in
    defer { next() }

    guard let fields = getPost(for: request, fields:
        ["username", "password"]) else {
        send(error: "Missing required fields", code: .badRequest,
        to: response)
        return
    }

    database.retrieve(fields["username"]!) { doc, error in
        if let error = error {
            // username doesn't exist!
```

```

    } else {
        // username exists already!
        send(error: "User already exists", code: .badRequest,
to: response)
    }
}
}
}

```

The **// username doesn't exist** comment is where the real work takes place: generating a salt, then creating the new user. I've added some comments below to help simplify it a little – add this comment where the **// username doesn't exist** comment is:

```

var newUser = [String: String]()

// force the CouchDB ID to be the username
newUser["_id"] = fields["username"]!

// add a "type" property so our views can filter correctly
newUser["type"] = "user"

let saltString: String

// create a salt using one of two methods
if let salt = try? Random.generate(byteCount: 64) {
    saltString = CryptoUtils.hexString(from: salt)
} else {
    // this in theory ought never to be used – it's an emergency
    fallback!
    saltString = (fields["username"]! + fields["password"]! +
"project4").digest(using: .sha512)
}

// we need to store the salt in the database so we can re-hash
the password at login time

```

```

newUser["salt"] = saltString

// calculate the password hash for this user
newUser["password"] = password(from: fields["password"]!, salt:
saltString)

let newUserJSON = JSON(newUser)

// send it off to CouchDB
database.create(newUserJSON) { id, revision, doc, error in
    defer { next() }

    if let doc = doc {
        // send back a message that everything worked
        response.send("OK!")
    } else {
        // error
        send(error: "User could not be created",
code: .internalServerError, to: response)
    }
}

```

If everything works and the user is created, that route just sends back "OK!" – hardly useful. What we really want is to log the new user in and redirect them back to the homepage. That smells like a good homework task to me...

## Posting messages and replies

After all this work, our forum system is still missing one rather fundamental component: the ability to post messages and replies. When we create the message.stencil template, I snuck in a hidden input like this:

```
<input type="hidden" name="title" value="Reply" />
```

I said at the time it makes our form parsing easy, and it's now time to see how. Message creation is either done with a new message, posting to "/forum/:forumid/", or with a reply to an existing message, posting to "/forum/:forumid/:messageid". The problem is, these two tasks are really similar: take some input form content, create a message out of it, then add it to CouchDB. Wouldn't it be nice to make a single route to handle both?

Well, it *just so happens* that's what we're going to do – it's almost as if I had planned ahead or something.

We can accomplish this magic by using a more advanced route: "/forum/:forumid/:messageid?". That is, "/forum/:forumid" followed optionally by a message ID. Our message reply form has a hidden "title" field in there because it means we can always rely on both "title" and "body" being presented, regardless of whether it's a new top-level post or a reply.

We're going to code this route in two parts again: the first part is ensuring we have all the required data, and the second part is creating the new message.

First, all the checks. We need to check that we have a "forumid" request parameter, because that's required whether we have a top-level post or a reply. Second, we need to make sure the user is logged in, because that's a rather fundamental requirement for making posts. Finally, we're going to check that both "title" and "body" fields were submitted by the user. If all three of those checks return successfully then we can proceed.

Here's the first part of the new route – add this to main.swift:

```
router.post("/forum/:forumid/:messageid?") {  
    request, response, next in
```

```

        guard let forumID = request.parameters["forumid"] else {
            try response.status(.badRequest).end()
            return
        }

        guard let username = request.session?["username"].string
        else {
            send(error: "You are not logged in", code: .forbidden,
            to: response)
            return
        }

        guard let fields = getPost(for: request, fields: ["title",
        "body"]) else {
            send(error: "Missing required fields", code: .badRequest,
            to: response)
            return
        }

        // OK to proceed!
    }
}

```

For the very last part of this project, we need to convert our user's form fields into a new document and send it to CouchDB. If it succeeds, we're either redirect the user to their new post if it was a top-level post, or just reload the current page if it was a reply.

Add this code in place of the **// OK to proceed!** comment:

```

var newMessage = [String: String]()
newMessage["body"] = fields["body"]!

// add the current date in the correct format
let formatter = DateFormatter()

```

```

formatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZ"
newMessage[ "date" ] = formatter.string(from: Date() )

// mark the message as belonging to the current forum
newMessage[ "forum" ] = forumID

// if we are replying to a message, use its ID as our parent
if let messageID = request.parameters[ "messageid" ] {
    newMessage[ "parent" ] = messageID
} else {
    // this is a top-level post, so it has no parent
    newMessage[ "parent" ] = ""
}

newMessage[ "title" ] = fields[ "title" ]!

// use the username value we unwrapped from the session
newMessage[ "user" ] = username

// mark this document as a message so our views work
newMessage[ "type" ] = "message"

// convert the dictionary to JSON and send it off to CouchDB
let newMessageJSON = JSON(newMessage)

database.create(newMessageJSON) { id, revision, doc, error in
    defer { next() }

    if let error = error {
        send(error: "Message could not be created",
code: .internalServerError, to: response)
    } else if let id = id {
        // the new document was created successfully!
    }
}

```

```
if newMessage[ "parent" ]! == "" {  
    // this is a top-level post – load it now  
    _ = try? response.redirect("/forum/\(forumID)/\(id)")  
} else {  
    // this was a reply – load the parent post  
    _ = try? response.redirect("/forum/\(forumID)/\  
(newMessage[ "parent" ]!)")  
}  
}  
}
```

That's it – we're done!

## Wrap up

This has been another huge project, but you've learned a massive amount along the way: sessions, encryption, CouchDB views, date formatting, and more. Plus, you made another complete project, with lots of room for you to add more if you're keen to practice.

At this point, you're hopefully starting to feel a bit more comfortable with CouchDB. Like I said earlier, its heavy reliance on closures is tiresome when you have several dependent queries to run, but sadly unavoidable at least right now.

If there's only one thing you remember from this project, make it this: securing user passwords is important to get right. I know it might seem like a hassle having to learn about hashes and salts, but it *matters*: if somehow your database gets hacked then you need to have done everything you can to keep your users safe.

## Homework

Even though this finished project is 300 lines of code, there's still a lot more to be done in order to finish it up.

To start with, lots of error messages just print a message to the screen and nothing more. That's fine when you're creating an API, but when you have a full website like this you need to show something useful to users. So, try replacing those simple text messages with a Stencil-rendered error page.

Second, creating a new user just prints "OK!" if everything worked. I'd like you to upgrade that page: save the new username to the user's session, then redirect the user back to the homepage so they can start browsing the available forums.

Third, add code to let users create their own forums. This is similar to creating users, although there's no need to add any password hashing and salting.

Finally, if you're feeling brave see if you can figure out how to let users edit messages they have posted.

# **Chapter 5**

## Meme Machine

## Setting up

This is the first of two easier projects in this book. By this point, it's very likely you're feeling overwhelmed by server-side Swift: databases, sessions, encryption, routing, templates and more can all get a bit much. So, in this project we're going to produce something small and simple, to give your brain a short break and to give your new knowledge a little time to sink in.

What we are we going to make? I'm glad you asked! We're going to make Meme Machine, which lets users upload and download images from our server. This requires learning how to list files from the filesystem and handle images being uploaded. To make things more interesting, I created a small server-side Swift framework that does simple image processing and resizing – we're going to use it to create thumbnails of all the pictures that get uploaded.

Create a new Swift executable package called project5, then modify its Package.swift file to this:

```
import PackageDescription

let package = Package(
    name: "project5",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
StencilTemplateEngine.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1),
        .Package(url: "https://github.com/twostraws/SwiftGD.git",
majorVersion: 1)
    ]
)
```

That pulls in Kitura, Kitura-Stencil, and Helium Logger, along with a new framework: SwiftGD. This uses the open source GD library for manipulating images, and wraps it in a simpler, "Swiftier" framework. I made it specially for this book, but it's useful for anyone

looking to manipulate images on the server.

If you're using my Docker image, GD is already installed and ready to go. If you're using your own custom Linux install, please install the libgd-dev package now. On macOS, run "brew install gd" to get the libraries.

Next, look for project5-files in the project files you received with this book, and copy the "public" and "Views" directories into your project folder. "public" contains the same Bootstrap code we've been using previously, but this time there's an extra directory called "uploads" in there. This in turn contains two more directories called "originals" and "thumbs", which is where we'll be storing the images that get uploaded.

Finally, we need to add some basic Kitura code to get us up and running. If you intend to use Xcode, generate the project file now and open main.swift in Xcode; otherwise just open main.swift in you preferred text editor. Now give it this content:

```
import Foundation
import HeliumLogger
import Kitura
import KituraStencil
import LoggerAPI
import SwiftGD

HeliumLogger.use()

let router = Router()

router.setDefault(templateEngine: StencilTemplateEngine())
router.post("/", middleware: BodyParser())
router.all("/static", middleware: StaticFileServer())

Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()
```

That's the same basic Kitura skeleton we've used previously, albeit with an extra **import** line

for my SwiftGD framework.

## Reading files from disk

We're going to start with the "/" route first, the homepage. This is going to read a list of images in the "uploads" and serve them up to a simple Stencil template.

First, we need to define three directories:

1. Our root directory is the path to "uploads". We're going to find this relative to the user's current directory path.
2. Our originals directory is the "originals" folder inside the root directory. This is where full-size images will be stored.
3. Our thumbs directory is the "thumbs" folder inside the root directory. This will be used to store images that have been scaled down to a maximum of 300px width.

These three paths are needed throughout our app. As a result, we're going to create them outside of any routes so they are available everywhere. Add these three lines of code directly underneath the **StaticFileServer** line in main.swift:

```
let rootDirectory = URL(fileURLWithPath: "\\\n(FileManager().currentDirectoryPath)/public/uploads")\nlet originalsDirectory =\nrootDirectory.appendingPathComponent("originals")\nlet thumbsDirectory =\nrootDirectory.appendingPathComponent("thumbs")
```

The first line defines **rootDirectory** to be the user's current path, with "/public/uploads" added to the end. We read this from the **FileManager** class from the Foundation framework, which lets us work with the filesystem. The second and third lines append "originals" and "thumbs" to the end of **rootDirectory**, giving us paths to both directories.

**Warning:** **currentDirectoryPath** points to wherever you launch your application from. This means running ".build/debug/project5" from your "project5" directory. If you don't do this, your directories will all be wrong and you'll hit major problems.

Now that we have paths to where our images are stored, we can write the "/" path. This needs to:

1. Get a list of all the images in the originals directory.
2. Strip out everything except the filename. (That is, when given /path/to/uploads/originals/hello.jpg, we just need hello.jpg)
3. Ensure that hidden files aren't shown. Hidden files on macOS and Linux are files that start with a period.
4. Pass the array of files to a Stencil template for rendering.

The first task is another job for **FileManager**: it has a **contentsOfDirectory()** method that returns the names of all filenames in a directory. This method throws an error if the directory can't be found (or could be found but not read), so we'll call it using **try?** to avoid problems.

It returns an array of **URL** objects to us, which contain the full path to each image. That's where the second task comes in: we need to convert /path/to/uploads/originals/hello.jpg to just hello.jpg so we can render reference both the original and the thumbnail images in our Stencil template.

We can convert the full array of **URL** objects into an array of filename strings using just one line of code, thanks to the **map()** method of arrays. **URL** objects have a property called **lastPathComponent** that contains just the filename part of the URL – the "hello.jpg" part. The **map()** method lets us read that property from every element in an array, and put it back into a new array, similar to the "map" functions of CouchDB views.

The syntax for this is a little strange at first. Here's an example:

```
let stringArray = urlArray.map { $0.lastPathComponent }
```

What you're seeing is called "closure shorthand syntax" – every item in the **urlArray** array is passed into the **map()** method as **\$0**, so we can read its last path component. That property, **lastPathComponent** gets added back into a new array, **stringArray**, which eventually contains the same number of items as **urlArray** did.

To give you a worked example, imagine **urlArray** contained the following three URLs:

1. /path/to/hello.jpg

2. /path/to/world.jpg
3. /path/to/swift.png

After running the `map()` code above, the `stringArray` result would contain the following three strings:

1. hello.jpg
2. world.jpg
3. swift.png

Once we have an array of the filenames, we're onto the third task: ensuring hidden files don't appear in the list. These are any files that start with a period ("."). so we can use Swift's `filter()` method to get rid of them. This works like `map()`: you give it a closure to run on each element (called `$0` inside the closure). Any element that evaluates to "true" inside your closure is included in the resulting array.

To remove items that begin with a period, we're going to use the `hasPrefix()` method of strings, like this:

```
let visibleFilenames = allFilenames.filter { !  
$0.hasPrefix(".") }
```

I've used `!` to flip the result: if we just wrote `$0.hasPrefix(".")` then `visibleFilenames` would contain only *hidden* files, which wouldn't be much good.

OK, that's everything for the "/" route, so add this code to main.swift:

```
router.get("/") {  
    request, response, next in  
    defer { next() }  
  
    let fm = FileManager()  
    guard let files = try? fm.contentsOfDirectory(at:  
originalsDirectory, includingPropertiesForKeys: nil) else  
{ return }
```

```

let allFilenames = files.map { $0.lastPathComponent }
let visibleFilenames = allFilenames.filter { !
$0.hasPrefix(".") }

try response.render("home", context: ["files": visibleFilenames])
}

```

All that's left now is to write `home.stencil` in the `Views` directory. This gets handed a "files" array by our Swift route, so we're going to loop over that to display a list of images that have been uploaded.

To make loading efficient, we're going to show the thumbnail on this homepage, but add a hyperlink to each of them so users can click to see the full-size originals. To make it look a little better, I've added a small amount of inline CSS to the images so they have a border as well as some space around them, and I also used a `target="_blank"` attribute for the hyperlink to make it open in a new window.

Here's the code for `home.stencil`:

```

{%- extends "master.stencil" %}

{%- block title %}Home{%- endblock %}

{%- block body %}
<h1>Files</h1>
{%- for file in files %}
<a href="/static/uploads/originals/{{ file }}"
target="_blank"></a>
{%- empty %}
<p>No files have been uploaded yet.</p>
{%- endfor %}
</ul>

```

As you can see, we can use `{} file {}` for both the original and the thumbnail, because all that's changing is the directory that contains them.

I provided a sample image and thumbnail with your "public" directory, so you should be able to build and run now.

## Multi-part form encoding

The second and final part of this project is to let users choose images, upload them, and create thumbnails – then we're done. Hurray for simple projects!

We've used URL-encoded form fields previously, but this time we need a different parser called multi-part encoding. Thanks to Kitura's **BodyParser** middleware, we don't need to do anything special to use this – it automatically detects content and parses it for us regardless of what type it is.

We're going to let the user choose as many files as they want, and upload them all in one. Once Kitura has finished parsing them, we'll be given an array of parts to sift through, telling us the file type and original filename of each part. We can also pull out the data for each part, although it's wrapped up in an enum with an associated type so we need to use some funky syntax like this:

```
guard case .raw(let data) = part.body else { continue }
```

That means "ensure **part.body** contains raw data, and if so pull out that data into a new constant called **data**.

Once we've pulled out the data for each part, we're going to copy it into our "originals" directory using the same filename as it originally had, then create a thumbnail using SwiftGD. To avoid problems, we're going to replace any spaces in filenames with a dash, meaning that "hello world.jpg" becomes "hello-world.jpg", but otherwise leave filenames as they are. We're also going to double-check the file type of each part, to ensure we're working with images rather than movies or something else.

We can write each image's data to disk using its **write(to:)** method. Once that completes, we can load it into a SwiftGD **Image** object, resize it down to thumbnail size, then save that back to the "thumbs" directory. SwiftGD offers a number of resizing methods, but the most useful here is **image.resizedTo(width:)** – we're going to ask for a width of 300 pixels, and have SwiftGD calculate the height to retain the correct aspect ratio.

That's it – nothing too difficult, I think, but I'll add some extra comments to walk you through. Add this route to main.swift:

```

router.post("/upload") {
    request, response, next in
    defer { next() }

    // pull out the multi-part encoded form data
    guard let values = request.body else { return }
    guard case .multipart(let parts) = values else { return }

    // create an array of the file types we're willing to accept
    let acceptableTypes = ["image/png", "image/jpeg"]

    for part in parts {
        // ensure this image is one of the valid types
        guard acceptableTypes.contains(part.type) else
        { continue }

        // attempt to extract its data; move onto the next part
        if it fails
            guard case .raw(let data) = part.body else { continue }

            // replace any spaces in filenames with a dash
            let cleanedFilename =
                part.filename.replacingOccurrences(of: " ", with: "-")

            // convert that into a URL we can write to
            let newURL =
                originalsDirectory.appendingPathComponent(cleanedFilename)

            // write the full-size original image
            _ = try? data.write(to: newURL)

            // create a matching URL in the thumbnails directory
            let thumbURL =
                thumbsDirectory.appendingPathComponent(cleanedFilename)
}

```

```

    // attempt to load the original into a SwiftGD image
    if let image = Image(url: newURL) {
        // attempt to resize that down to a thumbnail
        if let resized = image.resizedTo(width: 300) {
            // it worked - save it!
            resized.write(to: thumbURL)
        }
    }
}

// reload the homepage
try response.redirect("/")
}

```

All that's left now is to add a form to `home.stencil` that lets users select files. This time we're going to attach an **action** attribute that tells the form where to post its contents (we want `"/upload"`), as well as an **enctype** attribute that instructs web browsers to send file data as multi-part encoding.

File uploading is done using `<input type="file">`: users are presented with a "Choose files" button that opens a file browser for them to select with. We're going to allow multiple files to be uploaded at once, done by adding "multiple" inside the `<input>` tag. I should add that file upload inputs are one of the few – perhaps the only! – pieces of Bootstrap that have bad styling by default. If you're interested in making it look better try doing a Google search for "bootstrap file style" or similar and try some of the many suggestions!

Add this form to `home.stencil`, just before the end of the **body** block:

```

<form method="post" action="/upload" enctype="multipart/form-data">
<p><input type="file" name="upload" multiple/></p>
<p><button type="submit" class="btn">Upload</button></p>
</form>
{ % endblock %}

```

Remarkably enough, that's it! Build and run the code now, and you should be able to upload lots of images, get them resized, click to download them, and more. Nice!

## Wrap up

This was an intentionally small, simple project. Yes, it didn't cover much of the massive topics we've seen in the earlier projects, but you did learn two more useful skills: how to read directory contents on the filesystem, and how to accept file uploads.

In this project we focused specifically on images because they provide clear visual feedback, but the code from this project is great for other kinds of binary data too – movies, zip files, fonts, and so on. If you intend to handle text files, you'll need to rewrite the `.raw(let data)` part, because text files have the `.text` type rather than `.raw`.

We used only a small part of SwiftGD here, but resizing images is a task you're going to want to do frequently. It's also capable of converting formats (e.g. from PNG to JPEG), drawing shapes such as ellipses, lines, and rectangles, and filtering images to add blurs, colorizing, pixelation, and more. It's not ever going to do as much as Core Graphics, but it *does* run on Linux.

## Homework

Even though this is a small project, there are lots of things you could add to make it more complete. For example, you could add a check to avoid filename clashes if someone uploads a filename that exists already.

You could also add "Delete" links beneath each image, so users can remove them through their browser. Both of these can be implemented by exploring the `FileManager` class some more – I recommend you try using Xcode's autocomplete functionality.

# **Chapter 6**

## Templates

## Setting up

In this second technique project we're going to take a closer look at templating with Stencil. We've used it in several projects so far, but only really touched on a handful of features – it's time to take it a little further!

Create a new executable package called project6, then edit your Package.swift file to this:

```
import PackageDescription

let package = Package(
    name: "project6",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
StencilTemplateEngine.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/swift-html-
entities.git", majorVersion: 2)
    ]
)
```

You've met the first three of those dependencies before, but the fourth is one is new: it escapes HTML special characters so they are rendered as text rather than as tags. This means something like `<h1>` will be rendered as the literal string `<h1>` rather than a top-level heading.

Run `swift build` to fetch all the dependencies, then generate your Xcode project if you're using Xcode. Open main.swift and give it this basic code so we have something to build on:

```
import HeliumLogger
import Kitura
import KituraStencil
import Stencil
```

```
HeliumLogger.use()

let router = Router()
let namespace = Namespace()

// custom namespace code here!

router.setDefault(templateEngine:
StencilTemplateEngine(namespace: namespace))

Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()
```

Finally, create an empty "Views" directory for us to use with Stencil. We don't need any of the Bootstrap static file server stuff here; we're focusing just on Stencil.

## Recap on the basics

Before we look at some advanced functionality, let's just go over the basics of Stencil just quickly. Start by adding a simple route in main.swift to throw a variety of data at a Stencil template:

```
router.get("/") {
    request, response, next in
    defer { next() }

    let haters = "hating"
    let names = ["Taylor", "Paul", "Justin", "Adele"]
    let hamsters = [String]()
    let quote = "He thrusts his fists against the posts and
still insists he sees the ghosts"

    let context: [String: Any] = ["haters": haters, "names": names,
        "hamsters": hamsters, "quote": quote]
    try response.render("home", context: context)
}
```

That renders a template using two strings, an array with elements, and an empty array. Build and run the project, and leave it running – we can edit the Stencil template and hit refreshing without re-building the Swift project.

To begin with, try adding this to home.stencil in the "Views" folder:

```
<html>
<body>

{%- if haters %}
  <p>Haters are {{ haters }}.</p>
{%- endif %}

{%- if fakers %}
  <p>Fakers are {{ fakers }}.</p>
```

```
{% endif %}
```

```
</body>
```

```
</html>
```

The first **if** will evaluate to true because **haters** is set, so you'll see the message "Haters are hating." The second will evaluate to false because we didn't provide a **fakers** value in our Stencil context, so nothing will be printed there.

Loops are done using **for**, and can be placed inside **if** blocks. For example, we could print out a list of names only if the **names** variable is set in our context, like this:

```
{% if names %}
<h1>Names:</h1>
<ul>
  {% for name in names %}
    <li>{{ name|uppercase }}</li>
  {% endfor %}
</ul>
{% endif %}
```

That will print a title if **names** is set, then loop over each name and write it in uppercase.

Inside a **for** loop, Stencil automatically provides some extra variables to give you more information about the loop's progress:

- The **forloop.first** variable will be set when the current iteration is the first one.
- The **forloop.last** variable will be set when it's the last iteration.
- The **forloop.counter** variable will be set to the number of the current iteration, counting from 1.

To try these three out, we could rewrite the whole **names** loop like this:

```
{% for name in names %}
  {% if forloop.first %}
```

```

<li>The first name is {{ name|uppercase }}</li>
{%
  else %
    {%
      if forloop.last %
        <li>The last name is {{ name|uppercase }}</li>
      {%
        else %
          <li>Name number {{ forloop.counter }} is {{ name|uppercase }}</li>
      {%
        endif %
      {%
        endif %
      {%
        endfor %
    }
}

```

As well as looping over an array, you can read individual items using **first**, **last**, or a specific index. For example, try adding these four to your template:

```

<p>There are {{ names.count }} names:</p>
<p>The first name is {{ names.first }}</p>
<p>The second name is {{ names.1 }}</p>
<p>The third name is {{ names.2 }}</p>

```

They can be used inside conditions as well, meaning that you can use `{% if names.count == 5 %}` and similar.

Stencil's **for** blocks also support an **empty** block that will print some content if an array is empty. We sent in an empty array for the variable **hamsters**, so we might work with it like this:

```

{%
  for hamster in hamsters %
    <li>{{ hamster }}</li>
  {%
    empty %
      <li>There are no hamsters :( </li>
  {%
    endfor %
}

```

In this project, that will print "There are no hamsters", along with a sad face because everyone loves hamsters, right? Right.

Previously you've seen how you can use `{% extends "master.stencil" %}` to make one template build on another using blocks, and also `{% include 'copyright.stencil' %}` to import the contents of one template into another. But what we haven't seen are comments, which work just like comments in Swift: Stencil ignores them. Comments look like this:

```
{# This is a comment and will be ignored. #}
```

OK, that's the full range of basic Stencil features covered, and to be honest you can do a whole lot just with these. Sure, the advanced features we're about to cover are very useful, but if you really hated templates for some reason you could do most of your work in Swift.

## Filters and tags

Where Stencil really becomes powerful is when you start to connect parts of your template back to Swift code using filters and tags. A filter is a way of modifying template values as they are being presented, and tags let you inject whole pieces of content or even control flow using Swift.

First, we're going to add a new `reverse` filter, which reverses any text it's given – a nice and simple way to stop spoilers being revealed on a website, for example.

Previously we created a `format_date` Stencil filter, so you already know how this works: call `registerFilter()` with your filter name, along with a closure that will perform your transformation. The closure is given a single `Any?` value as its parameter, so we're going to attempt to typecast that to a string before we use it. Once we have a string, we'll reverse its characters and create a new string, then return that.

All that only takes a few lines of code, so add this below the `let namespace = Namespace()` line in main.swift:

```
namespace.registerFilter("reverse") { (value: Any?) in
    guard let unwrapped = value as? String else { return value }
    return String(unwrapped.characters.reversed())
}
```

For more advanced functionality, you should use tags. You've already used Stencil's built-in tags: `if`, `for`, `include`, and so on. However, you can also add your own tags and manipulate your template content however you wish.

Let's start with something simple: a `debug` tag that prints out all the variables in a context. This is useful when your projects are being developed by multiple people – if your template designer hits a problem it's useful for them to be able to see exactly what context you passed them.

Stencil makes this really easy to implement: it has a `registerSimpleTag()` method that's designed to work with tags that don't manipulate content, and also a `flatten()` method on its context that returns a dictionary containing all the values available for use in a template.

Our tags need to return a string of their parsed contents, but we can convert that dictionary to a string using the **String(describing:)** initializer – that will convert literally any Swift type into a string.

Add this code below the **reverse** filter:

```
namespace.registerSimpleTag("debug") { context in
    return String(describing: context.flatten())
}
```

Amazingly enough, that's it! To give it a try, add this line to `home.stencil`:

```
{% debug %}
```

You'll see output like this:

```
[ "names": [ "Taylor", "Paul", "Justin", "Adele" ], "haters": "hating", "hamsters": [ ], "quote": "He thrusts his fists against the posts and still insists he sees the ghosts", "loader": Stencil.FileSystemLoader ]
```

Finally, I want to introduce you to tags that work with template content. To demonstrate this, add this line to your `home.stencil` file:

```
<h1>This is a test</h1>
```

That writes "This is a test" in a large heading font. Now, if you had asked users to write some content into a text box for display on your forum from project 5, you don't really want them to add any HTML. In fact, letting them do so allows them to screw up your webpage layouts. Worse, they could inject JavaScript that steals user data and more – it's a whole world of pain.

The solution for this is to escape your HTML, which means to convert it to harmless text. We already imported the `swift-html-entities` package, but we're going to convert it to a Stencil tag so that any content inside our custom tag will automatically be escaped for safety.

We need to add a new file, which if you're not using Xcode can be done just by creating it

inside the Sources directory. If you *are* using Xcode, look in the project navigator for the "project6" group – that's where main.swift is. Now right-click on the "project6" title and choose New File, then select Swift File from the list of templates and click Next. Name it EscapeHTMLNode.swift, then make sure the only checked target in the Targets list is "project6".

**Warning:** This is important! Don't select LoggerAPI or one of the other frameworks – make sure only "project6" is selected.

We're going to write a new **EscapeHTMLNode** class in four simple steps. First, we need the skeleton of the class and a couple of imports. The class needs to inherit from **NodeType** so that it can be used inside Stencil, and we're also going to give it a **nodesToEscape** property that will store an array of all the Stencil nodes it contains. Those nodes are also **NodeType**, and can be anything – it could be some text, it couple of loops or conditions, and so on.

Start by adding this code to EscapeHTMLNode.swift:

```
import Stencil
import HTMLEntities

open class AutoescapeNode: NodeType {
    var nodesToEscape: [NodeType]
}
```

Next, we need an initializer: a method that gets called to create an **AutoescapeNode** object. All this is going to do is pass in the nodes it will contain, so this is nice and easy – add this method to the class:

```
public init(nodes: [NodeType]) {
    nodesToEscape = nodes
}
```

The third step is only fractionally harder: we're going to have our **nodesToEscape()** render themselves to a text string, then escape them. We imported the swift-html-entities framework already, which adds a **htmlEscape()** method to all Swift strings. When Stencil is ready for

our custom tag to draw itself, it will call a `render()` method and pass it the current context. We'll pass that straight onto the `renderNodes()` function to convert our `nodesToEscape` to a string, which we can then escape – it's surprisingly easy.

Add this method to the class:

```
open func render(_ context: Context) throws -> String {  
    let content = try renderNodes(nodesToEscape, context)  
    return content.htmlEscape()  
}
```

We don't need to catch any errors thrown by `renderNodes()` because `render()` is also a throwing function – any errors just automatically bubble upwards for Stencil to handle.

There's only one more step to complete, which is to write a `parse()` method. This gets called by Stencil when it finds the start of our custom tag. What happens is that our class becomes responsible for parsing the rest of the template – i.e., reading its text and converting it into instructions that Stencil can work with. Don't worry: we don't need to parse *everything*, just the bits we care about – we hand back control to Stencil once we've found the end of our tag.

Helpfully, Stencil does most of the work for us: we can ask it to carry on parsing the template until it finds "endautoescape", which will mark the end of our escape block. We'll be handed back an array of nodes that are between our start and end tags, which is what we need to create our `AutoescapeNode` object.

For safety reasons, we need to check that we haven't read the end of the template once parsing has finished; if there are no tokens left in the parser once it reaches "endautoescape" it means the closing tag wasn't found.

Before we look at the code, there's one more thing: the `parse()` method belongs to the `AutoescapeNode` class, not to instances of the class. It's a subtle difference, but an important one: this `parse()` method is responsible for creating our node type, so it can't actually be inside it.

Add this final method to the `AutoescapeNode` class:

```

class func parse(_ parser: TokenParser, token: Token) throws ->
NodeType {
    // find all the nodes inside our escape nodes
    let nodes = try parser.parse(until: ["endautoescape"]))

    // attempt to read the final token
    guard let _ = parser.nextToken() else {
        // there wasn't one - we reached the end!
        throw TemplateSyntaxError(`endautoescape` was not
found.)
    }

    // we have all our nodes: create a new AutoescapeNode from
them and send it back
    return AutoescapeNode(nodes: nodes)
}

```

That's our entire class written: it has a property to store its nodes, an initializer, a parsing method to load those nodes, and a rendering method to convert them to text and escape them.

Now that we have a custom tag written, we can connect it to Stencil by registering it. Add this line after the previous call to `registerSimpleTag()`:

```

namespace.registerTag("autoescape", parser:
AutoescapeNode.parse)

```

You can now go ahead and use the tag in your template. For example, if we wanted that `<h1>` to be rendered as text rather than HTML, we'd write this:

```

{%- autoescape %}
<h1>This is a test</h1>
{% endautoescape %}

```

Because `autoescape` is a full tag, you can add loops or other control structures inside there and have them all escaped.



## Wrap up

Stencil started off as a very simple template language for Swift, but has evolved quickly and continues to do so – it's really flying along, and great fun to work with.

As your server-side Swift projects start to expand, you'll come to appreciate the importance of separating your templates from your Swift code. HTML, JavaScript, and CSS are powerful in their own right, but trust me: you want them *nowhere near* your Swift code.

I strongly suggest you take the time to explore Stencil further. If you're the only person on your team, learning to master Stencil will benefit you greatly. If you're likely to have a dedicated template person, then it puts you in a better position to train them up. To get started, why not try creating a filter version of **autoescape** that can be applied to individual variables?

# **Chapter 7**

Barkr

## Setting up

I hope you enjoyed your short break, because this project is right back to the hard stuff: we're going to create a new server back-end to deliver a microblogging service like Twitter. Of course, Twitter is taken already, so we're going to be creating Barkr: a social media platform for pet owners.

That probably doesn't sound too complicated, but that's only because I haven't mentioned the big new thing you'll be learning. In project 2 you learned how to use the CouchDB NoSQL database, and we used it again in project 4 so you got some practice. Here, though, we're going to be using MySQL, which is probably the most popular open-source SQL database in the world.

SQL is a language all by itself, and although it's not complicated it does take some time to learn. Just like with CouchDB, I've written a dedicated SQL primer so you can experiment with SQL in isolation before going on to put it into practice in the rest of the project. If you've used SQL before feel free to skip over the primer!

If you're using Docker, you'll find MySQL already installed – just run the command "/etc/init.d/mysql start" to start the server. If you're using your own Linux distro, please install the packages "mysql-server", "mysql-client" and "libmysqlclient-dev", then start the MySQL server.

If you're on macOS, run these commands:

- "brew install mysql"
- "mysql.server start"

That will install and run the MySQL server on your Mac, so you're all set to start.

## An SQL primer

SQL is short for "Structured Query Language" – it's a dedicated programming language designed to read and write data in databases. It's big, it's old, and it's a little fragmented, meaning that some database systems support extensions that aren't available on other systems. However, SQL really is the standard way of interacting with major databases: Oracle uses it, MySQL uses it, Microsoft SQL Server uses it, and more. This means if you take the time to learn SQL to an intermediate level, that knowledge will pay off for years to come.

MySQL is a database server, like CouchDB. That means it's a separate piece of software running all the time, waiting for you to connect to it. This server might be on the same machine as your Kitura app, or it might be separate – it doesn't matter, as long as Kitura can reach it.

**Warning:** If you intend to use MySQL in production – i.e. on a live server – you should seek advice on how to secure your server, because it is not secure by default.

As well as a server component, MySQL also has a dedicated client app that lets you interact with your data directly. This is mostly used by administrators who need custom control over their data, but it's invaluable when you're learning – and it's what we'll be using here.

To get started, you need to launch the MySQL client. If you're using Linux, just running "mysql -u root" is probably enough. If you're using macOS with Homebrew, you'll need to use this:

```
/usr/local/Cellar/mysql/5.7.16/bin/mysql -u root
```

If you're using a custom Linux install, you might need to specify a root password like this:

```
mysql -u root -p
```

You'll be prompted for your password when you press return.

If everything has worked, you should see this in your terminal window:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.16-0ubuntu0.16.04.1 (Ubuntu)
```

`mysql>`

That last part is the MySQL prompt, where you can write your SQL. If you want to quit, press Ctrl+D.

The first thing we're going to do is create a database. Similar to CouchDB, you normally have one database per app, but you can create different kinds of data inside there. Run these commands in your MySQL prompt:

```
CREATE DATABASE swift;
USE swift;
```

It's not *required* to write your SQL in capital letters, but it's certainly common practice. Some text editors actually look for capital letters in order to enable syntax highlighting!

What we just did was create a new database called "swift", and switched to it. This means any future commands we execute will be run on tables in the "swift" database, rather than elsewhere.

Both the commands we just ran – and indeed all SQL commands – end with a semi-colon. If you press return after forgetting to type a semi-colon, your "mysql >" prompt will turn into "->", allowing you to add more to your command. Just press ";" and hit return to add the missing semi-colon.

We're using MySQL as the root user right now, which is fine when you're starting out but a disaster waiting to happen when you go much further. We need to create a new user, give them a password, then assign them access to the new "swift" database we just created. To do that, run these two commands:

```
GRANT ALL PRIVILEGES ON swift.* TO swift@localhost IDENTIFIED
BY 'swift';
FLUSH PRIVILEGES;
```

## Creating tables

The very first skills to learn are how to create tables in a database. Once you have your "swift" database created and enabled, you're all set to start creating tables using SQL. First, though, run this command:

```
`show tables;`
```

This is the command to make MySQL output a list of all tables in the currently selected database. You'll see "Empty set (0.00 sec)" returned, which means that a) you have no tables, and b) it took 0 seconds for MySQL to figure that out. MySQL reports execution time because complex queries can run slowly, which adds significant load to your server.

Creating a table is done using the **CREATE TABLE** SQL command. Run this SQL now:

```
CREATE TABLE `posts` (
  `id` INT,
  `user` VARCHAR(64),
  `message` VARCHAR(140),
  `parent` INT,
  `date` DATETIME
);
```

That creates a new table called "posts", and gives it five fields of data we want to store in that table. This is very different from CouchDB, where we could insert any kind of document with any fields, and have it be stored. With SQL databases, we need to describe exactly what each document type looks like and what data it will store. You can think of MySQL tables as being like one document type in CouchDB: rather than mixing "message" and "user" in one place, SQL databases would separate them into individual tables.

In the SQL above, we created five fields using four different data types. Let's take a closer look:

- The **id** field is declared as **INT**, which is an integer. This holds numbers up to 2 billion.
- The **user** field is declared as **VARCHAR(64)**, which means it holds a VARiable

length CHARacter array, up to 64 letters in size. There are also **CHAR** fields, which don't vary in length.

- The **message** field is declared as **VARCHAR (140)**, so it holds up to 140 characters.
- The **parent** field is declared as **INT**, so again it holds a number up to 2 billion.
- The **date** field is declared as **DATETIME**, which stores dates and times. This is more flexible than using an ISO 8601 string like we did with CouchDB.

There are two more things to note about the SQL. First, all our custom names – the name of the table, as well as the names of each field – are written inside backticks. This key is usually to the left of the 1 key or the Z key on your keyboard, depending on your layout. This isn't *required*, but it *is* good practice: if you don't use them, it's possible your table or field names might use SQL reserved names either now or in the future. The backticks tell MySQL "I mean my name, not any SQL stuff," so it's a good idea to use them everywhere you use your names.

If you haven't pressed returned already, please do so now. MySQL will attempt to create the "posts" table for you, with those five fields. If it exists already, or if you somehow don't have permission to create such a table, it will report an error.

You should notice that MySQL outputs "Query OK" first, which tells us that the query executed just fine - as expected. However, if you try running the same query again, MySQL will flag an error up because "posts" already exists and therefore can't be created again.

Once a table is created, you can run this command to have MySQL print out a small report about the fields in the table:

```
DESC `posts`
```

It will show the following information:

- Field: the name of the field.
- Type: the data type.
- Null: does this field allow missing information?
- Key: is this field a key? (A key is a value that has special importance.)
- Default: what is the default value of this field?
- Extra: stores comments about each field.

If you want to destroy a table you created – along with all its data – you would run this command:

```
DROP TABLE `posts`
```

Please don't delete the "posts" table; we'll be using it for the time being.

**Warning:** **DROP TABLE** deletes your table and all its data, without any "undo" ability. Be careful!

## Inserting data

To insert data into your table, you use the **INSERT INTO** SQL command, like this:

```
INSERT INTO `posts` VALUES (1, "twostraws", "Hello, world!", 0,  
NOW());
```

The **INSERT INTO** command above tells your database that you would like your data placed into the specified table - "posts", in the example. The data in between the parentheses is what goes into your fields, and it needs to match the table definition precisely.

Our table definition had the fields "id", "user", "message", "parent", and "date", the values we're entering are 1, "twostraws", "Hello, world!", "0", and "NOW()". Just like in Swift, character data needs to be surrounded by single or double quote marks whereas numbers don't. **NOW()** is a function, and inserts the current date and time.

If you want to insert a new document (called a "row") using only some of the fields, just list the ones you intend to use. For example:

```
INSERT INTO `posts` (`id`, `user`, `date`) VALUES (1,  
"twostraws", NOW());
```

MySQL will use default values for any fields you don't enter. Just like Swift, SQL has the concept of "optionals", called "null values." When you don't provide a value when inserting a row, MySQL will place null in there.

## Reading data

Now that we have some data in our table, we can try reading it back using the **SELECT** SQL command. Here is a basic example:

```
SELECT `id`, `user` FROM `posts`;
```

**SELECT** takes this basic form:

```
SELECT `field1`, `field2`, ... FROM `table`;
```

You can read as many fields as you like, or use **SELECT \*** to read them all at once. As your skills progress with SQL, you can also create more complicated queries such as selecting multiple rows from across different data sets, perform functions on data before its returned, and so on.

Try running a few queries now, pressing return after each one:

```
SELECT `id` FROM `posts`;
SELECT `user`, `message` FROM `posts`;
SELECT * FROM `posts`;
```

Each time you run a query, MySQL will find all rows that match then show them to you neatly formatted as a table. Being able to write custom queries and see your results immediately is really helpful while you're learning!

MySQL draws each field with its own header so you can see what's in there. This becomes important later on, particularly when you start running functions on data. MySQL shows all fields in the order you selected them, but if you run **SELECT \*** then MySQL selects all fields from the table and shows them in the order they were created in the **CREATE TABLE** call.

**SELECT** is probably the most complex SQL statement of them all - I'll be demonstrating more of its power later on. Right now, there's just one last thing you need to learn before you can continue: how to limit your result. **SELECT** has a number of clauses you can add that affect the data selected, and one of them is **WHERE**: it allows you to force MySQL to return only rows

that match criteria you specify, similar to views in CouchDB.

Our test entry in the database has an "id" value of 1, so what do you think this next query will do?

```
SELECT * FROM `posts` WHERE `id` > 1;
```

As you probably guessed, it will look through all of **posts**, checking the "id" values against 1, and return only rows where the "id" field is greater than 1. If you try that on your database, you will find MySQL returns no rows as expected.

Now try this:

```
SELECT * FROM `posts` WHERE `id` <= 1;
```

This time the "Hello, world!" message should be back in the rows returned, because this row matches the **WHERE** clause we specified.

You can provide more specific **WHERE** clauses using **AND** to add more than one filter, but you should be aware that when you use **AND** with MySQL it will match both **WHERE** clauses, which isn't quite how English works. For example, if you say "show me all the bears and tigers" you probably mean "show me all animals that are either bears OR tigers" but MySQL would read that to mean "animals that are bears and are also tigers," which is clearly no animal – or, if it is an animal, is one you probably never want to meet!

The other popular clause is **LIMIT**, which allows you to force MySQL to only return a certain number of results. **LIMIT** can be used in two ways: **LIMIT n** and **LIMIT m, n**. The first way allows you force MySQL to only return the first *n* rows that match your **WHERE** clause (if you have one), whereas the second option allows you to force MySQL to return the first *n* rows after the first *m* that match.

If we had hundred rows of matching data, here are three possible SELECT statements:

```
SELECT * FROM `some_table` LIMIT 1;  
SELECT * FROM `some_table` LIMIT 10;  
SELECT * FROM `some_table` LIMIT 20, 10;
```

The first one will return just the first row, the second one will return the first ten rows, and the last one will skip the first twenty rows, and return the next ten. Each type of **LIMIT** has its own use - the last option, for example, allows you to do search engine-style results, "See next ten matches".

**Tip:** Some databases use the syntax **SELECT TOP 5 \* FROM..** as opposed to **SELECT \* ... LIMIT 5;**. You will need to read your database documentation to determine the correct usage for your system.

There are two particular **SELECT** keywords that will be of particular use to you: **DISTINCT** and **AS**.

The **DISTINCT** keyword is used in **SELECT** statements to remove duplicate result rows. If you wanted to find all distinct users in your table, you would use this:

```
SELECT DISTINCT `user` FROM `posts`;
```

Note that **DISTINCT** matches all attributes that you have selected - that is, if you select "user", "message", and "date", then all three must be the same for a match to be made.

Normally when you **SELECT** records from your table, they are made available to you in Swift by their original field names. When you use **AS**, MySQL lets you temporarily change the field names to something that is more useful to you. To try it yourself, run this query:

```
SELECT `user` AS `username` FROM `posts`;
```

Even though the field is called **user**, you'll see **username** printed out in the MySQL results. That might seem pointless, but it's important when you start using MySQL functions. For example, one function is called **CONCAT()**, and it merges two or more fields into one. You can mix and match fields and your own text, so for example we could write a query like this:

```
SELECT CONCAT(`user`, ' says: ', `message`) FROM `posts`;
```

That will print "twostraws says: Hello, world!" for our data, because **CONCAT()** has joined it all into a single result. But if you look at the column header, you'll see it has the full

**CONCAT()** call in there, which is horrible to work with once you start using Swift. So, as the **AS** keyword is useful to rename results, like this:

```
SELECT CONCAT(`user`, ' says: ', `message`) AS  
`formatted_message` FROM `posts`;
```

When you run *that* you'll see "formatted\_message" in the column header, which is much nicer.

## Updating data

You've seen creating data (using **INSERT INTO**) and reading data (using **SELECT**), so the next stop is *updating* data – changing what's already there. This uses the **UPDATE** command, optionally adding any **WHERE** clauses just like you had with **SELECT**.

You specify all the fields you want to change, and the values you want to change them to. For example:

```
UPDATE `posts` SET `user` = "taylorswift13", `parent` = 556;
```

That statement will change all records in "posts" so that they have the user "taylorswift13" and parent "30". We can filter the rows to be updated by using a **WHERE** clause, just like with **SELECT**. So, to change all people with username "twostraws" to username "adele", we would use this statement:

```
UPDATE `posts` SET `username` = 'adele' WHERE `username` =  
'twostraws';
```

**UPDATE** can also use the LIMIT clause to update only the first *n* matching rows, but this is not commonly used.

## Deleting data

Finally, let's look at how you delete data from a table. Note that when you delete something from MySQL, it's deleted for real – gone, no more, etc. This is inherent to all databases, not just MySQL.

Here is the most basic way to delete data from a table:

```
DELETE FROM `posts`;
```

That will delete all rows from a table, in the same way **SELECT \*** will select all rows from a table. **DELETE** takes both the **WHERE** and **LIMIT** clauses, allowing you to construct more complicated statements such as this one:

```
DELETE FROM `posts` WHERE `user` = 'twostraws' LIMIT 3;
```

## Working with keys

A "key" field is one with special significance. Keys come in several forms, but the one you'll be meeting most often is called a *primary key*. This is a table field just like any other, but it has one important restriction: it must always contain unique values. So, if we made "id" our primary key, we're saying that each post will have its own, unique ID number.

Working with unique integer primary keys is so common that MySQL gives us a shortcut: we can ask it to assign numbers for us, starting at 1 and automatically incrementing as each new row is added.

To demonstrate this, let's recreate the **posts** table an automatically incrementing "id" field. First we need to delete it, like this:

```
DROP TABLE `posts`;
```

Now recreate it with this:

```
CREATE TABLE `posts` (
  `id` INT PRIMARY KEY AUTO_INCREMENT,
  `user` VARCHAR(64),
  `message` VARCHAR(140),
  `parent` INT,
  `date` DATETIME
);
```

Because we just destroyed and recreated our table, the single row we added has been deleted too. We can recreate it using **INSERT INTO** again, but this time we're going to specify which fields we're adding so we can purposefully skip the "id" field. Run this:

```
INSERT INTO `posts` (`user`, `message`, `parent`, `date`)
VALUES ("twostraws", "Hello, world!", 0, NOW());
```

We can skip "id" because MySQL will fill it in for us using its automatic incrementer. To check the result, try this:

```
SELECT * FROM `posts`;
```

You'll see it has the ID 1, even though we didn't specify it.

## Complex queries

The reason for SQL's enduring popularity is that it lets you construct complex, expressive queries that let you find precise pieces of data efficiently.

To demonstrate this, we need some more sample data to work with. Look for project7-files in the project files for this book, and open 0.sql – that contains the tables we'll be using in this project, along some sample data. I'd like you to select all the text, copy it to your clipboard, then paste it into the MySQL app so that all the example tables and data get created.

That test data contains a few command like this one:

```
SELECT SLEEP(1);
```

That causes MySQL to pause for one second. This is a little hack so that the **NOW()** function returns different dates for each of the posts – it makes querying more interesting!

Now that we have some interesting data, we can try more complex queries. Let's start simple and work our way up:

Select all posts by every user:

```
SELECT * FROM `posts`;
```

Select all posts by the user "twostraws":

```
SELECT * FROM `posts` WHERE `user` = 'twostraws';
```

Select all posts by the user "twostraws", newest first:

```
SELECT * FROM `posts` WHERE `user` = 'twostraws' ORDER BY `date` DESC;
```

The **ORDER BY** keyword lets you specify one or more fields to sort on, where **ASC** sorts upwards (A-Z, 0-9) and **DESC** sorts in reverse.

Select all posts by "twostraws" or "adele", newest first:

```
SELECT * FROM `posts` WHERE `user` = 'twostraws' OR `user` = 'adele' ORDER BY `date` DESC;
```

Select all posts where the ID is greater than or equal to 3 and less than or equal to 5:

```
SELECT * FROM `posts` WHERE `id` >= 3 AND `id` <= 5;
```

Select all posts by any user where the message contains the word "cats":

```
SELECT * FROM `posts` WHERE `message` LIKE '%cats%';
```

The **LIKE** keyword performs a fuzzy match, and "%" means "anything". So, "%" means "any text, followed by 'cats', followed by any text." That will match two posts in our test data, one from "twostraws" and one from "adele".

## MySQL functions

You've already met **CONCAT()**, which joins text strings together. MySQL has many more like that, some of which work on individual columns, and others that work on groups of data.

The results of functions are fields just like the others in your data, so you can sort them if you wish. For example, we could retrieve the text for all messages, count the number of letters, and sort the results so that the longest messages are shown first – all in one command. Here it is:

```
SELECT `message`, LENGTH(`message`) AS `length` FROM `posts`  
ORDER BY `length` DESC;
```

Using **AS** to rename the result of the **LENGTH()** means we can order the results more easily. As you've probably figured out, **LENGTH()** counts the number of characters in a field.

Another useful function is **MAX()**, which returns the single greatest value out of a range. For example, we could write a query that finds that maximum date (i.e. latest date) of any post written by the user "taylorswift13", like this:

```
SELECT MAX(`date`) FROM `posts` WHERE `user` = "taylorswift13";
```

**MAX()** goes through all rows that match your **WHERE** clause, then calculates one result. The same is true of the **COUNT()** function, which simply returns how many rows it found. For example:

```
SELECT COUNT(*) FROM `posts` WHERE `user` = "adele";
```

That will count how many posts the user "adele" has made.

Just like with Swift, SQL lets you stack functions one after the other. For example, we can use **AVG()** to calculate the average of several results, along with **LENGTH()** to measure text, like this:

```
SELECT AVG(LENGTH(`message`)) FROM `posts` WHERE `user` =  
"twostraws";
```

That means "find the message length of all posts by "twostraws", then calculate the average."

# Integrating MySQL with Kitura

It's time to put the MySQL app to one side, and move to Kitura. It's worth keeping the MySQL app open in a terminal window, though – if you hit problems, it's helpful to be able to double-check your data with a quick SQL query.

First, let's get the basics out of the way: create a new executable package called `project7`, then give the following `Package.swift`:

```
import PackageDescription

let package = Package(
    name: "project7",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
BlueCryptor.git", majorVersion: 0, minor: 8),
        .Package(url: "https://github.com/vapor/mysql.git",
majorVersion: 1),
    ]
)
```

You've met the first three of those before, but the last one is new: "`vapor/mysql`". You might recognize Vapor, because it's an alternative server-side Swift framework to Kitura, so it seems strange to use it here when this book is about Kitura development.

The simple truth is that smart developers are *pragmatic* developers. The Vapor team have some great code, the Kitura team have some great code, and it's a smart move to pick and choose what works best rather than saying something bizarre like "I write Kitura code, so I must only use Kitura frameworks!"

Go ahead and run "`swift build`" now, because it might not work. If you're using my Docker image or your own custom Linux build, it should work fine. But if you're using macOS you'll

probably see an error like this: "ld: library not found for -lmysqlclient for architecture x86\_64".

To fix this, run the command "swift build -Xlinker -L/usr/local/lib" instead. Unfortunately, you need to run that command every time rather than just "swift build", so get used to press the up cursor key on your keyboard to load old commands!

If you're using Xcode, you need to select your project, choose the project11 target, then go to the Build Settings tab and filter for "Other Linker Flags". There won't be any by default, but I'd like you to add one: "-L/usr/local/lib" – that tells Xcode to look where Homebrew installed MySQL's library.

Now that everything builds correctly, open main.swift and give it this content:

```
import Cryptor
import Foundation
import Kitura
import KituraNet
import HeliumLogger
import LoggerAPI
import MySQL
import SwiftyJSON

HeliumLogger.use()
let router = Router()
router.post("/", middleware: BodyParser())

Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()
```

The first route we're going to create is "/:user/posts", which will load the messages from a specific user. This will use a MySQL **SELECT** query to filter our "posts" table, then convert the result into a SwiftyJSON object.

Before we can run queries, though, we need to connect to MySQL. Rather than create one connection and try to hold it open forever, we're going to create a new connection every time a

route is executed. This means all the queries for a given Kitura request will be handled by the same MySQL connection.

Vapor separates database connections into two classes, **Database** and **Connection**. The **Database** class lets us specify a host, username, password, and database name, and prepares the MySQL API to start connections. The **Connection** class represents one real network connection from our client to the MySQL, able to execute queries and get data back.

While it's possible to ignore to ignore the **Connection** class and run queries directly on a **Database** object, it ends up creating a new connection for every query you make. So, what we're going to do is create a **Database** object using our login credentials, then call its **makeConnection()** method to get a **Connection** object we can use and re-use as needed.

To simplify all this, we're going to create a **connectToDatabase()** method to run through the entire process for us. It's going to return both the **Database** and **Connection** objects, and we just need to feed in our database credentials.

If you just asked yourself, "what credentials?", it's OK. We added them in the previous chapter: we created a database called "swift", then created a user called "swift" with the password "swift" – that was the whole "GRANT ALL PRIVILEGES" bit.

So, our credentials are nice and easy to remember: the host is "localhost", then the user, password, and database are all "swift".

Add this function to main.swift directly underneath all the **import** lines:

```
func connectToDatabase() throws -> (Database, Connection) {
    let mysql = try Database(
        host: "localhost",
        user: "swift",
        password: "swift",
        database: "swift"
    )

    let connection = try mysql.makeConnection()
```

```
    return (mysql, connection)
}
```

Now that we have a way to connect to MySQL, we can get back to the "/:user/posts" route. It's easy to read the ":user" part of the route using a **guard** block like this:

```
guard let user = request.parameters["user"] else { return }
```

However, getting that into the SQL query takes some thinking. Your initial thought might be something like this:

```
SELECT `id`, `user`, `message`, `date` FROM `posts` WHERE  
`user` = '\(user)' ORDER BY `date` DESC;
```

That uses Swift string interpolation to construct the complete SQL query. That approach definitely works, but it also opens your app to the most serious type of vulnerability: SQL injection.

To illustrate this, imagine if an evil hacker requested the following URL:

```
/';DROP TABLE posts;--/posts
```

If we took that value and placed it directly into our query, the result would be this:

```
SELECT `id`, `user`, `message`, `date` FROM `posts` WHERE  
`user` = ''; DROP TABLE posts;--' ORDER BY `date` DESC;
```

The "--" part is the SQL equivalent to "//" in Swift – it means "everything that follows is a comment."

Thanks to our string interpolation, we're actually running two SQL queries: one that selects all posts where user is "" (an empty string), and a second one that drops the whole "posts" table. By submitting a carefully crafted string, the evil hacker has managed to execute their own SQL on our server.

This whole attack is called SQL injection, because hackers can literally inject their own SQL

into your code, with disastrous results. The solution is to separate your query from any parameters you want to send, because that way the database ensures every parameter is safe and not some horrible hack.

Here's how that looks in Swift:

```
guard let user = request.parameters["user"] else { return }

let (db, connection) = try connectToDatabase()

let query = "SELECT `id`, `user`, `message`, `date` FROM
`posts` WHERE `user` = ? ORDER BY `date` DESC;"
let posts = try db.execute(query, [user], connection)
```

The `?` in the query string gets merged with the values passed in as the second parameter to `execute()`, in the order they appear. The end result is the same as using string interpolation, but MySQL ensures that the `user` value is definitely safe before using it – SQL injection is no longer possible. You need to pass in our `connection` object as the third parameter. Note: there is actually a `connection.execute()` method, but if you use that you'll get warnings about `db` not being used and you risk it being released before we're finished with it.

Once we have data back from MySQL, we need to convert it to JSON. This ought to be simple, but sadly it isn't – Vapor's MySQL code sends back an array of `Node` objects, which are similar to but entirely incompatible with SwiftyJSON objects. So, we need to unpack the nodes into dictionaries, then repack that into SwiftyJSON ready to send. We can do most of that with a loop:

```
var parsedPosts = [[String: Any]]()

for post in posts {
    var postDictionary = [String: Any]()
    postDictionary["id"] = post["id"]?.int
    postDictionary["user"] = post["user"]?.string
    postDictionary["message"] = post["message"]?.string
    postDictionary["date"] = post["date"]?.string
```

```
    parsedPosts.append(postDictionary)
}
```

So, to recap, we need to:

1. Pull out the ":user" parameter from our request.
2. Connect to the MySQL server using our "swift" credentials.
3. Create a query string using question marks to mark values that will be provided separately.
4. Call the **execute()** method to merge that query string with our parameters and get a result back.
5. Convert the result into dictionaries of data.
6. Wrap that up in a SwiftyJSON object, and send it out.

Here is all that in code, with comments matching the numbers above:

```
router.get("/:user/posts") {
    request, response, next in
    defer { next() }

    // 1. Figure out which user to load
    guard let user = request.parameters["user"] else { return }

    // 2. Connect to MySQL
    let (db, connection) = try connectToDatabase()

    // 3. Create a query string with gaps in
    let query = "SELECT `id`, `user`, `message`, `date` FROM
`posts` WHERE `user` = ? ORDER BY `date` DESC;"

    // 4. Merge the query string with our parameter
    let posts = try db.execute(query, [user], connection)

    // 5. Convert the result into dictionaries
```

```

var parsedPosts = [[String: Any]]()

for post in posts {
    var postDictionary = [String: Any]()
    postDictionary["id"] = post["id"]?.int
    postDictionary["user"] = post["user"]?.string
    postDictionary["message"] = post["message"]?.string
    postDictionary["date"] = post["date"]?.string
    parsedPosts.append(postDictionary)
}

var result = [String: Any]()
result["status"] = "ok"
result["posts"] = parsedPosts

// 6: Convert the response to JSON and send it out
let json = JSON(result)

do {
    try response.status(.OK).send(json: json).end()
} catch {
    Log.warning("Failed to send /:user/posts for \(user): \(error.localizedDescription)")
}
}
}

```

Build and run your code, then try browsing to <http://localhost:8090/twostraws/posts> in your browser. If everything has worked, you should see JSON containing three posts.

## Generating tokens

The second route we're going to make is "/login", which isn't something we've tackled yet. Sure, we handled logging in as part of the forum system in project 4, but there we were able to store an authenticated username using Kitura-Session. This time we're building a back-end, not a website, so we need a way for client apps to authenticate themselves securely.

One option is to send the username and password with every request. So, if the user wants to post a message they send their username, password, and message; if they want to post another, they repeat the process. That works, and it's certainly simple to do, but it isn't very flexible.

Instead, I want to walk you through building a very simple token authentication system: the user sends their username and password to our API once, and receives back a random string of letters and numbers that identifies them uniquely. That token doesn't contain any part of their username or password; it's just random – but it *is* unique. When the user wants to post messages, they attach their token and we use that to look them up on the server. More advanced token implementations *do* contain user information, but we're keeping it simple here.

There are lots of advantages to using a token-based authentication system, even a simple one. For example, if I say your token is "abc123", you can pass that to someone else you trust to use our API on your behalf. You might not trust them with your password, but the token is different from your password.

Tokens can also be revoked, i.e. deleted from the system, if you want to terminate access. So, you could create tokens to connect our API to all your favorite social media services, then later on decide you don't really want Facebook to continue its access. Rather than changing your password and having everything else break, you can just revoke your Facebook token.

It's also common for tokens to *expire*, i.e. to stop being valid for a user after a set period of time has elapsed. In this project we'll be using 24 hours, but it could be any number.

Before we're able to write the code for the "/login" route, we first need to add three helper functions we've used before: the `removingHTMLEncoding()` string extension, plus `getPost()`, and `password(from:salt)`. I suggest you save yourself the typing and just copy these from previous projects:

```

extension String {
    func removingHTMLEncoding() -> String {
        let result = self.replacingOccurrences(of: "+", with: " ")
        return result.removingPercentEncoding ?? result
    }
}

func getPost(for request: RouterRequest, fields: [String]) -> [String: String]? {
    guard let values = request.body else { return nil }
    guard case .urlEncoded(let body) = values else { return nil }

    var result = [String: String]()

    for field in fields {
        if let value =
body[field]?.trimmingCharacters(in: .whitespacesAndNewlines) {
            if value.characters.count > 0 {
                result[field] = value.removingHTMLEncoding()
                continue
            }
        }
    }

    return nil
}

return result
}

func password(from str: String, salt: String) -> String {
    let key = PBKDF.deriveKey(fromPassword: str, salt: salt,
prf: .sha512, rounds: 250_000, derivedKeyLength: 64)
}

```

```
    return CryptoUtils.hexString(from: key)
}
```

Now that we have those helper functions, let's turn to "/login". This needs to perform three SQL queries in total: one to retrieve authentication credentials for the user, one to delete any expired tokens, and one more to store the new token so that future calls work.

The first query is easy enough: when a user submits their username and password, we can use that username to find their record in the **users** table. If you look at 0.sql in your project files, you'll see that each user has a "password" and "salt" field.

```
SELECT `password`, `salt` FROM `users` WHERE `id` = ?;
```

We can authenticate the user just like we did in project 4: take the saved salt, use it to calculate their hashed password using the password text that was submitted, and compare the result against the saved password hash.

Before we're ready to generate a new token, we first want to delete any existing tokens that have expired. This is as simple as using **DELETE** with a **WHERE** clause, like this:

```
DELETE FROM `tokens` WHERE `expiry` < NOW()
```

Now, in theory that ought to be run every time any API call is made, but as long as you have people authenticating frequently just having it inside the "/login" route will be enough.

Finally, generating a new token. We can generate a nice and long random string using **UUID().uuidString**. **UUID** is one of Apple's Foundation classes, and generates universally unique identifiers on demand. We're going to attach that to the user's username, then make the token expire one day from now.

Manipulating dates is tricky, and you'll often see people say "well, there 60 seconds in a minute, 60 minutes in an hour, and 24 hours in a day, so that means there are 86,400 seconds in a day – I'll just add 86400 to my time in seconds." That approach causes all sorts of issues with timezones and daylight savings time, so it's not recommended. Instead, MySQL has a dedicated way of adding time intervals to **DATETIME** values, like this:

```
DATE_ADD(NOW(), INTERVAL 1 DAY)
```

We can put that right into our SQL to save tokens, like this:

```
INSERT INTO `tokens` VALUES (?, ?, DATE_ADD(NOW(), INTERVAL 1 DAY));
```

That's the entire route covered, so please add this to main.swift:

```
router.post("/login") {
    request, response, next in
    defer { next() }

    // make sure our two required fields exist
    guard let fields = getPost(for: request, fields: ["username", "password"]) else {
        send(error: "Missing required fields", code: .badRequest,
            to: response)
        return
    }

    // connect to MySQL
    let (db, connection) = try connectToDatabase()

    // pull out the password and salt for the user
    let query = "SELECT `password`, `salt` FROM `users` WHERE `id` = ?;"
    let users = try db.execute(query, [fields["username"]!], connection)

    // ensure we got a row back
    guard let user = users.first else { return }

    // pull both values out from the MySQL result
```

```

        guard let savedPassword = user["password"]?.string else
    { return }
        guard let savedSalt = user["salt"]?.string else { return }

        // use the saved salt to create a hash from the password
        // that was submitted
        let testPassword = password(from: fields["password"]!, salt:
        savedSalt)

        // compare the new hash against the existing one
        if savedPassword == testPassword {
            // success – clear out any expired tokens
            try db.execute("DELETE FROM `tokens` WHERE `expiry` <
        NOW()", [], connection)

            // generate a new random string for this token
            let token = UUID().uuidString

            // add it to our database, alongside the username and a
            // fresh expiry date
            try db.execute("INSERT INTO `tokens` VALUES (?, ?, ,
        DATE_ADD(NOW(), INTERVAL 1 DAY));", [token,
        fields["username"]!], connection)

            // send the token back to the user
            var result = [String: Any]()
            result["status"] = "ok"
            result["token"] = token

            let json = JSON(result)

            do {
                try response.status(.OK).send(json: json).end()
            } catch {

```

```
        Log.warning("Failed to send /login for \(user): \(  
        error.localizedDescription)  
    }  
}  
}
```

Because that new route uses HTTP POST, you need to test it using Curl. Something like this ought to work:

```
curl localhost:8090/login -d "username=twostraws" -d  
"password=twostraws"
```

You should receive back the OK status, along with a new token valid for a day.

## Reading insert IDs

The final route for this app is "/post/:reply?", which works similar to posting messages in the forum project. If you recall, posting original messages and posting replies to other messages use very similar logic, so we can combine them into a single route by making the ":reply" parameter optional.

This route works similarly to the previous routes: use `getPost()` to read the required form fields ("token" and "message" in this case), connect to MySQL and check the token is valid, then insert the new message into our "posts" table.

There are two small extras, though. First, we need to look for the ":reply" request parameter, and provide a default value if it doesn't exist. Regardless of what value it is, we'll be converting it to an integer like this:

```
Int(replyTo) ?? 0
```

That uses the nil coalescing operator to provide a default value. Creating an integer from a string might fail, because you might try to convert the text "fish" rather than something sensible. As a result, `Int(someString)` returns an `Int?` rather than an `Int`. Using the nil coalescing operator lets us remove the optionality, because it tells Swift to unwrap the converted integer if the process succeeded, but if it failed Swift will just use 0. Either way, it's guaranteed to have a full `Int`.

Second, we're going to send back to the user the ID number of the post they just created. This can be done using a special `SELECT` query:

```
SELECT LAST_INSERT_ID() as `id`;
```

That will return the ID number of the row most recently inserted by the current MySQL connection. After inserting the new message, we'll run that query and send its result back to our user.

Add this final route to main.swift:

```
router.post("/post/:reply?") {
```

```

request, response, next in
defer { next() }

// ensure our two required fields exist
guard let fields = getPost(for: request, fields: ["token",
"message"]) else {
    _ = try? response.status(.badRequest).send("Missing
required fields").end()
    return
}

// user the value of "reply" from the URL, or set a default
value
let replyTo: String

if let reply = request.parameters["reply"] {
    replyTo = reply
} else {
    replyTo = ""
}

// connect to MySQL
let (db, connection) = try connectToDatabase()

// check that we're using a valid token
let readQuery = "SELECT `user` FROM `tokens` WHERE `uuid`"
= ?;"
let users = try db.execute(readQuery, [fields["token"]!],
connection)
guard let username = users.first?["user"]?.string else
{ return }

// insert a new post for this user
let writeQuery = "INSERT INTO `posts` (user, message,

```

```

parent, date) VALUES (?, ?, ? NOW())"
    try db.execute(writeQuery, [username, fields["message"]!,
Int(replyTo) ?? 0], connection)

// pull out the ID of the new post
let lastInsertID = try db.execute("SELECT LAST_INSERT_ID()
as `id`;", [], connection)
guard let insertID = lastInsertID.first?["id"]?.string else
{ return }

// send the result back to the client
var result = [String: Any]()
result["status"] = "ok"
result["id"] = insertID

let json = JSON(result)

do {
    try response.status(.OK).send(json: json).end()
} catch {
    Log.warning("Failed to send /post for \(username): \
(error.localizedDescription)")
}
}
}

```

That's it – the app is complete!

## Over to you: fuzzy search

I realize SQL is tricky, so before we leave this project we're going to do a little guided homework. I want you to add a route for "/search", to let users search for messages that interest them. For example, "/search?text=cats" should return any rows that contain the word "cats".

This route is very similar to("/:user/posts", so I'd like you to try it yourself. I've written my solution in case you get stuck, but it's a good idea to try it yourself first.

Here are two lines to get you started:

```
guard let search = request.queryParameters["text"] else
{ return }
let searchFuzzy = "%\($search)%"
```

The first line reads the value of "text" in the URL (that's "cats" in the example above), and the second line adds percent symbols either side so that a **LIKE** search can work.

Give it a try yourself, using the "/:user/posts" route as an example – the two really are very similar. Once you've tried it yourself, you can find my solution below.

```
router.get("/search") {
  request, response, next in
  defer { next() }

  // ensure we have a search to work with
  guard let search = request.queryParameters["text"] else
  { return }

  // add percent signs either side so it matches anywhere
  let searchFuzzy = "%\($search)%"

  // connect to MySQL
  let (db, connection) = try connectToDatabase()

  // run the query, adding our search parameter
```

```

let query = "SELECT `id`, `user`, `message`, `date` FROM
`posts` WHERE `message` LIKE ? ORDER BY `date`;"
let posts = try db.execute(query, [searchFuzzy], connection)

// convert the result to dictionaries
var parsedPosts = [[String: Any]]()

for post in posts {
    var postDictionary = [String: Any]()
    postDictionary["id"] = post["id"]?.int
    postDictionary["user"] = post["user"]?.string
    postDictionary["message"] = post["message"]?.string
    postDictionary["date"] = post["date"]?.string
    parsedPosts.append(postDictionary)
}

var result = [String: Any]()
result["status"] = "ok"
result["posts"] = parsedPosts

// convert the dictionaries to JSON, and send it off
let json = JSON(result)

do {
    try response.status(.OK).send(json: json).end()
} catch {
    Log.warning("Failed to send /search for \(search): \(error.localizedDescription)")
}
}
}

```

OK, now we really *are* finished with this project!

## Wrap up

This has been another epic project, but again you've learned some hugely important skills. SQL alone is a fundamental skill for any serious back-end web developer, and by completing this project you've taken the first step towards mastering it.

Also in this project you met **UUID** for the first time as part of a simple token-based authentication system. I've said it previously and I'll say it again: storing user data is fraught with risk, so it's important you take every step you can to secure it. Using HTTPS on all your sites is a good start, but using strong password encryption is important too. Adding tokens into the mix gives benefits such as distribution and revocation, which helps make the whole system better.

## Homework

We've implemented only a small part of a real microblogging system here, and one particular part is missing: the ability to follow other people so that you get a timeline. This is perfect for homework, because it has a number of interesting challenges:

1. Creating a new table to store follower data. You should store one "follow" per row: the username affected, and the username they are following.
2. Adding a "/follow/:username" route to add to the "followers" table, as well as an "/unfollow/:username" route to unfollow.
3. Adding a "/timeline" route to fetch a user's timeline, which should be a list of all posts by people they follow.

To help you accomplish that third task, take a look at this query:

```
SELECT * FROM `posts` WHERE `user` IN (SELECT `id` FROM `users`);
```

What you're seeing is a subselect: a **SELECT** statement used inside another **SELECT** statement. MySQL executes the inner one first:

```
SELECT `id` FROM `users`;
```

That produces a list of all the IDs in our "users" table, as a collection of rows containing a single value. Those rows then get fed into the second query, like this:

```
SELECT * FROM `posts` WHERE `user` IN (list_of_ids_here);
```

Now, obviously selecting all posts by all users is the same as just selecting all posts, but it gives you a start for your own query. You can write a query along the lines of this:

```
SELECT * FROM `posts` WHERE `user` IN (SELECT
`followed_username` FROM `followers` WHERE `follower_username` =
?);
```

In that example, **followed\_username** would be the username of the person you're following, whereas **follower\_username** would be your own username. That query means "show me all posts where the user is one of the name in my following list," which is exactly what you need.

# **Chapter 8**

## ASCII art

## Setting up

I have some good news: it's time for the second and last easy project. I realize learning a whole new language just to use databases is quite a brain drain, and – sorry! – we're going to be going into databases in more detail in the very next chapter. So, enjoy this brief respite while you can.

In this project, we're going to build an ASCII art generator. We'll be doing it over the web, but realistically you could call this from the command line or elsewhere quite easily. Plus, using the web gives me a chance to teach you another important skill: ajax.

I'll be explaining what ajax is soon enough, but first let's get the basics out of the way. Create a new Swift executable package called `project8`, and give it this `Package.swift`:

```
import PackageDescription

let package = Package(
    name: "project8",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
StencilTemplateEngine.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1),
        .Package(url: "https://github.com/twostraws/SwiftGD.git",
majorVersion: 1)
    ]
)
```

As you can see, that brings in my `SwiftGD` library for image manipulation so we can read image pixel data easily.

Run "`swift build`" to fetch all the dependencies, and optionally also run "`swift package generate-xcodeproj`" if you're using Xcode. Finally, open `main.swift` and give it this skeleton content:

```
import Foundation
import HeliumLogger
import Kitura
import KituraStencil
import LoggerAPI
import SwiftGD

HeliumLogger.use()
let router = Router()
router.setDefault(templateEngine: StencilTemplateEngine())

Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()
```

## Updating pages interactively

Our app is going to have two routes: "/" for the homepage, and "/fetch", which will fetch an image from the web and convert it to ASCII art. The code for the first route is trivial, so please add this now:

```
router.get("/") {
    request, response, next in
    defer { next() }

    try response.render("home", context: [:])
}
```

That just loads `home.stencil`, but for a change that template is going to contain some code of its own. Not Swift, of course – that would be both counter-productive and more than a bit repulsive. Instead, we're going to be writing a very small amount of JavaScript using the popular jQuery framework. jQuery comes bundled with Bootstrap as standard, but we aren't using Bootstrap here so we'll load jQuery ourself.

The `home.stencil` template will contain a form whether the user can enter the URL of an image to convert to ASCII art. But when they click the submit button, we're not just going to send the form off to Kitura just like we've done in our previous projects. While that might work well enough, a much more common approach is to use a technology called ajax, sometimes written as Ajax or even AJAX.

Ajax stands for "asynchronous JavaScript and XML," and what it means is that web browsers can fire a background request off to a server without leaving the current page, then process the results when they come back. If you use any webmail sites like Gmail or Hotmail, this is how they reload your inbox without having to refresh the page – they just fetch the information in the background then refresh. Avoiding the page reload is both faster and more resource efficient, and of course it also means the user can carry on using your site while the data is being fetched.

Although the X in Ajax stands for XML, the technique is used to fetch other types of data – not least plain text and JSON. In this project, we'll use jQuery's Ajax system to fetch the ASCII art

and display it on our page, without triggering a refresh.

Create a new "Views" folder in your project directory, then create main.stencil inside there. We're going to write it in two steps: the HTML and the JavaScript.

First, the HTML. This will add a text input where the user can enter an image URL, and a button to send that URL for processing. The button *won't* submit the form, though – and in fact we don't even need a form here because there's nothing to submit. Instead, it will call a JavaScript function called **submitURL()** that we'll write in a moment.

We're also going to add a **<pre>** tag, which is used to show unformatted text. I'm going to add a **style** attribute to it to make the text very small and use a fixed-width font so the ASCII art effect works well.

Add this content to home.stencil now:

```
<html>
<body>
<p>Enter an image URL: <input type="text" id="url" /></p>
<p><button onClick="submitURL();">Submit</button></p>

<pre id="output" style="font: 1px/1px monospace;">
</pre>
```

**STEP TWO GOES HERE!**

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

As you can see, the "STEP TWO GOES HERE!" part needs to be filled in. This is where we'll use JavaScript: we're going to write a **submitURL()** function that pulls out the value of the "url" text field and sends it off to the "/fetch" route.

To make this work, you're going to need to learn a small amount of JavaScript. Well, jQuery-flavored JavaScript, at any rate.

First, to read a value from the "url" field, we'll use this code:

```
var image = $("#" + url).val();
```

The `$(...)` syntax is jQuery in action, and `#url` means "find the document element that has the "id" attribute "url". The `val()` method is what pulls out the current value, which will be whatever the user typed in there.

Once we have a value, we can turn it into a request using the `$.get()` method. Yes, that's a period after the dollar sign. This takes three parameters: the URL to fetch, any data to send, and what to do when the response is received.

Parameter one is easy: `/fetch`. Parameter two is also easy, although the syntax might be new to you: we're going to use `{ url: image }`, which means "create a dictionary with the key `"url"` and the value being whatever is held in the `"image"` variable.

Parameter three is a little more tricky, because it's an inline function. These are a bit like closures, except they don't have any special capturing behavior. This function needs to accept the data that was received from the server as a parameter. All we're going to do is take that data, and send it straight into the `"output"` container, like this:

```
$("#output").text(data);
```

You've already seen the `$("#output")` syntax to find an element with a specific ID, but the `text()` method adjusts the contents of our `<pre>` to whatever the server sent back.

The full `$.get()` line looks like this:

```
$.get("/fetch", { url: image }, function(data) {
  $("#output").text(data);
});
```

Because we don't have Bootstrap this time, we need to import jQuery for ourselves. It's one of the most popular web frameworks in the world, so several companies host it free of charge on their own content delivery network. So, to pull in jQuery, we just need to add a line like this:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/
3.1.1/jquery.min.js"></script>
```

That's a copy that Google hosts free of charge – thanks, Google!

That's all the JavaScript you'll need for this project, so please add this to main.stencil in place of the "STEP TWO GOES HERE!" text:

```
<script>
function submitURL() {
    var url = $("#url").val();

    $.get("/fetch", { url: url }, function(data, status) {
        $("#output").text(data);
    });
}
</script>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/
3.1.1/jquery.min.js"></script>
```

Step one: complete!

## Reading remote images

jQuery's `.get()` call will load our "/fetch" route, passing in the URL that the user requested as a query parameter. We need to read that out, convert it to a valid **URL** object, then download it locally so we can read it. Once it's down, we can attempt to create a SwiftGD **Image** object out of it, but as you might imagine there are lots of steps here that might fail.

To make our code easy to follow, we're going to create a function to do everything above. It will return an **Image?**, so whatever code calls it can just check and unwrap one value, rather than worry about all the other problems points along the way.

Although it won't require a lot of code, this function is actually pretty packed with functionality. First, it's going to get the "url" query parameter and remove any percent encoding it finds, like this:

```
guard let imageFilename = request.queryParameters["url"] else
{ return nil }
guard let imageFilenameDecoded =
imageFilename.removingPercentEncoding else { return nil }
```

If either of those fail, the method returns nil because clearly there's something fundamentally wrong.

Next, it attempts to convert the user's URL into an actual **URL** object. There's a **URL(string:)** initializer we can use, but it returns nil if the conversion failed so again we'll exit if that goes wrong:

```
guard let url = URL(string: imageFilenameDecoded) else { return
nil }
```

If we have a valid URL, the next step is to download the image into a **Data** object. This is actually just one line of code in Swift, because **Data** has a **contentsOf:** initializer that accepts a remote URL. That connects to the remote server and fetches the content in full, automatically – it makes this whole task nice and easy!

The **contentsOf** initializer is a *throwing* function: it might fail if the content can't be

downloaded for some reason. So, we're going to call it using `try?`, which means "return nil if you fail, rather than throwing an error." That in turn means we can just unwrap it normally using `if let`, like this:

```
if let imageData = try? Data(contentsOf: url) {
```

If that succeeds, `imageData` will contain the binary data of the image the user requested. The next step is to save it to disk in a temporary location so that SwiftGD can read it. You can get a path to the user's temp directory using the `NSTemporaryDirectory()` function, so we're going to take that, append "input.png" to the end, and convert it to a file URL like this:

```
let temporaryName =
NSTemporaryDirectory().appending("input.png")
let temporaryURL = URL(fileURLWithPath: temporaryName)
```

Finally, we can save the image data to that temporary URL and pass it to SwiftGD to load as pixels, like this:

```
_ = try? imageData.write(to: temporaryURL)

if let image = Image(url: temporaryURL) {
    return image
}
```

As you can see, loading the image is also wrapped in `if let`, because the user might have tried to feed in a path to a movie or something else entirely.

That's the complete function broken down, so it's time to reassemble it into working code.

Please add this to main.swift, directly below the `import` lines:

```
func image(from request: RouterRequest) -> Image? {
    guard let imageFilename = request.queryParameters["url"]
    else { return nil }
    guard let imageFilenameDecoded =
imageFilename.removingPercentEncoding else { return nil }
```

```

        guard let url = URL(string: imageFilenameDecoded) else
        { return nil}

        if let imageData = try? Data(contentsOf: url) {
            let temporaryName =
NSTemporaryDirectory().appending("input.png")
            let temporaryURL = URL(fileURLWithPath: temporaryName)
            _ = try? imageData.write(to: temporaryURL)

            if let image = Image(url: temporaryURL) {
                return image
            }
        }

        return nil
    }
}

```

So, that new function will pull a URL out of our request, and either return a valid SwiftGD **Image** object or nil depending how things went. But that's only part of the problem: the most important part of our project is to read through the image data that was loaded and convert it to ASCII art.

If somehow you weren't already familiar, ASCII art is the process of drawing pictures out of ASCII letters, i.e. alphanumeric characters and other symbols. It's actually not that hard to do: we just need to loop over all the pixels in our image, and map the brightness of each pixel to one of these values: "@", "#", "\*", "+", ";", ":", ",", ".", "'", and " ".

Those values weren't chosen by accident – I chose them because they more or less reflect density from most dense (@) up to least dense (a space). This means darker pixels will use the denser ASCII characters so they appear darkest on the screen, whereas lighter pixels will use less-dense characters.

Let's crack on with writing the route. First, add this skeleton:

```
router.get("/fetch") {
```

```

request, response, next in
defer { next() }

// 1

// 2

// 3
}

```

We're going to replace those three numbered comment markers as we go.

The first thing this route needs to do is fetch the image the user requested. We already wrote the code for that, so we can just call `image(from: request)` to fetch it. Next, we're going to create an array called `asciiBlocks` that contains the 10 characters you already saw, so know how to map pixel brightnesses.

Next comes two sizes: the overall image size, and a block size. We're reading the image size once up front to avoid doing it again and again in a loop, which is never a bad idea. The block size is there so you can try adjusting how pixelated our output is. If you set `blockSize` to 1 it will draw every single pixel as an ASCII character, whereas if you set it to 10 it will draw only every tenth character.

Finally, at least for this first step, we're going to create a two-dimensional string array called `rows`. We're potentially going to be adding quite a few rows to that, so as a performance optimization we're going to call `reserveCapacity()` on it in order to allocate enough space up front.

Replace the `// 1` comment with this:

```

guard let image = image(from: request) else { return }
let asciiBlocks = ["@", "#", "*", "+", ";", ":", ",", ".", "^",
" "]
let imageSize = image.size
let blockSize = 2

```

```
var rows = [[String]]()
rows.reserveCapacity(imageSize.height)
```

The next step is the most interesting, because it's where the ASCII art conversion takes place. This is surprisingly easy to do because SwiftGD includes a `get(pixel:)` method, which returns the color of a pixel at a specific point. We can use that to loop over the height and width of an image, pulling out colors, and mapping them to our `asciiBlocks` array to draw the appropriate character.

The main chunk of the work is here:

```
let color = image.get(pixel: Point(x: x, y: y))
let brightness = color.redComponent + color.greenComponent +
color.blueComponent
let sum = Int(round(brightness * 3))
```

The first line retrieves the color at the X/Y coordinates provided, the second adds together the red, green, and blue components of the color, each ranging from 0 to 1, and the third line rounds that sum and converts it to an integer. You'll notice it also multiplies the sum by 3, which is no accident: there are 10 items in the `asciiBlocks` array, so we need to index into that using a number between 0 and 9. We have three color components ranging from 0 to 1, so multiplying their sum by 3 will give us a range of 0 to 9 – perfect.]

Replace the `// 2` comment with this:

```
// loop over the height of the image
for y in stride(from: 0, to: imageSize.height, by: blockSize) {
    // create a new row and reserve enough capacity for all our
    pixels
    var row = [String]()
    row.reserveCapacity(imageSize.width)

    // loop over the width of the image
    for x in stride(from: 0, to: imageSize.width, by: blockSize)
```

```

{
    // get the pixel at the current location
    let color = image.get(pixel: Point(x: x, y: y))

    // figure out its brightness
    let brightness = color.redComponent +
color.greenComponent + color.blueComponent

    // multiply by three then round to an integer
    let sum = Int(round(brightness * 3))

    // map that brightness to an ASCII character
    row.append(asciiBlocks[sum])
}

// append this to our result
rows.append(row)
}

```

The last step is nice and easy: we need to convert that two-dimensional array into a single string, separated using spaces and line breaks.

To do that, we're going to use two new methods: **reduce()** and **joined()**. The **joined()** method lets us stitch an array together with a separator we specify, which in our case will be an empty space. We'll use this on each row, turning something like `[ "+", "*", "#"]` into `"+ * #"`, then add a line break to the end. To write a line break you don't just press return inside a string – that would cause all sorts of build problems. Instead, you write `\n` inside your string, which is a special sequence that becomes a line break.

The **reduce()** method converts an array of items into a single item by running a closure repeatedly. In our case, each time the closure runs we're going to add the result of our **joined()** call to the results of previous **joined()** calls, building one big string of all the rows.

Do you remember shorthand closure syntax – the use of **\$0** to represent values inside closures?

That exists in `reduce()` too: `$0` represents "the result of all the values computed so far" and `$1` represents "the next value you want to add to the result." So, we can use them like this:

```
$0 + $1.joined(separator: " ") + "\n"
```

In that code, `$0` will be a string containing all the rows converted to ASCII so far, and `$1` will be the current row that needs to be converted to a string. The first time `reduce()` runs there is no previous value because nothing has happened yet, so we need to supply a parameter to be used as the initial value. For us, that's an empty string.

Once we have one big string representing the joined, reduced version of our two-dimensional array, we just need to send it back to the client. That will trigger the completion function of the Ajax `$.get()` request, and update the UI.

Add this code in place of the `// 3` comment:

```
let output = rows.reduce("") {
    $0 + $1.joined(separator: " ") + "\n"
}

try response.send(output).end()
```

That's the complete route, so build and run the app, then try sending it a URL to an image. Something like [http://apache.org/img/asf\\_logo.png](http://apache.org/img/asf_logo.png) makes for interesting ASCII art because it combines an image with fine text. Try adjusting the `blockSize` variable and seeing how it affects the result.

## Wrap up

Even though this was an easy project, I hope you learned something along the way – not least about Ajax. I realize that adding *another* language to your arsenal is probably starting to sound quite unappealing at this point, but it's an unavoidable fact that the internet is a heterogenous environment: things like HTML, CSS, JavaScript, and SQL are part of the ecosystem, and if you want to be a player you've got to learn the game.

In this project you got a brief taster of ajax as a method of sending and receiving data without forcing a page reload. From a server-side Swift perspective, ajax calls work just like any other route – you perform your calculations, then send back a result. jQuery has a special `$.getJSON()` function that performs an ajax call and evaluates the result as JSON, making it easy to receive structured data for your site. What you do with it is then down to you, but I'm afraid it *does* involve writing more JavaScript.

That's the last of the two easy projects in the book, so I'm afraid your little holiday is about to come to a crashing end as we head back into the murky world of SQL...

## Homework

Looking to avoid SQL for another few hours? Here are some suggested tasks you can try and code to make this project better.

First, no matter whether the user provides a URL to a PNG or a JPEG, we save it using the same filename: `input.png`. Can you update that code so that it uses `input.jpg` or `input.png` depending on the type? And what happens if that file already exists?

Second, take a good look at this line of code:

```
row.reserveCapacity(imageSize.width)
```

That ensures the `row` array has enough capacity for the full width of the image. But what about our `blockSize` value? See if you can adjust this code so that it reserves less space.

Third, speaking of `blockSize`, wouldn't it be neat if the user could specify a block size alongside their URL?

Finally, take a look at this line of code:

```
let brightness = color.redComponent + color.greenComponent +
color.blueComponent
```

That calculates a brightness from 0 to 3 by adding together the red, green, and blue components equally. It works, but it's naïve: our eyes are significantly more sensitive to the color green than they are to red, and even more so compared to blue. A much better brightness calculation is this:

- 0.299 times the red component
- 0.587 times the green component
- 0.114 times the blue component

That adds up to 1, so you'll need to adjust the multiplication when you've finished.

# **Chapter 9**

## Databases

## Setting up

We've been using fairly simple SQL so far, but it's time to take it up a notch. I'm not doing this because I enjoy torturing you: you're going to learn important things that will allow you to build more powerful, more efficient databases, and I've done my best to keep it as brief as possible.

The techniques you'll be learning are:

- Indexing, to make your searches faster
- Normalization, to store data efficiently
- Referential integrity, to keep your data organized
- Transactions, to keep your data consistency
- Default values, to avoid missing data

We'll be covering all of these entirely inside the MySQL monitor app – you don't need to worry about Xcode for this project.

So, connect to the MySQL monitor using the "mysql -u root" command on Linux, or by running "/usr/local/Cellar/mysql/5.7.16/bin/mysql -u root" on macOS. Then run **use swift** to open the database we created earlier.

## Indexing for performance

When you use SQL to run a query that filters results, e.g. `WHERE user = 'twostraws'`, MySQL needs to examine every row to see whether it matches your criteria. This is slow when you have many rows: a million rows means a million checks, which can cause your apps to take significantly longer to execute.

The SQL solution is indexing, which lets you mark certain fields as being important for searching. If you say that the "user" field in the "posts" table should be indexed, it will cause MySQL to read all those values into a special cache that can be searched far more efficiently. So when you say "show me all the posts by twostraws," MySQL already has that answer saved in its index and can perform the query in a fraction of a second.

The cost of creating an index must not be ignored: every time you add, update, or delete rows, MySQL needs to update its indexes. However, most databases spend far more time reading data than writing data, so indexes are usually worth the effort.

Let's try indexes now. We're going to create a test table, then fill it with random numbers. MySQL doesn't have a nice way of doing this, so we're going to cheat. First, run this query to create the "test\_numbers" table:

```
CREATE TABLE `test_numbers`(`number` INT);
```

As you can see, it holds one field of information, an integer called "number". There's no magic command to fill that with random data, but we can do it by repeating a query a few times.

First, run this:

```
INSERT INTO `test_numbers` VALUES (RAND() * 1000000);
```

The `RAND()` function returns a random number between 0 and 1. We're multiplying that by one million, so we'll end up with a number between 0 and 1,000,000.

Now I'd like you to run this command:

```
INSERT IGNORE INTO `test_numbers` SELECT `number` * RAND() FROM `test_numbers`;
```

That looks like two queries in one, but what it does is run a **SELECT** then pipe that directly into an **INSERT**. The **SELECT** part pulls out all the numbers in the table and multiplies them by **RAND()** to create new numbers. The **INSERT** part takes the result of the **SELECT** – a range of new random numbers – then adds them to the table.

When you run that query, you'll see this output:

```
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

That means 1 new row of data was inserted. Now press the up cursor key on your keyboard to re-load the command, and press return to run it again. This time you'll see 2 rows inserted, because there were 2 rows previously. Run it again and you'll get 4 rows, then 8 rows, then 16, 32, 64, 128, and so on.

I'd like you to keep running the command until you see the output "Query OK, 4194304 rows affected". The query will execute slightly slower each time you run it, because more work is being done. That means 4,194,304 new rows were inserted, giving us 8,388,608 rows in total – over eight million. That's not a massive data set in all honesty, but it's enough to prove the need for indexes.

Each time that query was run, it took all the numbers in the table, multiplied them by a random number between 0 and 1, and re-inserted the results. Over time, that meant the numbers became progressively smaller, because they would always be less than their input. So, even though we have 8 million numbers that started off as high as a million, only a small number will be greater than 1000.

To demonstrate this, try running this query:

```
SELECT COUNT(*) FROM `test_numbers` WHERE `number` > 1000;
```

Your answer will be different to mine because of randomness, and the amount of time your query takes will depend on the performance of your machine. I got the result 797,916, and it took 22.20 seconds to calculate.

Think about that: 22 seconds just to count how many rows matched a simple query. Eight million rows might seem like a lot to you, but in terms of databases it's quite average. Worse, we're not even performing a complicated query: integer comparison is fast compared to string comparison, and our **WHERE** clause only checks one field.

The reason for the slowness is because MySQL has to perform what's called a full table scan: it has to go over every single row to do its comparison. If we add an index – if we tell MySQL that **number** is something we want to search for – then it can check all the rows ahead of time and store their values in an optimized way.

Run this command:

```
ALTER TABLE `test_numbers` ADD INDEX (`number`);
```

That tells MySQL to read all the **number** values into its index (its value cache) for faster searching. My computer took 31 seconds to create that index, but that's a price you only pay once.

Now try running the **SELECT** query again:

```
SELECT COUNT(*) FROM `test_numbers` WHERE `number` > 1000;
```

This time MySQL responded with the same result after just 0.22 seconds. That took 22.20 seconds without the index, so just making that one small change made the query run 100 times faster – it's gone from “we can't use that in production” to “of course we can do that.”

You can index any kind of data in MySQL. We used integers here, but it works just as well with strings or dates. When you use a primary key, like we did with “id”, MySQL automatically indexes it. As you saw, it takes time to create indexes, but it also takes a small amount of time to update them so if you over-index your data you might end up causing more problems than you solve.

**Warning:** It's common to see newcomers indexing too much data, or, worse, indexing the wrong fields entirely. Unless you know what you're doing, it's best to hold off on indexes until you have an idea of the performance hotspots!

# Normalization for efficiency

Consider the following table, designed to store customer purchases:

```
CREATE TABLE `purchases` (
    `id` INT PRIMARY KEY AUTO_INCREMENT,
    `item` VARCHAR(255),
    `customer_name` VARCHAR(255),
    `address` VARCHAR(255),
    `city` VARCHAR(255),
    `zip` VARCHAR(255),
    `country` VARCHAR(255)
);
```

As you can see, there's a field in there for "item", which will store something like "Hacking with macOS", and then there are fields for the customer's name and delivery address.

If the same customer purchased 10 different things, that would be 10 rows in the table. However, it would also mean 10 rows containing identical data: the customer's name, address, city, zip code, and country. That's hugely wasteful, and will cause your database to run more slowly than it ought to. And what happens if the item name changes? Either we update every row with the new name, or we have some rows with the new name and some with the old.

The solution is called normalization, and it's the process of splitting one table up into two, three, or even more. In the case of the "purchases" table, we can split it up into several others:

1. We could create a "items" table that contains an ID and item name. The "purchases" table could then store an item ID, so if the name changes it doesn't matter.
2. We could create a "customers" table that contains an ID and a customer name. The "purchases" table then points at that customer ID, so we can run queries such as "show me all purchases by customer 12345."
3. We could create a "delivery\_addresses" table that contains a customer ID (pointing to the "customers" table), as well as all their address information. The ID number of a specific delivery address would be stored in "purchases", which allows customers to have more than one delivery address on file.

The end result would be SQL something like this:

```
CREATE TABLE `purchases` (
  `id` INT PRIMARY KEY AUTO_INCREMENT,
  `item_id` INT,
  `customer_id` INT,
  `delivery_address` INT
);

CREATE TABLE `items` (
  `id` INT PRIMARY KEY AUTO_INCREMENT,
  `name` VARCHAR(255)
);

CREATE TABLE `customers` (
  `id` INT PRIMARY KEY AUTO_INCREMENT,
  `name` VARCHAR(255)
);

CREATE TABLE `delivery_addresses` (
  `id` INT PRIMARY KEY AUTO_INCREMENT,
  `customer_id` INT,
  `address` VARCHAR(255),
  `city` VARCHAR(255),
  `zip` VARCHAR(255),
  `country` VARCHAR(255)
);
```

Using this approach, an entry in the “purchases” table contains the ID number of an item, the ID number of a customer, and the ID number of a delivery address – there’s no more duplication.

Now, this creates a bit of an issue: when all the data is split up, how can you read it back

sensibly? Try adding some test data:

```
INSERT INTO `purchases` VALUES (1, 1, 1, 1);
INSERT INTO `items` VALUES (1, "Hacking with macOS");
INSERT INTO `customers` VALUES (1, "Haul Pudson");
INSERT INTO `delivery_addresses` VALUES (1, 1, "Address",
"City", "Zip", "Country");
```

With the data for an individual purchase split across four tables, how can you write a query to say “find purchase number 1, and show what was bought, who bought it, and what was delivered?”

You could run individual queries to fetch each piece of data, but that soon becomes inefficient when dealing with lots of rows. A better option is to use queries that read from multiple tables at once, telling MySQL how the data matches up.

Let’s start simple. Run this query:

```
SELECT `item_id`, `customer_id`, `delivery_address` FROM
`purchases` WHERE `id` = 1;
```

That will put out the item ID, customer ID, and delivery address from our sole row in the “purchases” table. We can tell MySQL we want to read the “name” field from the “items” table for whichever item has the same ID number that was stored in “item\_id”.

To do this, we can write both table names in the **FROM** clause of our **SELECT**. However, there’s a catch: both tables have an “id” field, so when we say we want to match ID number 1 we need to be specific: we want to match ID number 1 in the “purchases” table, and match it against the *items* ID that correlates to the “item\_id” field. If a field name is used in only one of the tables, you don’t need to be specific – MySQL can figure it out.

Try this query:

```
SELECT `item_id`, `name`, `customer_id`, `delivery_address`
FROM `purchases`, `items` WHERE `purchases`.`id` = 1 AND
`items`.`id` = `item_id`;
```

Notice how I'm selecting "item\_id" and "name" in the same query, even though they come from different tables. Because we're matching them up correctly in the **WHERE** clause, MySQL understands exactly what we mean.

When working with multiple tables, it's common to give your tables aliases. These are usually one to three letter nicknames that apply for a single query, to make them easier to read and write. For example, we could rewrite the previous query like this:

```
SELECT `item_id`, `name`, `customer_id`, `delivery_address`  
FROM `purchases` p, `items` i WHERE p.`id` = 1 AND i.`id` =  
`item_id`;
```

That gives "purchases" the alias "p", and "items" the alias "i".

You can use these aliases in the field names you select if they also have name collisions. To try this out, let's take our query further: we want to know the customer name as well. That's stored in the "name" field of "customers", but we're also selecting the "name" field of "items". As a result, we need to be specific: we want the "name" field from the "items" table as well as the "name" field from the customers" table.

This makes our query even longer, so take a deep breath and try this out:

```
SELECT `item_id`, i.`name`, `customer_id`, c.`name`,  
`delivery_address` FROM `purchases` p, `items` i, `customers` c  
WHERE p.`id` = 1 AND i.`id` = `item_id` AND c.`id` =  
`customer_id`;
```

Of course, the logical conclusion to all this is trying to add the delivery information too. Try and wrap your brain around this one:

```
SELECT `item_id`, i.`name`, p.`customer_id`, c.`name`,  
`delivery_address`, d.`address`, d.`city`, d.`zip`, d.`country`  
FROM `purchases` p, `items` i, `customers` c,  
delivery_addresses d WHERE p.`id` = 3 AND i.`id` = `item_id`
```

```
AND c.`id` = p.`customer_id` AND d.`id` = p.`delivery_address`;
```

Now, if it's any consolation I should say I have never seen an SQL query that joins four tables simultaneously. If you lost your way here or in one of the earlier queries, try going back to the first simple example – all we did was add more fields with each query.

There's one more thing to say before we move on, which is that normalization is computationally expensive. Running a query against the original table – with all its duplication – is probably faster than running the final query, which reads across four tables. Broadly speaking normalization is a good thing that you should use to organize your data efficiently, but if something is performance critical don't think twice about “denormalizing” – intentionally duplicating small amounts of data to avoid expensive queries.

## Referential integrity for organization

When you use normalization, an ID number in one table points to a full row of data in another. This is important for storing your data intelligently, but it causes a problem: when one piece of data relies on another, what happens when you delete something?

To try this out, open the 0.sql from project 7, select all the text, copy it to your clipboard, then paste it into MySQL. This will delete and recreate the tables for our microblogging platform.

Now run this query:

```
SELECT * FROM `posts`;
```

You'll see three from "twostraws", two from "adele", and one from "taylorswift13".

Now run this:

```
DELETE FROM `users` WHERE `id` = 'twostraws';
```

That deletes the "twostraws" user, so now repeat the **SELECT** query again:

```
SELECT * FROM `posts`;
```

You'll see the posts for "twostraws" are still there, even though we deleted that user. This is the kind of data hole that can easily cause problems in your apps, so SQL gives us a way of linking the data explicitly so that if one item goes anything that references it always goes.

This link is called a "foreign key" – a field in one table that points to a primary key in another. Once you tell MySQL that field A in table 1 field B in table 2, it can take action on your behalf to ensure no data is left behind by accident.

In our case, we want to alter the "posts" table so that its "user" field is reliant on a matching "id" value in the "users" table. If the user gets deleted or updated, we want those changes to be reflected in the "posts" table. The "reflecting of changes" is called *cascading* in SQL, and the SQL command to set up that link is this:

```
ALTER TABLE `posts` ADD CONSTRAINT FOREIGN KEY (`user`)
```

```
REFERENCES `users`(`id`) ON UPDATE CASCADE ON DELETE CASCADE;
```

Try copy and pasting 0.sql into MySQL again, to reset the “users” and “posts” table. Run this:

```
SELECT * FROM `users`
SELECT * FROM `posts`;
DELETE FROM `users` WHERE `id` = 'twostraws';
SELECT * FROM `posts`;
```

This time you’ll see that deleting the user “twostraws” has also deleted their posts – perfect!

## Transactions for consistency

All the SQL queries we've been running so far have run in isolation: we run one, get the results, run another, get those results, and so on. That works well enough for basic apps, but often your queries depend on each other: update row 1, then update row 2, then delete row 3, then select the results. What happens if the first two queries work well, but the third one goes wrong for some reason? The answer is that your database is in an inconsistent state: it's not like it was before you started, and it's not where it *ought* to be once you've finished.

MySQL's solution for this is called *transactions*. Once you start a transaction, you can run as many queries as you want, but they only have a permanent effect if you commit the transaction at the end. If you decide part-way through that things are going sour, you can *rollback* the transaction – effectively reverting all the changes you would have made – without causing any harm.

As you might imagine, transactions are a basic requirement in any industry where data must be flawless, not least the banking sector. MySQL guarantees that your transactions are *atomic*, which is a fancy way of saying “all or nothing”. That is, either all the queries in your transaction will be executed successfully or none of them will – even if you switch off your server in the middle of it all.

To use transactions in the MySQL monitor app, you need to know three commands: **START TRANSACTION**, **COMMIT**, and **ROLLBACK**. The first one starts a transaction, the second makes all queries from a transaction permanent, and the third erases all the queries from a transaction as if they never happened.

While a transaction is in progress – i.e., in between **START TRANSACTION** and either **COMMIT** or **ROLLBACK** – any **SELECT** queries you make will show your data as if it had changed. So if you delete some rows inside a transaction then try to select them, they will appear deleted even though the transaction hasn't been committed yet. MySQL even handles auto-incrementing numbers correctly, ensuring they don't get re-used even if a transaction is rolled back.

Let's try it now. Run these queries:

```
START TRANSACTION;
```

```
SELECT * FROM `posts`;
DELETE FROM `posts`;
SELECT * FROM `posts`;
ROLLBACK;
SELECT * FROM `posts`;
```

That runs the same **SELECT \*** three times. The first time you'll see lots of posts, the second time you'll see none, and the third time you'll see them all back again because the transaction was rolled back.

Unless you're just playing around, I strongly recommend you use transactions everywhere in your web apps. They keep your data safe, they keep your data consistent, and they actually perform faster too because MySQL can bulk execute its writes.

In my testing, the best way to start a transaction in Swift was to tell MySQL to stop auto-committing its changes, like this:

```
try db.execute("SET autocommit=0;", [], connection)
```

Then you can go ahead and run as many queries as you want. To rollback or commit a transaction, use one of these two once you're done:

```
try db.execute("ROLLBACK;", [], connection)
try db.execute("COMMIT;", [], connection)
```

Note: if you start a transaction then don't commit it before your route ends, it will automatically be rolled back. This is a common source of programmer confusion, because you'll think everything worked but your data will disappear!

## Default values for safety

Database fields have types like **VARCHAR(255)** or **INT**, but they can also be **NULL**: they can have no value at all. This is a bit like Swift's optionals, except without the safety – MySQL lets you use null values freely, and, quite frankly, I wish you the best of luck.

Try running this query:

```
SELECT 1 < NULL, 1 > NULL, NULL = NULL, NULL != NULL;
```

MySQL will happily tell you that “`1 < NULL`” is null, “`1 > NULL`” is null, “`NULL = NULL`” is null, and “`NULL != NULL`” is null. In short, null is basically impossible to use in any meaningful way – it’s not greater than, less than, equal to, or even unequal to anything else, it’s just null.

MySQL automatically assigns null to a field if you insert a row without giving that field a value. To avoid having null in your data, you have two options.

1. Declare the field as being **NOT NULL**, which means a value must be specified otherwise the **INSERT** query will fail.
2. Provide your own default value to be used in place of **NULL**.

Where possible, I recommend you do both: make values **NOT NULL** and provide sensible defaults if they exist.

To give you an example, here’s the existing definition for the **posts** table, as taken from 0.sql:

```
CREATE TABLE `posts` (
  `id` INT PRIMARY KEY AUTO_INCREMENT,
  `user` VARCHAR(64),
  `message` VARCHAR(140),
  `parent` INT,
  `date` DATETIME
);
```

It seems clear that “user” and “message” must be required fields, so we can declare them **NOT NULL** and force them to be entered. There’s no point giving them a default value, though, because there’s nothing sensible.

As for “parent”, that can be **NOT NULL**, and we can give it a default value of 0 – that is, every message is a new one, rather than a reply, unless we specify otherwise. Finally, there’s “date”. This can also be **NOT NULL**, but we can specify the value **CURRENT\_TIMESTAMP** for its default, which is the equivalent of using **NOW()** when adding a row.

So, an improved table definition would be this:

```
CREATE TABLE `posts` (
  `id` INT PRIMARY KEY AUTO_INCREMENT,
  `user` VARCHAR(64) NOT NULL,
  `message` VARCHAR(140) NOT NULL,
  `parent` INT NOT NULL DEFAULT 0,
  `date` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

## Wrap up

You're still a long way from being a database expert, but I think you're at least edging your way out of "newbie" status. The next two projects both use databases – one CouchDB, one MySQL – so you have more than enough time to practice!

If you have ever trouble diagnosing a slow **SELECT** query, try putting the **EXPLAIN** keyword before it inside the MySQL monitor app. This will ask MySQL to tell you how it intends to search your data: what indexes it will use, how many rows it needs to check, and so on. This can sometimes highlight problems, such as the need for an extra index.

One last tip: although MySQL is awesome at handling international characters, many systems don't come with that support enabled. To see what character set your tables are using, run the **SHOW TABLE STATUS** command and look under "Collation". If you see something like "latin1\_swedish\_ci" then you're going to have a hard time handling things like Japanese characters.

Given how great Swift is with Unicode, you're almost certainly going to want the same quality level in MySQL. So, I recommend you always explicitly create your tables using the "utf8mb4" character set, which supports the full range of UTF-8 characters.

It's not complicated to do, but at the same time you're almost certainly not going to remember this one by heart. What you need to do is paste the following text after the closing parenthesis of your table definitions:

```
CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

For example:

```
CREATE TABLE `users` (
  `id` VARCHAR(64) PRIMARY KEY,
  `password` VARCHAR(128) NOT NULL,
  `salt` VARCHAR(128) NOT NULL
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

# **Chapter 10**

## Instant Coder

## Setting up

In this project we're going to build Instant Coder, which is a bit like Uber for coders. You can register on the site, tell it what you're preferred language is, then others can come in and search for developers who can help them on their project.

This project is going to bring together lots of things you've learned so far: CouchDB and views, static file serving, sessions, form submission, Stencil templates, and more. And of course you're also going to learn something new, because we're going to be using the Kitura-Credentials framework for authenticating users against a third-party service.

Kitura-Credentials is an umbrella framework that handles authentication with a number of services. Kitura comes with Facebook, Google, and GitHub built in, but you can add your own easily enough. Given the nature of this project – Instant Coder – we'll be using GitHub, which means that users will be able to authenticate on our site using their GitHub credentials.

There's a lot to do in this project, so let's dive right in: create a new executable package called project 10, then give it this Package.swift:

```
import PackageDescription

let package = Package(
    name: "project10",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
CouchDB.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
CredentialsGitHub", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
StencilTemplateEngine.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1)
    ]
)
```

You'll see "Kitura-CredentialsGitHub" is in there for the first time; that's the part that handles GitHub authentication for us.

In the project files for this book, look for the "project10-files" and directory and copy its "public", "Views", and "Working" folders. "public" just contains the usual Bootstrap files, "Views" contains the master template we'll be building on, and "Working" contains some CouchDB JSON we'll get to in a moment. When that's done, run "swift build" to fetch the dependencies, and also "swift package generate-xcodeproj" if you're using Xcode.

I want to focus on what's new and interesting in this project, so let's get all the set up out of the way up front. This is a complex project, so there's quite a lot of setting up to do. First, add this skeleton code:

```
import CouchDB
import Credentials
import CredentialsGitHub
import Foundation
import HeliumLogger
import Kitura
import KituraNet
import KituraSession
import KituraStencil
import LoggerAPI
import SwiftyJSON

HeliumLogger.use()

let router = Router()
router.setDefault(templateEngine: StencilTemplateEngine())
router.all("/static", middleware: StaticFileServer())
router.all(middleware: Session(secret: "He thrusts his fists
against the posts and still insists he sees the ghosts"))
router.post("/", middleware: BodyParser())
```

```
Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()
```

There are a couple of new **import** lines in there for Credentials and CredentialsGitHub, but otherwise it's all code you've seen before.

We also need to pull in three functions we've used extensively before, so you can copy and paste these from a previous project: **removingHTMLEncoding()**, **getPost()**, and **send(error:)**. Add these below the **import** lines:

```
extension String {
    func removingHTMLEncoding() -> String {
        let result = self.replacingOccurrences(of: "+", with: " ")
        return result.removingPercentEncoding ?? result
    }
}

func getPost(for request: RouterRequest, fields: [String]) -> [String: String]? {
    guard let values = request.body else { return nil }
    guard case .urlEncoded(let body) = values else { return nil }

    var result = [String: String]()

    for field in fields {
        if let value =
body[field]?.trimmingCharacters(in: .whitespacesAndNewlines) {
            if value.characters.count > 0 {
                result[field] = value.removingHTMLEncoding()
                continue
            }
        }
    }
}
```

```

        return nil
    }

    return result
}

func send(error: String, code: HTTPStatusCode, to response: RouterResponse) {
    _ = try? response.status(code).send(error).end()
}

```

You're probably sick of seeing them, but don't worry – that's the last time we'll be pasting them into main.swift, because in the very next project you'll learn how organize your code more neatly.

There are a few more small tasks before we can start the real coding in this project. First, we need a CouchDB database to work with, so please start CouchDB then run this Curl command from your terminal:

```
curl -X PUT localhost:5984/instantcoder
```

Now that we have a database in place, we can inject the CouchDB design document that describes the views we need for this app. I'll be explaining what they are during the project, but for now I'd like you just to load them into CouchDB with these two commands:

```
cd Working
curl -d @views.json -H "Content-type: application/json" -X PUT
localhost:5984/instantcoder/_design/instantcoder
```

That's it – all the dull stuff is done. From here on it's new code for this project. Let's go!

## Say hello with GitHub

Kitura-Credentials does not have a complicated approach to authentication, but it might require you rewire your brain a little because it can feel a bit like you’re giving up some control. It works as middleware, just like **BodyParser** and **StaticFileServer**: you tell it which directories require authentication and what service you want to use, and it does the rest.

Before we can write any code, we need to register an app with GitHub app so that we have authentication credentials to use. Log in to your GitHub account, or create one if for some reason you don’t already have one. Now browse to <https://github.com/settings/developers> and click the “Register a new application” button.

You’ll see various fields to fill in, so use this:

- Application name: Instant Coder
- Homepage URL: Whatever your personal website is
- Application description: Skip this
- Authorization callback URL: <http://localhost:8090/login/github/callback>

The callback URL is where we want GitHub to redirect our users once they choose to allow or deny authentication.

When you click Register Application you’ll be taken to your application’s settings screen on GitHub, and you should see two important pieces of information: Client ID and Client Secret. Both of these are long hexadecimal strings that you’ll need to place into your Swift code in a moment, identifying your app uniquely.

Now that we have a valid GitHub app to authenticate against, the next step is to decide which routes we want to protect. We’re going to be creating seven routes in this project:

- “/”: the homepage will just render a welcome message using Stencil.
- “/signup”: used when new users come to the site and we need to create an account for them.
- “/projects/all”: shows all coding projects in the system
- “/projects/mine”: shows all *our* coding projects, allowing us to delete any we don’t want.

- “/projects/search”: searches for coders who can use a certain language.
- “/projects/new”: creates a new project
- “/projects/delete/:id/:rev”: deletes a project

We can break those URLs down into three groups. First, “/” is available to everyone, whether or not they have authenticated. Second, “/signup” is available to users who have authenticated using GitHub but haven’t created an Instant Coder account yet. Third, all the other pages are available only to users who have an Instant Coder account. Because most of our pages are somewhere under the “/project” route, it’s enough to ask Kitura-Credentials to secure “/signup” and “/project”.

Now, you might think “why do we need a user account – can’t we just use GitHub?” It’s true that you could just use GitHub for your account information, but only if you didn’t want to store any sort of extra information about your users. In this project, everyone who comes to the site will be asked to name their preferred coding language so we can create some search functionality. As a result, we’re going to create an Instant Coder account for them when they authenticate for the first time.

Adding credentials support to your app is done in three steps:

1. Creating an instance of **CredentialsGitHub** and registering it with Kitura-Credentials. This is where our client ID and secret are used.
2. Assigning routes for the GitHub credentials system to use to trigger authentication. We don’t provide the code for these routes, they are handled by Kitura.
3. Telling our router to attach the new credentials to any of our own routes we want to secure.

We’ve already added lots of skeleton code to our app, so I’d like you to make some space between these two lines:

```
router.post("/", middleware: BodyParser())
Kitura.addHTTPServer(onPort: 8090, with: router)
```

First, creating our GitHub credentials and registering them. Add these lines just below the

**BodyParser()** line:

```
let credentials = Credentials()
let gitCredentials = CredentialsGitHub(clientId: "YOUR CLIENT
ID HERE", clientSecret: "YOUR CLIENT SECRET HERE", callbackUrl:
"http://localhost:8090/login/github/callback", userAgent:
"server-side-swift")
credentials.register(plugin: gitCredentials)
```

As you might imagine, you need to insert your client ID and secret in the appropriate places. That prepares Kitura to work with authentication, but we still need some routes that actually trigger the process of working with GitHub. That's step 2: assigning routes for GitHubCredentials to work.

Most of the routes we've made so far have been handled with our own code, with **StaticFileServer** being the notable exception. With static files, we just point it at a route such as "/static" and let it handle everything else for us. Kitura-Credentials works the same way: we give it a route for handling GitHub communication, and it does the route.

We need to give Kitura-Credentials two routes: one to start the authentication process, and one to be called by GitHub once the user has granted approval to our app. We don't need to attach any code to these routes, we just tell our router that they are handled by the credentials object we configured. If authentication fails for some reason, we're going to redirect the user back to "/login/github" so they can try again.

Add these three lines of code below the credentials code you just added:

```
router.get("/login/github", handler:
credentials.authenticate(credentialsType: gitCredentials.name))
router.get("/login/github/callback", handler:
credentials.authenticate(credentialsType: gitCredentials.name))
credentials.options["failureRedirect"] = "/login/github"
```

The last step is to tell the router to use our **credentials** object for the "/projects" and "/signup" routes. We haven't written these yet but that's OK – add these two lines now:

```
router.all("/projects", middleware: credentials)
router.all("/signup", middleware: credentials)
```

Despite not having written those routes yet, we already have enough to begin the GitHub authorization flow for real. Build and run your project, then point your browser at <http://localhost:8090/login/github> – you should get redirected to <https://github.com/login/oauth/authorize> where you’re asked to authorize the application. **Don’t click that button just yet!**

When you click “Authorize application” (which, again, I really don’t want you to click just yet!) GitHub will redirect you back to “/login/github/callback”, which is handled by Kitura-Credentials. It detects that authorization has succeeded, and stores some information in the session so GitHub won’t be bothered until the next time the user comes to your site.

Once we’ve added some actual routes to our app we can test out authorization for real. For now, though, just close the GitHub authorization window.

# Rendering the basic UI

Our users can be in one of three states:

1. They are new to the site, so they have no account and have not authenticated with GitHub.
2. They have authenticated with GitHub, but haven't created an Instant Coder account yet.
3. They have authenticated with GitHub and have created an Instant Coder account.

The first of those is easy enough to handle, because we can just render some HTML telling them to log in.

Back in project we added a `context(for:)` function that set up some basic Stencil context for all our pages. We need the same thing here, but this time we're not going to read the current username from the user's session. Instead, we're going to use `request.userProfile` – a struct given to us by Kitura-Credentials, which will contain our user's ID number and display name. On GitHub, your user ID is an integer (mine is 190200), and your display name is your name how you'd like it to be written (mine is “Paul Hudson”).

We're also going to use this method to set an array of languages we want to use in the app. This will be used in Stencil to print out a list of languages for users to choose from, rather than hardcoding each language in the templates.

Add this function along with the others, just before the `HeliumLogger.use()` line:

```
func context(for request: RouterRequest) -> [String: Any] {
    var result = [String: Any]()
    result["username"] = request.userProfile?.displayName
    result["languages"] = ["C", "C++", "C#", "Go", "Java",
    "JavaScript", "Objective-C", "Perl", "PHP", "Python", "Ruby",
    "Swift"]

    return result
}
```

With that, we can write the first of our own routes, “/“. This will set up some basic context, then pass rendering off to home.stencil. Add this route directly before the call to

```
Kitura.addHTTPServer():
```

```
router.get("/") {
    request, response, next in
    defer { next() }

    var pageContext = context(for: request)
    pageContext["page_home"] = true

    try response.render("home", context: pageContext)
}
```

Now, one part of that bears further explanation: the `pageContext["page_home"]` context variable being set to true. This is in there because the master.stencil file I gave you is more interesting than our previous templates. Here’s what’s new:

```
<div class="container">
    <ul class="nav nav-tabs">
        {%
            if not username %}
            <li role="presentation" {%
                if page_home %}
                    class="active" {%
                        endif %}><a href="/">Home</a></li>
                {%
                    else %}
                    <li role="presentation" {%
                        if page_projects_mine %}
                        class="active" {%
                            endif %}><a href="/projects/mine">My
                    Projects</a></li>
                    ...
                    {%
                        endif %}
                </ul>
                <div class="main-container">
                    {%
                        block body %}{%
                        endblock %}
                </div>
            </div>
```

I've put a “...” in there because there's more of the same. That's the only part of master.stencil that's different from previous projects, and what it does is create a tab UI across the top of our pages to help make navigation easier. If the user isn't logged in they'll only see the first tab; if they *are* logged in they'll see four other tabs.

All this matters because Bootstrap has a dedicated “active” class for the tab that is currently selected. Using `{% if %}` conditions I've made it so that we can draw a tab as active just by passing a variable into the Stencil context – “page\_home” in the case of the homepage. This means individual pages don't need to worry about drawing their tab, it all gets handled by master.stencil.

The home.stencil file itself just fills in the **title** and **body** blocks, as before. However, it also includes a button linking the user to the “/projects/mine” route. This is behind our Kitura-Credentials wall, which means clicking that link will kick off the authentication process.

Add this content to home.stencil in the Views directory:

```
{% extends "master.stencil" %}

{% block title %}Welcome{% endblock %}

{% block body %}
<h1>Code yourself happy</h1>
<p class="lead">Need a coder in a hurry? List your project on Instant Coder and we'll match you with a coder available for work immediately, from all around the world.</p>
<p><a class="btn btn-lg btn-primary" href="/projects/mine" role="button">Log in using GitHub &raquo;</a></p>
{% endblock %}
```

That's the first of our three states complete!

## Creating new accounts

Once a user has clicked the “/projects/mine” link, Kitura-Credentials will start the GitHub authentication process. That happens elsewhere, and it’s effectively out of our control - the user can close their browser tab and walk away if they choose. But assuming they *don’t* and instead authorize our app, they get redirected back to “/login/github/callback” where Kitura-Credentials stores their details in their session and continues their original request.

That part – the continuing their original request – is where we are now. They asked to go to “/projects/mine”, and Kitura-Credentials stepped in and did its whole authentication thing for us. That’s completed now, so it’s time for them to go move to the next step: creating an Instant Coder account. You don’t *need* this step – you can just create an account silently with default options – but I’d rather show you how it’s done so you can use it in your own projects.

First, we’re going to create a GET and POST route for “/signup”. This is where users will be sent when they have authenticated using GitHub but have yet to create an account. The Swift code for this is trivial, so add it now:

```
router.get("/signup") {
    request, response, next in
    defer { next() }

    guard let profile = request.userProfile else { return }

    var pageContext = context(for: request)
    try response.render("signup", context: pageContext)
}
```

The **guard let profile** is the part that guarantees the user has authenticated using GitHub. If that is nil it means they haven’t authenticated yet, in which case they really ought not to be on this page so exit.

We haven’t created `signup.stencil` yet, so let’s take a look at it now. It’s going to print a welcome message, drawing on `{{ username }}` so the user can see they have authenticated successfully. It’s then going to show a form asking the user to choose their primary programming language from a selection box. This language selection form control

will be used in other places, so we're going to make it a re-usable template that we can **include** as needed. The whole form will submit itself back to “/signup”, this time using POST, at which point we can complete their account creation and forward them on to the real app.

To make it look better we're going to be adding a little more HTML: each field will be wrapped in a **<div>** (page division) with the class “form-group”. We'll then add a label for each field, then the field itself, like this:

Create signup.stencil in the Views directory, and give it this content:

```
{% extends "master.stencil" %}

{% block title %}Sign up{% endblock %}

{% block body %}
<h1>Sign up</h1>
<p class="lead">Welcome to Instant Coder, {{ username }}!
Please select your primary programming language below so we can
create an account for you:</p>

<form method="post" action="/signup" style="max-width: 500px;">
    <div class="form-group">
        <label for="language">My primary programming language is...
    </label>
    {% include 'languages.stencil' %}
</div>

    <button type="submit" class="btn btn-primary">Create
Account</button>
</form>
{% endblock %}
```

Now create languages.stencil and add this code – it just loops over the **languages** array and prints each element one by one:

```

<select name="language" class="form-control">
    {%
        for language in languages %
    }
    <option>{{ language }}</option>
    {%
        endfor %
    }
</select>

```

That was the easy part. The more complex part is handling the POST back from the form because we need to do several things.

First, we need to get the easy checks out of the way: if `request.userProfile` isn't set it means they aren't authenticated with GitHub, and if `getPost()` doesn't return a value for the "language" field it means they didn't submit the form correctly. Either way we can bail out, because there's no point continuing.

Second, we need to talk to CouchDB. We're going to be using each user's ID number as their document ID, so we can just use `retrieve()` to pull out the user's document. If it returns a document it means the user has an account already, so we'll store that document in their session under "gitHubProfile" then forward them on to "/projects/mine".

If the user doesn't already have an account, we'll create one for them. However, there's a small hiccup: Kitura-Credentials returns to us only the user's ID number ("190200") and display name ("Paul Hudson"). What we *really* want is their GitHub login name, which for me would be @twostraws – that's how users are normally identified on GitHub. To get that information we need to make an API request to GitHub to fetch more user information. For example, to get my user profile, you'd make this request:

`http://api.github.com/user/190200`

We already have the user's ID, so we can use string interpolation to find their profile, like this:

`http://api.github.com/user/\\(profile.id)`

Back in project 8 I showed you how to use the `Data(contentsOf:)` initializer to download a remote image, and we can use the same technique here to fetch the GitHub profile.

It's in JSON format, so we can pass it straight into SwiftyJSON's **JSON** class to parse it.

Once it's parsed as JSON, we're going to do something a bit creative with it. The data we get back contains lots of interesting information, such as the user's location, email address, avatar URL, and more, so we're going to use that as the basis of our user document in CouchDB. However, we want the user's ID number to be our document ID, and in CouchDB that means putting a value in the “\_id” field. So, we're going to write this code:

```
gitHubJSON["_id"].stringValue = gitHubJSON["id"].stringValue  
_ = gitHubJSON.dictionaryObject?.removeValue(forKey: "id")
```

That copies GitHub's “id” value into “\_id”, then deletes “id” from the parsed JSON so it doesn't appear twice.

We're also going to add in two more values: “type” so we can write views to query our documents, and “language” to store the value they just submitted.

That completes the user's document, so we can then send it off to CouchDB. If we get a document back from **create()** it means the whole process worked, so we'll store our document JSON in the user's session under “gitHubProfile” – the same place we used for if they had an account already.

At this point, we either found an account for them and loaded it into “gitHubProfile”, or we created an account for them and put it into “gitHubProfile”. Either way, they have a profile loaded and ready to go, so the final step is to forward them on to “/projects/mine”.

That's the entire route explained in detail, so it's time for the code. First, add this CouchDB connection code directly after the **import** lines:

```
let connectionProperties = ConnectionProperties(host:  
"localhost", port: 5984, secured: false)  
let client = CouchDBClient(connectionProperties:  
connectionProperties)  
let database = client.database("instantcoder")
```

Now add this new route below the **router.get("/signup")** route:

```

router.post("/signup") {
    request, response, next in
    defer { next() }

    // bail out if we're missing GitHub authentication details
    // or a valid form submission
    guard let profile = request.userProfile else { return }
    guard let fields = getPost(for: request, fields:
        ["language"]) else { return }

    // check if the user ID already has an account
    database.retrieve(profile.id) { user, error in
        if let error = error {
            // user wasn't found!

            // fetch their full profile from GitHub
            let gitHubURL = URL(string: "http://api.github.com/
user/\(profile.id)")!
            guard var gitHubProfile = try? Data(contentsOf:
                gitHubURL) else { return }

            // adjust it to fit the format we want: "_id" rather
            // than "id", plus "type" and "language"
            var gitHubJSON = JSON(data: gitHubProfile)
            gitHubJSON["_id"].stringValue =
                gitHubJSON["id"].stringValue
            _ = gitHubJSON.dictionaryObject?.removeValue(forKey:
                "id")
            gitHubJSON["type"].stringValue = "coder"
            gitHubJSON["language"].stringValue =
                fields["language"]!

            // save the document in CouchDB
        }
    }
}

```

```

database.create(gitHubJSON) { id, rev, doc, error in
    if let doc = doc {
        // it worked! Activate their profile
        request.session?["gitHubProfile"] = gitHubJSON
    }
}
} else if let user = user {
    // user was found, so just log them in
    request.session?["gitHubProfile"] = user
}
}

// redirect them to the logged-in homepage
_ = try? response.redirect("/projects/mine").end()
}

```

We now just need to implement one more route – “/projects/mine” – to bring this site to life.

## Routing to success

The next step in this project is to implement the “/projects/mine” page, which will show a list of projects owned by the current user and give them the option to delete them. The route itself isn’t terribly complicated: it will use `queryByView()` to find projects owned by the user, then render them using a Stencil template. What *is* complicated is figuring out who they are, because the web is non-linear.

Let me explain. We’ve just created a path through our website: users come to the homepage and click the login button, they get redirected to GitHub where they approve our app, GitHub redirects them back to our app, where Kitura-Credentials continues their original request. That last part is important: they asked for “/projects/mine”, so once GitHub authentication has completed that’s where Kitura-Credentials will send them.

The problem is, we need to make sure we have an account for them. Not only is that where we store their preferred language, but we also use things like their GitHub login name. So, when “/projects/mine” is called - or indeed any protected part of our site – we need to handle various situations.

1. If `request.userProfile` is empty it means they haven’t authenticated with GitHub, so we’ll redirect the user to the homepage.
2. If “gitHubProfile” is already in their profile, it means they authenticated, have an account, and are logged in, so we’ll read from that.
3. If “gitHubProfile” isn’t set but they have an account with us, we’ll load it and set “gitHubProfile”.
4. If “gitHubProfile” isn’t set and they *don’t* have an account with us, we’ll redirect them to “/signup”.

All that needs to happen on every load of a protected page, so the best thing to do is put it into a function call that either returns their full profile or nil if there’s a problem.

Add this function just before the call to `HeliumLogger.use()`:

```
func getUserProfile(for request: RouterRequest, with response: RouterResponse) -> JSON? {
    // if they haven't authenticated using GitHub, bail out
```

```

guard let profileID = request.userProfile?.id else {
    _ = try? response.redirect("/").end()
    return nil
}

if let _ = request.session?
["gitHubProfile"].dictionaryObject {
    // they are authenticated and logged in; return their
profile
    return request.session?["gitHubProfile"]
} else {
    // they aren't logged in; see if they have an account
database.retrieve(profileID) { user, error in
    if let _ = error {
        // user wasn't found — they need to sign up
        _ = try? response.redirect("/signup").end()
    } else if let user = user {
        // user was found, so just log them in
        request.session?["gitHubProfile"] = user
    }
}

// send back their profile
return request.session?["gitHubProfile"]
}
}

```

Now, at this point you'd be forgiven for being thoroughly confused exactly what path the user takes. To be fair, it *is* complicated the first time they visit the site, because they get bounced around quite a lot:

- First they go to “/” and see the button to login and click it.
- That takes them to “/projects/mine”, which is being guarded by Kitura-Credentials so they silently get forwarded on to “/login/github”.

- “/login/github” forwards them on to GitHub, where they authorize the app and are sent back to “/login/github/callback”.
- “/login/github/callback” saves their credentials and sends them back to their original request: “/projects/mine”.
- “/projects/mine” calls our new **getUserProfile()** function and realizes they don’t have an account, so they get forwarded to “/signup”.
- Once signup is complete, they get forward back to “/projects/mine” for real this time.

The *second* time they visit the site, most of those barriers are invisible: Kitura-Credentials forwards to GitHub for authentication, but because permission has already been granted GitHub silently approves the request and forwards them back to Kitura-Credentials. That then forwards them to “/projects/mine”, where **getUserProfile()** successfully retrieves the account they created earlier, so they don’t see the signup form either.

I realize this probably all sounds chaotic right now, but in just a few minutes you’ll be able to try it for yourself and I think you’ll agree it’s pretty slick.

Now that we have the **getUserProfile()** function in place, the next step is to create a CouchDB view that “/projects/mine” can use. You already imported my views.json file in the introduction to this project, so now it’s time to take a look inside there at what we have. Find this view:

```
"projects_by_owner": {
    "map" : "function(doc) { if (doc.type === 'project')
{ emit(doc.owner, doc); } }"
},
```

That’s the one we’re going to be using here. First, it looks for any documents with the type “project” (which is none just yet), then groups them according to their owner. This means we can query this view using our GitHub ID as a key, and get back a list of all the projects the current user owns. If the query succeeds, we’ll use **arrayValue** to convert it to data Stencil can consume, then pass it off to a new template for rendering.

Add this route now:

```

router.get("/projects/mine") {
    request, response, next in
    defer { next() }

    // make sure they are fully authenticated and logged in
    guard let profile = getUserProfile(for: request, with:
response) else { return }
    guard let gitHubID = profile["login"].string else { return }

    // put together basic page context
    var pageContext = context(for: request)

    // attempt to find all our projects
    database.queryByView("projects_by_owner", ofDesign:
"instantcoder", usingParameters: [.keys([gitHubID as
Database.KeyType])]) { projects, error in
        if let error = error {
            // this ought never to happen, but just in case...
            send(error: error.localizedDescription,
code: .internalServerError, to: response)
        } else if let projects = projects {
            // store our projects in the context ready for Stencil
            pageContext["projects"] = projects["rows"].arrayObject
        }
    }

    // active the "My Projects" tab
    pageContext["page_projects_mine"] = true

    // render the content using projects_mine.stencil
    try response.render("projects_mine", context: pageContext)
}

```

We haven't made `projects_mine.stencil` yet, but that's next. It needs to loop over all the

projects it's given, and print out information about them. But because these are *our* projects, we want to let users delete them if they want.

Deleting things is always a bit risky, and that's putting it mildly. In fact, it's not uncommon to see apps never delete things, and instead just set a "deleted" flag to keep something hidden. Not only do users often feel "delete regret" where they wish they hadn't deleted something – usually mere seconds after deleting it – but it's also possible a user clicked to delete something by accident.

We're going to add deleting now, so that users can delete projects they no longer want. However, we're going to do it partly in Swift and partly in JavaScript so that users have a confirmation message: are you quite sure you want to delete this thing?

To delete a document in CouchDB, you need to call `delete()` with the document ID and revision you want to scrap. These are both long, unguessable values, so we're going to include them right in our links, like this:

```
<a href="/projects/delete/{{ project.value._id }}/{{ project.value._rev }}>Delete?</a>
```

When the user clicks that link, it will take them to "/projects/delete/projectid/projectrevision", at which point we'll perform the deletion. To make it a little harder to mis-click we're going to add two extra attributes. First, this:

```
class="text-danger"
```

That highlights the link in red – "this is dangerous". Second, this:

```
onClick="return confirm('Are you sure you want to delete  
this?');"
```

That shows a confirmation window in JavaScript, displaying the text in quote marks. If the user clicks No, the browser does *not* follow the link – it's as if it were never clicked. If they click Yes, then the link is followed as per usual, and the document is deleted.

Create a new file in the Views directory called "projects\_mine.stencil", then give it this

content:

```
{% extends "master.stencil" %}

{% block title %}Welcome{% endblock %}

{% block body %}
<h1>Your projects</h1>
<ul>
{% for project in projects %}
<li class="lead">
    <strong>{{ project.value.name }}</strong> <span
class="badge alert-info">{{ project.value.language }}</span>
<br />
    {{ project.value.description }}<br />
    <a href="/projects/delete/{{ project.value._id }}/
{{ project.value._rev }}" class="text-danger" onClick="return
confirm('Are you sure you want to delete this?');">Delete?</a>
</li>
{% empty %}
<li class="lead">You don't have any projects yet. <a href="/
projects/new">Create a project now</a>.</li>
{% endfor %}
</ul>
{% endblock %}
```

In order to finish off this route, we just need to handle the deletion clicks. This will use the route “/projects/delete/:id/:rev”, then pass the “id” and “rev” values directly on to CouchDB. When that completes, successfully or otherwise, we redirect users straight back to “/projects/mine”.

Add this route now:

```
router.get("/projects/delete/:id/:rev") {
    request, response, next in
```

```
defer { next() }

guard let profile = getUserProfile(for: request, with:
response) else { return }

guard let id = request.parameters["id"] else { return }
guard let rev = request.parameters["rev"] else { return }

database.delete(id, rev: rev) { error in
    _ = try? response.redirect("/projects/mine")
}

}
```

At long last, we've completed the full flow from new user to authenticated user, to logged-in user, to user browsing their projects. There's nothing to see yet, but that will come in due course – I strongly recommend you build and run your code to try it out so you can see exactly how the authentication flow works.

## Creating new projects

Now that Kitura-Credentials is configured and running, the rest of this project is surprisingly easy – well, *easy* because you’ve learned it all already, you clever clogs, you.

Let’s look at two more routes now. One I’ll go through with you from scratch, and one you can try yourself then read my solution.

First, “/projects/new”. This is where users can list new projects they want other people to apply for. The GET part of this is easy enough: ensure we have a valid user profile, set up the Stencil context with the correct tab highlighted, then render it.

Add this route to main.swift:

```
router.get("/projects/new") {
    request, response, next in
    defer { next() }

    guard let profile = getUserProfile(for: request, with:
response) else { return }

    var pageContext = context(for: request)
    pageContext["page_projects_new"] = true

    try response.render("projects_new", context: pageContext)
}
```

The projects\_new.stencil template is a little more complex than our previous templates because it needs to have three form fields: a project name, some description text, and a language selection box to help prospective coders find it. For the “language” field we’re just going to **include** the languages.stencil file we made earlier.

Create projects\_new.stencil in the Views directory, and give it this content:

```
{% extends "master.stencil" %}
```

```

{%- block title %}Welcome{% endblock %}

{%- block body %}
<h1>Create new project</h1>

<form method="post" style="max-width: 500px;">
  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" name="name" class="form-control" />
  </div>

  <div class="form-group">
    <label for="description">Description:</label>
    <textarea name="description" rows="5" class="form-control"></textarea>
  </div>

  <div class="form-group">
    <label for="language">Language:</label>
    {% include 'languages.stencil' %}
  </div>

  <button type="submit" class="btn btn-primary">Create</button>
</form>
{%- endblock %}

```

So, that shows the form and waits for user feedback. The flip side is handling the POST back from that form, and again it's nothing you haven't seen before:

1. Check the user has logged in with a valid account.
2. Use `getPost()` to check all three form fields are present.
3. Use their submitted fields as the basis for our CouchDB document.
4. Add “type” and “owner” fields, then convert the document to JSON

5. Send that off to CouchDB using `create()`.
6. If that fails, show an error; if it succeeds, send them back to “/projects/mine” where the new project should be listed.

We've done all that before, but I've added comments to the code matching the numbers above just as a reminder. Add this route to main.swift:

```

router.post("/projects/new") {
    request, response, next in

    // 1: check we have a fully authenticated user
    guard let profile = getUserProfile(for: request, with:
response) else { return }

    // 2: make sure all three fields are present
    guard let fields = getPost(for: request, fields: ["name",
"description", "language"]) else {
        send(error: "Missing required fields", code: .badRequest,
to: response)
        return
    }

    // 3: base our new document on their submitted fields
    var newProject = fields

    // 4: add "type" and "owner" so we can create CouchDB views,
then convert to JSON
    newProject["type"] = "project"
    newProject["owner"] = profile["login"].stringValue
    let newDocument = JSON(newProject)

    // 5: send the document to CouchDB
    database.create(newDocument) { id, revision, doc, error in
        // 6: show an error or redirect depending on the result
    }
}

```

```

        if let error = error {
            send(error: error.localizedDescription,
code: .internalServerError, to: response)
            return
        }

        _ = try? response.redirect("/projects/mine")
next()
}
}

```

That's the complete project creation route complete, so give it a try now – you should be to create new projects and see them appear in the “My Projects” tab.

The next route is “/projects/all”, but this is one I'd like you to try yourself. Don't worry – my solution is below. But if you've ever caught yourself thinking, “I wonder how much of all this I'm actually remembering?” here's your chance to find out. Seriously, trying things yourself, working hard to remember, then reading a solution is a great way to help things sink in.

This route needs to:

1. Use **getUserProfile()** to ensure the user is fully authenticated.
2. Create the basic page context using **context(for:)**
3. Use **queryByView()** to find all projects in the database.
4. If that fails for some reason, show an error. Otherwise, user **arrayValue** to convert the CouchDB documents into something Stencil can use.
5. Activate the “page\_projects\_all” tab by setting that value in the page context.
6. Render a “projects\_all” Stencil template.

I'd like you to code the route and the project\_all.stencil template file by yourself, following those instructions above. The projects\_all template is similar to projects\_mine, but we don't want to add deletion links this time.

In terms of what CouchDB view to use, take a look in views.json and see which one looks good. We don't need any filtering here – it should show *all* projects.

OK, go ahead and give it a try now. My solution is below when you're ready.

This space left intentionally blank.

This one too.

I don't want you seeing my solution by *accident*, after all.

Still here? I'm going to take you at your word that you've tried it yourself. You wouldn't lie to me, would you?

Here's my solution:

```
router.get("/projects/all") {
    request, response, next in
    defer { next() }

    guard let profile = getUserProfile(for: request, with:
response) else { return }
    var pageContext = context(for: request)

    database.queryByView("projects", ofDesign: "instantcoder",
usingParameters: []) { projects, error in
        if let error = error {
            send(error: error.localizedDescription,
code: .internalServerError, to: response)
        } else if let projects = projects {
            pageContext["projects"] = projects["rows"].arrayObject
        }
    }

    pageContext["page_projects_all"] = true
    try response.render("projects_all", context: pageContext)
}
```

And here's my projects\_all template:

```
{% extends "master.stencil" %}

{% block title %}Welcome{% endblock %}

{% block body %}
<h1>All projects</h1>
<ul>
{% for project in projects %}
<li class="lead">
    <strong>{{ project.value.name }}</strong><br />
    {{ project.value.description }}<br />
    <span class="badge alert-
info">{{ project.value.language }}</span> Contact: <a
href="https://www.github.com/
{{ project.value.owner }}">{{ project.value.owner }}</a>

</li>
{% empty %}
<li class="lead">There aren't any projects yet. <a href="/
projects/new">Create a project now</a>.</li>
{% endfor %}
</ul>

{% endblock %}
```

Done!

## Finding places to work

There's one more route we're going to look at before this project is complete: “/projects/search”. This isn't *tricky*, but it is the first time we've used two sets of CouchDB queries in one page so I'd rather walk you through it than use it as homework.

So far, our site lets people log in, create accounts, create projects, and view projects. However, it doesn't let them easily find people and projects that use a specific language – to be able to say “show me all the projects that use Swift,” or “show me all the Python coders available for work,” for example.

We're going to code this all as a single route, using a query parameter to perform the search. Inside views.json you'll find the "projects\_by\_language" and "coders\_by\_language" views. The two are almost identical:

```
"projects_by_language": {  
    "map" : "function(doc) { if (doc.type === 'project')  
    { emit(doc.language, doc); } }"  
},  
"coders_by_language": {  
    "map" : "function(doc) { if (doc.type === 'coder')  
    { emit(doc.language, doc); } }"  
}
```

Both emit **doc.language** as the key, so we can filter on a particular language. This means we can run both queries using the same data, then combine them into a single result for Stencil to render.

When the page first loads the user won't have specified a language to search for yet, so we won't show any results. The page will show them our list of languages to choose from, and when they do the page will be reloaded with the “language” query parameter set – at which point we can run our queries.

Add this final route to main.swift now:

```
router.get("/projects/search") {
```

```

request, response, next in
defer { next() }

// Check we have a fully authenticated user
guard let profile = getUserProfile(for: request, with:
response) else { return }

// set up the basic context for Stencil
var pageContext = context(for: request)

// if the user specified a search language
if let languageParameter =
request.queryParameters["language"] {
    // find all matching projects
    database.queryByView("projects_by_language", ofDesign:
"instantcoder", usingParameters: [.keys([languageParameter as
Database.KeyType])]) { projects, error in
        if let error = error {
            send(error: error.localizedDescription,
code: .internalServerError, to: response)
        } else if let projects = projects {
            // add them to our Stencil context
            pageContext["projects"] =
projects["rows"].arrayObject
        }
    }

    // find all matching coders
    database.queryByView("coders_by_language", ofDesign:
"instantcoder", usingParameters: [.keys([languageParameter as
Database.KeyType])]) { projects, error in
        if let error = error {
            send(error: error.localizedDescription,
code: .internalServerError, to: response)
    }
}
}

```

```

    } else if let projects = projects {
        // add them to the Stencil context
        pageContext["coders"] =
            projects["rows"].arrayObject
    }
}

// activate the correct tab in master.stencil
pageContext["page_projects_search"] = true

// render it all
try response.render("projects_search", context: pageContext)
}

```

The projects\_search.stencil template takes a little more code than normal, because it needs to show both sets of results as well as the language selection box. Way back in project 1 I said that Bootstrap uses a 12-column grid system for flexible layouts, and we're going to lean on that here: we're going to use the class "col-md-6" for each result of data, making them take up half the screen each. Other than that, the template is straightforward: loop over both arrays, printing out each item.

Create projects\_search.stencil and give it this content:

```

{%- extends "master.stencil" %}

{% block title %}Search Projects{% endblock %}

{% block body %}
<h1>Project search</h1>

<form class="form-inline">
<p class="lead">Search by language: {% include
'languages.stencil' %}</p>

```

```

<input type="submit" value="Search" class="btn btn-primary" />
</p>
</form>

<div class="row">
    <div class="col-md-6">
        <h1>Projects</h1>
        <ul>
            {% for project in projects %}
                <li class="lead">
                    <strong>{{ project.value.name }}</strong> <span
                        class="badge alert-info">{{ project.value.language }}</span>
                    <br />
                    {{ project.value.description }}
                </li>
            {% empty %}
                <li class="lead">No projects found.</li>
            {% endfor %}
        </ul>
    </div>

    <div class="col-md-6">
        <h1>Coders</h1>
        <ul>
            {% for coder in coders %}
                <li class="lead">
                    <a href="https://www.github.com/{{ coder.value.login }}">{{ coder.value.name }}</a>
                </li>
            {% empty %}
                <li class="lead">No coders found.</li>
            {% endfor %}
        </ul>
    </div>

```

```
{% endblock %}
```

That's it! We're done with this project. Go ahead and give it a try – you might find it useful to register a second GitHub account for testing purposes.

## Wrap up

I think this has been a project of two halves. The first half was almost certainly bewildering, at least to begin with: Kitura-Credentials requires you to take your hands off the wheel and trust it to do its job, which is tricky at first. The routing and redirecting happens invisibly, and although it's hard to understand at first it works great in practice.

The second half was much easier, perhaps even boring at this point in your Kitura career – it's just going over similar steps to what we've done before. It's still good practice, though, so I hope you took the time to follow along!

In this project we used GitHub with Kitura-Credentials, but it works just as well with Facebook and Google too. One advantage of GitHub is that it doesn't require approval to use, whereas Facebook are much more cagey about letting you create apps – they like to see what you've made before it goes out to the public.

## Homework

There are three things you could consider adding to this project.

For something easy, try removing all the `send(error:)` calls and replacing them with meaningful error messages. This is a user-facing website, so it's better to show helpful, attractive errors rather than plain text.

For something slightly harder, try updating the “/projects/search” route to use Ajax, just like we covered in project 8.

Finally, try adding another field to projects that mark them as open or closed, where “open” projects are those actively seeking help, and “closed” are not. Ideally the “/projects/all” route should show only those projects that are marked “open”, so you'll need to adjust the CouchDB view.

# **Chapter 11**

## AppleFanatic

## Setting up

This is the last major project in the book, and it's a biggie. But now that you've mastered all the Kitura fundamentals, it's time to learn something quite different: how to organize your code neatly.

So far, we've been putting all our code in main.swift, which is perfectly fine while you're learning but is a terrible way to build larger projects. That's changing here: I'm going to show you how to split your project up sensibly, using multiple HTTP servers and also multiple routers. You're also going to learn how to add Markdown support to your projects, so that users can submit formatted messages.

Of course, all this is going to be part of another real project, and this one is full of possibilities for you to extend once we're done. We're going to build a content management system, a CMS, which lets people publish posts to a website. In our case, that will be a fictional news site called "AppleFanatic" – your source for unending gushing about Apple products. We'll be using MySQL for storage, to give you a little more practice with queries, and just to make your life [fun/horrible/delete as appropriate] we'll be using normalized tables.

Let's start with the basics: create a new executable package called project 11, then give it this Package.swift:

```
import PackageDescription

let package = Package(
    name: "project11",
    dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura.git",
majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
HeliumLogger.git", majorVersion: 1),
        .Package(url: "https://github.com/IBM-Swift/
BlueCryptor.git", majorVersion: 0, minor: 8),
        .Package(url: "https://github.com/IBM-Swift/Kitura-
StencilTemplateEngine.git", majorVersion: 1),
        .Package(url: "https://github.com/crossroadlabs/
```

```

    Markdown.git", Version("1.0.0-alpha.2")!),
        .Package(url: "https://github.com/twostraws/
SwiftSlug.git", majorVersion: 0, minor: 2),
        .Package(url: "https://github.com/vapor/mysql.git",
majorVersion: 1)
]
)

```

There are two new packages in there: SwiftSlug by yours truly, and crossloadlabs/Markdown. The former is used to generate “slugs”, which are URLs based on free text strings. For example, “hello, world!” would become “hello-world”. The latter is a Markdown framework, which converts text strings into Markdown.

Markdown is a simple text formatting language that adds styling and formatting to documents in a way that makes them easy to read even when viewed as plain text. It’s commonly used online, and you’ll find it on Stack Overflow, Reddit, GitHub and others – see <https://daringfireball.net/projects/markdown/> for a full reference.

The crossroadlabs/Markdown framework uses a popular C library called Discount, which is the same library used by GitHub. If you’re using my Docker image you have Discount already installed ready to go, but if you’re using your own Linux server you should install the “libmarkdown2-dev” package. On macOS, run “brew install discount” to install it.

In the project files for this book, find project11-files and copy “public”, “Views” and “Working” into your project folder. The “public” folder contains all the usual Bootstrap code, the “Working” folder contains some SQL to set up our database, and “Views” contains a new master.stencil file that contains a few tweaks – more on that later.

Run “swift build” to fetch all the project dependencies. If you’re using macOS you’ll need to add the “-Xlinker -L/usr/local/lib” flag like before, otherwise the MySQL code won’t build properly. It’s also likely you’ll see a stream of warnings saying something like “ld: warning: object file (/usr/local/lib/libmarkdown.a(resource.o)) was built for newer OSX version (10.12) than being linked (10.10)” It’s just a warning, and it isn’t fixable unless you’re using Xcode 8.2 or later.

If you *are* using Xcode 8.2 or later, in theory(!) this command should make everything work without warnings:

```
swift build -Xlinker -L/usr/local/lib -Xswiftc -target -Xswiftc  
x86_64-apple-macosx10.12
```

Alternatively, you might find it easier to generate an Xcode project then change its “macOS Deployment Target” to 10.12. You should also select the MySQL target then go to the Build Settings tab and filter for "Other Linker Flags". Just like in project 7, you need to add "-L/usr/local/lib" to the project11 target in order to tell Xcode to look where Homebrew installed MySQL's library.

If you're using Docker or your own Linux, just running “swift build” is enough. Remember when I said right at the beginning that I recommended you use Docker? It's possible that the Swift package manager will show you messages telling you to install “libmarkdown2-dev”, but you can safely ignore them – it's a bug.

Now that everything is hopefully building cleanly, please open main.swift and give it this content:

```
import Foundation  
import Kitura  
import HeliumLogger  
import LoggerAPI  
  
HeliumLogger.use()  
let router = Router()  
  
Kitura.run()
```

You'll notice there are no calls to **Kitura.addHTTPServer()** in there this time, which is new. This is because it's time for you to learn how to organize your projects neatly, and here we have three distinct components:

- A back-end CMS API. This is responsible for fetching and creating content, and serves

up JSON.

- A front-end CMS that communicates with the API to let users create and edit content.
- A front-end website that serves up content for end users.

If you were building a truly production-ready CMS, you'd want that in three completely separate projects so they were completely independent. While learning, though, we're going to cheat a little: all three will be developed inside one Xcode project, but we're going to keep the code for the back-end and the front-end separate so you can see how it's done.

## Bootstrapping our servers

Our front end code and our back end code will be designed to run on different servers. That doesn't necessarily mean different physical computers – they could be running on the same physical server, just in two separate Docker instances. They can even run in the *same* Docker instance if you want, which is how we'll be developing them here. What matters is that the two talk to each other only across a public interface – there are no shared variables they exchange that would make it hard to pull them apart later on.

The first step in this project is get our two servers up and running. This means creating two new classes to represent each side of the system, setting them up with basic routing, and telling Kitura to start running them. Way back in the introduction to this book I mentioned that my Docker configuration opens up port 8089 as well as port 8090, and this is why – it lets us communicate directly with the back end while we're developing, to make sure our APIs work as expected.

We need to add two new files to our project. If you're not using Xcode that's as easy as creating them in the Sources directory. If you *are* using Xcode, I'd like you to find the "project11" group in the project navigator – that's the one that is directly above main.swift. Now right-click on it and choose New File, then select Swift File and name it BackEnd.swift. **Make sure you add it to the project11 target, and not to LoggerAPI or anything else.** Now do the same thing again, this time naming the file FrontEnd.swift. Again, make sure you add it to the project11 target and not anything else.

To begin with, we're going to create **BackEnd** and **FrontEnd** classes with just enough information to get them started. Both of them will have a lazy property called **router** that configures their own router and returns it, ready for main.swift to use to create a Kitura server.

First, here's BackEnd.swift:

```
import Foundation
import HeliumLogger
import Kitura
import KituraNet
import LoggerAPI
import MySQL
```

```

import SwiftyJSON

class BackEnd {
    lazy var router: Router = {
        let router = Router()

        router.post("/", middleware: BodyParser())

        return router
    }()
}

```

Yes, there are quite a few imports there – they'll be used later, but we might as well add them now.

All that **router** code does is set up a router with a **BodyParser** instance and return it.

Next, the front end. This is also going to set up a **BodyParser** to catch posts, but we're also going to configure a **StencilTemplateEngine** instance with an empty namespace. We'll be adding to that later on; this is just a stub.

```

import Foundation
import Kitura
import KituraNet
import KituraStencil
import Markdown
import Stencil
import SwiftyJSON
import SwiftSlug

class FrontEnd {
    lazy var router: Router = {
        let router = Router()
        router.setDefault(templateEngine:
            self.createTemplateEngine())
    }()
}

```

```

    router.post("/", middleware: BodyParser() )

    return router
}()

func createTemplateEngine() -> StencilTemplateEngine {
    let namespace = Namespace()
    return StencilTemplateEngine(namespace: namespace)
}
}

```

Again, lots of imports, and again we'll be needing them later.

Now that we have basic code for the front end and back end, we can start them up when our app launches. Head back to main.swift and add this code directly before the **Kitura.run()** line:

```

let backend = BackEnd()
let frontend = FrontEnd()

Kitura.addHTTPServer(onPort: 8089, with: backend.router)
Kitura.addHTTPServer(onPort: 8090, with: frontend.router)

```

As you can see, that runs the back end on port 8089, and the front end on port 8090 – the two really are completely separate.

Build and run your app now, remembering to add all those build flags if you're on macOS. Once it's running, you should be able to reach <http://localhost:8089/> and <http://localhost:8090/> in your web browser, and see the default Kitura homepage on both.

## Fetching stories from a database

The first real work we’re going to do is set up the back end to serve stories from our database, and have the front end render them. If you were working in a team, chances are you’d have one team developing the back end and another developing the front end, but we don’t have that luxury here so you’ll have to wear both hats and get used to changing them!

You should already have copied my Working directory into your project folder. If you look inside there you’ll find 0.sql, which contains two table definitions along with some test data. First, there’s the “categories” table, which contains a list of categories writers can choose from when creating stories. Second, there’s the “posts” table, which contains quite a few fields:

- “title” is the story title.
- “strap” is the strapline or subheading – a line or two of taster text that runs under the title.
- “content” is the story content as plain text.
- “category” is an integer pointing into the “categories” table.
- “slug” is the URL where the story can be read online.
- “date” is the date a given story was posted.

If the MySQL server isn’t already running, please start it now: “/etc/init.d/mysql start” on Docker and Linux, or “mysql.server start” on macOS.

Next, launch the MySQL monitor app to connect to the server on the terminal. On Docker and Linux, that’s “mysql -u root”, and on macOS that’s “/usr/local/Cellar/mysql/5.7.16/bin/mysql -u root”. Now copy and paste my entire 0.sql into the MySQL monitor app to create the tables and data.

MySQL is now configured and ready to go, so the next step is to write our first back end route: “/stories”. This will connect to MySQL, load all the available stories sorted by date, then send them back as JSON. As you saw, each post knows its category ID, which points to a row in the “categories” table. We want to send back category *names* not *numbers*, so we’re going to write an SQL query that merges the two. If you skipped over the SQL technique project, you’re probably about to regret it, because here’s the query we’ll be using:

```
SELECT p.`id`, `title`, `strap`, c.`name` AS `category`,
```

```
`slug`, `date` FROM `posts` p, `categories` c WHERE  
p.`category` = c.`id` ORDER BY `date` DESC;"
```

That loads all the story data, resolving the category ID against the category name from “categories”, all in one. If we didn’t do it this way, we’d need to load the categories separately and try to resolve them in code, which isn’t very efficient.

Just like in project 7, each of our routes will form their own connection to the database to use for the life of that request. So, we’re going to add a function to handle all that and return the database and connection. Add this method to the **BackEnd** class:

```
func connectToDatabase() throws -> (Database, Connection) {  
    let mysql = try Database(  
        host: "localhost",  
        user: "swift",  
        password: "swift",  
        database: "swift"  
    )  
  
    let connection = try mysql.makeConnection()  
    return (mysql, connection)  
}
```

Finally, we can write our "/stories" route. Because our code is all neatly organized now, each of our routes will be implemented as a method on their parent class, rather than free-floating functions in main.swift. This means they accept their parameters in the old-fashioned way, rather than like a closure:

```
func getAllStories(request: RouterRequest, response:  
RouterResponse, next: () -> Void) throws {
```

That doesn’t specify a route, and it doesn’t need to – that’s done separately.

This method will connect to MySQL, run our posts + categories combining query, then convert the result into dictionaries that can be sent out using JSON. Add this method to the **BackEnd**

class now:

```
func getAllStories(request: RouterRequest, response: RouterResponse, next: () -> Void) throws {
    defer { next() }

    // connect to MySQL
    let (db, connection) = try connectToDatabase()

    // pull out all stories, newest first
    let query = "SELECT p.`id`, `title`, `strap`, c.`name` AS
`category`, `slug`, `date` FROM `posts` p, `categories` c WHERE
p.`category` = c.`id` ORDER BY `date` DESC;"
    let posts = try db.execute(query, [], connection)

    // convert the MySQL result to native Swift types
    var parsedPosts = [[String: Any]]()

    for post in posts {
        var postDictionary = [String: Any]()
        postDictionary["id"] = post["id"]?.int
        postDictionary["title"] = post["title"]?.string
        postDictionary["strap"] = post["strap"]?.string
        postDictionary["category"] = post["category"]?.string
        postDictionary["slug"] = post["slug"]?.string
        postDictionary["date"] = post["date"]?.string

        parsedPosts.append(postDictionary)
    }

    // convert the result to JSON and send it out
    response.status(.OK).send(json: JSON(parsedPosts))
}
```

Before you try and run that, we need to do one more thing: we need to hook that method up to a route. Head up to the **router** lazy property and add this code directly before the **return router** line:

```
router.get("/stories", handler: self.getAllStories)
```

**Note:** You need to use **self** there because we're inside an automatically executed closure.

We're going to add some front-end code in just a moment, but it's a smart idea to pause here, run your existing code, and try it out in your browser. In this case, you should be able to get to <http://localhost:8089/stories> in your web browser and see two example stories returned.

Our front-end class is going to be making a lot of requests to the back end, and for that we'll be using KituraNet's **HTTP** class. This is very powerful, and very configurable, but suffers from the same irritation as Kitura-CouchDB: even though it operates synchronously, it *looks* like it's asynchronous, because it uses a completion handler closure. I said previously that I'm going to show you how clean up this synchronous/asynchronous mess, and now is the time: we're going to create a method that wraps **HTTP** beautifully, and in doing so make the rest of our front-end routes almost comically simple.

There are three steps here. First, we're going to add three properties to our **FrontEnd** class so that we can configure where our back-end server is. Add these now:

```
var backEndSchema = "http"  
var backEndHost = "localhost"  
var backEndPort: Int16 = 8089
```

The port number needs to be **Int16** because that's what KituraNet requires. I've set them to default values that work for this project, but if you create code to make the front end and back end work on different servers you would need to change these.

The second step is to create a **fetch()** method. This will accept three parameters: the path to fetch, the HTTP method to use, and any body text to include. The **body** parameter is how we'll send data across to the back end, such as when we create a story much later on.

KituraNet's **HTTP** class is designed to connect to a server then send and receive data. You

create a connection using the `request()` method, passing it an array of client request options – the host name you want to connect to, the HTTP method to use, and so on. You can also pass a dictionary of custom HTTP headers, and we’re going to use that to notify the back end that any data we send will be JSON. That gets picked up by the `BodyParser` on the back end, so we can read the submitted data.

Once you’ve configured what the request should do, it’s time to step into the not-really-asynchronous mess. The `request()` method accepts your client request options as its first parameter, plus a trailing closure that should be executed when the request completes. What we’re going to do is create a `Data` instance beforehand, and use that inside the request closure, like this:

```
var responseBody = Data()

let req = HTTP.request(requestOptions) { response in
    if let response = response {
        guard response.statusCode == .OK else { return }
        _ = try? response.readAllData(into: &responseBody)
    }
}
```

Inside the closure, we check for the HTTP response code `.OK`, and if we got that then we’ll read the response into the `responseBody` object we created outside the closure.

Now, remember: that code looks like it’s asynchronous, but isn’t really. By declaring `responseBody` outside the closure, we’re able to use it once the request has completed. Swift recognizes we’re using it inside the closure and captures it correctly, ensuring that `requestBody` inside and outside the closure both point to the same thing.

Cunningly, the `request()` method doesn’t actually request anything, it just creates a request ready to go and attaches our completion closure for later. To actually send the request and get things moving, we need to call the request’s `end()` method. We’ll use this with one parameter, which is the body text to send to the server. This will often be empty, but it will contain JSON when we’re sending data to the back end.

When you call `end()`, the request is made for real, the response is received, our completion closure is run, and all being well `responseBody` is filled with data. To end the method, we'll check the size of `responseBody`: if it's greater than 0 it means it was successfully filled with data, so we'll convert it to a `JSON` object and return it; otherwise we'll return nil because something failed along the way.

To recap:

1. Configure client request options specifying the host name, path, and so on.
2. Attach headers telling the back end that we're sending JSON.
3. Create an empty `Data` object called `responseBody` to hold the back end's response.
4. Create a HTTP request with a completion closure that fills `responseBody` with whatever the back end sent back.
5. Call `end()` on the request, sending in our body text, to make the request actually happen.
6. If `responseBody` was filled with data, convert it to JSON and return it, otherwise return nil.

Open `FrontEnd.swift` and add this method to the `FrontEnd` class now:

```
func fetch(_ path: String, method: String, body: String) ->
JSON? {
    // 1: Configure our client request options
    var requestOptions: [ClientRequest.Options] = []

    requestOptions.append(.schema("\(backEndSchema)://"))
    requestOptions.append(.hostname(backEndHost))
    requestOptions.append(.port(backEndPort))
    requestOptions.append(.method(method))
    requestOptions.append(.path(path))

    // 2: Tell the back end we're sending JSON
    let headers = ["Content-Type": "application/json"]
    requestOptions.append(.headers(headers))
```

```

// 3: Create an empty Data object to hold the response
var responseBody = Data()

// 4: Create a request that will fill responseBody if it succeeds
let req = HTTP.request(requestOptions) { response in
    if let response = response {
        guard response.statusCode == .OK else { return }
        _ = try? response.readAllData(into: &responseBody)
    }
}

// 5: Send the request, passing in any body text
req.end(body)

// 6: If it all worked, convert the response to JSON and send it back, otherwise return nil
if responseBody.count > 0 {
    return JSON(data: responseBody)
} else {
    return nil
}
}

```

So, step one was creating properties to point to our back end, and step two was writing a **fetch()** method to wrap up KituraNet’s not-quite-asynchronous **HTTP** class in a friendly way. Step three is nice and easy: we’re going to write two more small methods called **get()** and **post()** that wrap **fetch()**. You see, even though **fetch()** is a huge improvement over the raw KituraNet code, it’s easy to make mistakes by writing “GET” as “get” (yes, they are different – thanks HTTP spec writers!), and we can also make it easier to send dictionaries of data rather than converting JSON by hand.

First, the **get()** method – add this to main.swift now:

```
func get(_ path: String) -> JSON? {
    return fetch(path, method: "GET", body: "")
}
```

As you can see, that just passes the call onto **fetch()**, filling in the method and body with default values.

Second, the **post()** method. This does a little more work because it's going to accept a field list of the type **[String: Any]**, and convert that to JSON like this:

```
let string = JSON(fields).rawString() ?? ""
```

The **rawString()** method returns the **JSON** object as raw JSON text, rather than Swift objects. It might return nil, so I've added the nil coalescing operator to provide a default string in case that happens.

Add this **post()** method below **get()**:

```
func post(_ path: String, fields: [String: Any]) -> JSON? {
    let string = JSON(fields).rawString() ?? ""
    return fetch(path, method: "POST", body: string)
}
```

All that code might seem pointless, but trust me: it makes our front-end routes significantly easier, as you'll see in just a moment.

We already added “/stories” on the back end, and on the front end I want to map that to “/” so the homepage is the list of all the stories you can read. Just like with the back end, each front-end route is a separate method by itself, with the route connection happening in our **router** lazy variable.

First, add this method to build a default Stencil context for all our pages:

```
func context(for request: RouterRequest) -> [String: Any] {
    var result = [String: Any]()
    return result
}
```

```
}
```

Yes, it's empty right now, and yes, Xcode will warn you that **result** isn't being changed so it should be **let** rather than **var**. Don't change it, though – we'll be adding to this later.

Now it's time to add the route for “/”. It is, quite remarkably, just five lines of code. Add this method to the **FrontEnd** class now:

```
func getHomePage(request: RouterRequest, response:  
RouterResponse, next: () -> Void) throws {  
    defer { next() }  
    var pageContext = context(for: request)  
    pageContext["title"] = "Top Stories"  
    pageContext["stories"] = get("/stories")?.arrayObject  
    try response.render("home", context: pageContext)  
}
```

Let me break that down:

1. We add a **defer** for **next()** so that Kitura carries on its routing pipeline.
2. We call **context(for:)** to get our (empty) default Stencil context.
3. We add a “title” value to the context that says “Top Stories”
4. We set the context’s “stories” parameter to be the result of calling our **get()** method on “/stories”, using **arrayObject** to convert it to Stencil-ready values.
5. We render `home.stencil` with the context we created. We haven’t written this yet, but will get to it soon.

The “title” context value is the result of a small change in `master.stencil`: I took out the **title** block and instead made it share a single variable with the title and **<h1>** tag.

The real work in that route happens in a single line:

```
pageContext["stories"] = get("/stories")?.arrayObject
```

That runs our **get()** method, which in turn runs **fetch()**, which in turn de-asynchronizes

KituraNet's **HTTP** class. It means we can prepare a request, execute it, and parse the response all in just one line of code. Even better, that line of code uses optional chaining so that **arrayObject** will only be used if a non-nil value is returned. *Even* even better, if a nil value is returned then setting **pageContext["stories"]** to nil does nothing – it just clears the value in the dictionary.

So, I realize that writing **fetch()** might have seemed like an epic detour, but I hope you can see the huge benefit it gives us – the **getHomePage()** method couldn't be much simpler.

As with the back end, just writing **getHomePage()** isn't enough to make it work; we still need to connect it to a route. Our template will use Bootstrap's CSS and JavaScript, so we're going to fire up **StaticFileServer** at the same time. Add these two lines directly before the **return router** in the lazy **router** property:

```
router.all("/static", middleware: StaticFileServer())  
  
router.get("/", handler: self.getHomePage)
```

All we need to do now is write `home.stencil`. Initially this is nice and easy, because it's just going to loop over the **stories** array and print it out:

```
{% extends "master.stencil" %}  
  
{% block body %}  
{% for story in stories %}  
<h2><a href="{{ story.id }}>{{ story.title }}</a></h2>  
<p>{{ story.strap }}</p>  
{% endfor %}  
  
{% endblock %}
```

Go ahead and run the project now – you should be able to load <http://localhost:8090/> in your web browser and see our two example stories. Their links don't work yet, but we'll tackle that next...

## IDs, slugs, and Markdown

The next step in this project is to make our homepage links work, so that readers can click a story and start reading. This requires two more routes – one for the back end to fetch the story, and one for the front end to render it – but it also requires some extra template work.

The template work takes two forms. First, here's how `home.stencil` writes its links:

```
<h2><a href="{{ story.id }}">{{ story.title }}</a></h2>
```

That will create story links like <http://localhost:8090/1>, which is functional but not pretty or in any way optimized for search engines. What we *want* is to show have links like this: <http://localhost:8090/reviews/1/iphone-7-review> – something that includes the category and article name right in the URL. This helps rank our pages higher on Google, and makes them friendlier for users to read and share, so these clean URLs are always a good idea.

To implement this, we're going to add a new filter to our Stencil engine that turns a story object (ID, category, title) into a fully fledged link. It's *possible* to do all that inside Stencil, like this:

```
<h2><a href="/{{ story.category }}/{{ story.id }}/{{ story.slug }}">{{ story.title }}</a></h2>
```

However, these story links could appear anywhere, and it's not pleasant having to duplicate work. What we *want* is something like this:

```
<h2><a href="{{ story|link }}>{{ story.title }}</a></h2>
```

That means adding a “link” filter that converts a story into a URL. The second thing we’re going to do is add a “markdown” filter, which accepts Markdown-formatted text, converts it to HTML, and returns it. We already added the Markdown framework from Crossover Labs, so we’ll get that to do most of the work.

Let's dive in with some code, starting with the “link” filter. We used `registerFilter()` in projects 4 and 6, but just to recap: it sends us a value of type `Any?`, so it's we need to check and unwrap that before we can use it. In our case, we expect “link” to be called with a story

object, which is `[String: Any]`, so we'll check for that. We're also going to check that the dictionary contains values for "id", "slug", and "category", just in case data is missing for some reason. In theory it shouldn't happen, but in practice theory is usually wrong.

If the filter is given a dictionary, and all three values can be unpacked, then we'll return a URL like this:

```
return "/\\(category.lowercased())/\\(id)/\\(slug)"
```

I've used `lowercased()` there because our categories are things like "News" and "Reviews", and it's generally a good idea to keep your URLs lowercased.

In the `FrontEnd` class, find the `createTemplateEngine()` method and add this code directly after the `let namespace = Namespace()` line:

```
namespace.registerFilter("link") { (value: Any?) in
    guard let unwrapped = value as? [String: Any] else { return value }

    // ensure all three fields are present
    guard let category = unwrapped["category"] as? String else
    { return value }
    guard let id = unwrapped["id"] else { return value }
    guard let slug = unwrapped["slug"] else { return value }

    return "/\\(category.lowercased())/\\(id)/\\(slug)"
}
```

That's the first filter done. The second one is to convert Markdown-formatted text to HTML. Markdown looks like this:

```
This is in *italics*.
This is a [link](https://www.hackingwithswift.com)
* This is a bullet point
```

It's human readable, but also easily converted to HTML, which is where the Markdown framework comes in. As with the "link" filter, this needs to start by converted its **Any?** value into a string, so we know we have something safe to work with.

Once have a text string, we can convert it to HTML by passing it to the **Markdown** class and reading its **document()** property. Creating a **Markdown** object and reading **document()** can both throw errors, so we're going to wrap them in **try?** calls so we can check and unwrap them safely rather than trying to handle errors.

If the Markdown parsing succeeded, and a document can be read, we'll return it – that's our converted HTML. Otherwise, we'll just return what input string we were given, because there's clearly something wrong with it.

One more thing: the Markdown framework gets a little tetchy if you use the wrong kind of line endings. A strange peculiarity of computers is that no one can agree on the best way to store line breaks in text. On Windows and the web, you get "\r\n", which means "move to the start of the line, then go to the next line". On Unix (Linux and macOS), you get "\n", which is considered to mean the same thing. Markdown likes to receive its line breaks in Unix style, which means we're going to do a quite string substitution and change any instances of "\r\n" to just "\n".

That's it – add this filter below the "link" filter:

```
namespace.registerFilter("markdown") { (value: Any?) in
    guard let unwrapped = value as? String else { return value }
    let trimmed = unwrapped.replacingOccurrences(of: "\r\n",
with: "\n")

    if let md = try? Markdown(string: trimmed) {
        if let doc = try? md.document() {
            return doc
        }
    }

    return unwrapped
}
```

```
}
```

With those two Stencil filters in place, we can use them in `home.stencil`: story links should use the `link` filter, and we can pass `story.strap` to the `markdown` filter so that writers can add basic formatting to their strap lines.

Find these two lines in `home.stencil`:

```
<h2><a href="{{ story }}>{{ story.title }}</a></h2>
<p>{{ story.strap }}</p>
```

Now change them to this:

```
<h2><a href="{{ story|link }}>{{ story.title }}</a></h2>
{{ story.strap|markdown }}
```

You'll notice I stripped out the `<p>` tag for the strap – that's because the converted text will automatically have its own `<p>` tag so we don't need to add it ourselves.

Build and run your project, then try loading the homepage again. Try clicking on one of the story titles – it won't work yet, but you should at least be able to see the URL looking good.

## Stories and errors

When you click a story link on the homepage, you see an error because we haven't told Kitura how to route those URLs. To make this route work, we need to write some back-end code to fetch the story data (including its content), plus some front-end code to handle rendering. However, we also need to write some code to handle errors gracefully: if the user requests a page that doesn't exist we need to handle it gracefully.

Let's start with the back-end code, so we can see the correct data is being sent. We're going to create a “/story/:id” route that loads one story from the database and returns it as JSON. The SQL query for that is almost identical to the one from “/stories”, although this time we're requesting one specific story ID:

```
SELECT p.`id`, `title`, `strap`, `content`, c.`name` AS
`category`, `slug`, `date` FROM `posts` p, `categories` c WHERE
p.`category` = c.`id` AND p.`id` = ?;
```

When we run that query we'll get an array back, containing all the matching documents. In this case, there should always be precisely 1 result, which means we write code like this:

```
guard let post = posts.first else {
    send(error: "Unknown story ID.", code: .notFound, to:
response)
    return
}
```

That is, “if the **posts** array is empty, return a not found error and bail out.”

If there *was* anything in the MySQL results array, the first (and only) row will be placed into **post** and we can carry on. Just like with “/stories”, we'll convert the result into a dictionary object, then convert *that* to JSON and send it back.

Add this method to the **BackEnd** class now:

```
func getStory(request: RouterRequest, response: RouterResponse,
next: () -> Void) throws {
```

```

defer { next() }

// ensure a story was requested
guard let storyID = request.parameters["id"] else {
    response.status(.badRequest).send("Missing story ID.")
    return
}

// connect to MySQL
let (db, connection) = try connectToDatabase()

// find the story they requested
let query = "SELECT p.`id`, `title`, `strap`, `content`,
c.`name` AS `category`, `slug`, `date` FROM `posts` p,
`categories` c WHERE p.`category` = c.`id` AND p.`id` = ?;"
let posts = try db.execute(query, [storyID], connection)

// if we got nothing back, return a 404
guard let post = posts.first else {
    response.status(.notFound).send("Unknown story ID.")
    return
}

// convert the MySQL result to native Swift types
var postDictionary = [String: Any]()
postDictionary["id"] = post["id"]?.int
postDictionary["title"] = post["title"]?.string
postDictionary["strap"] = post["strap"]?.string
postDictionary["content"] = post["content"]?.string
postDictionary["category"] = post["category"]?.string
postDictionary["slug"] = post["slug"]?.string
postDictionary["date"] = post["date"]?.string

// convert that result dictionary to JSON and send it back

```

```
        response.status(.OK).send(json: JSON(postDictionary))
    }
```

We still need to connect that to the “/stories/:id” route, so add this to the list of back end routes:

```
router.get("/story/:id", handler: self.getStory)
```

That’s the back end finished for this route, so before we continue try opening <http://localhost:8089/story/1> in your web browser to make sure it loads correctly. You should see the complete story returned, including its “content” field.

Next, open FrontEnd.swift again. Here we have three tasks in total, starting with handling errors.

In most of the projects so far we’ve just printed an error message and moved on, but it’s now time for you to do better than that. Add this method to the **FrontEnd** class:

```
func renderError(_ message: String, _ response: RouterResponse,
_ next: () -> Void) {
    _ = try? response.send("Error: \(message)").end()
}
```

We’re going to us that to render any errors that might crop up, for example “page not found”. I’ll be adding code throughout the project to call **renderError()** at the right times, but it’s down to *you* to fill that in with something attractive. You might, for example, want to render a template that displays the error text somehow – it’s down to you. Consider this homework in advance!

Now that we have a method in place to show errors when they happen – and it looks great, I’m sure! – we can write a route to load stories from the back end. Our URLs look like this:

<http://localhost:8090/tutorials/2/how-to-write-macos-apps>

Now, most of that is there to look good for users and to help search engines index our pages

nicely. What we care about - in fact the only thing we care about – is the ID number, which is in there nestled between the category and the slug. So, we’re going to write a route to match “/:category/:id/:slug”, but we really only care that the “id” parameter exists. If it doesn’t, we’ll render an error.

Once we have a valid ID, we’ll pass it to the “/story/:id” route on the back end, using our **get()** method. Remember, that could return an error if the story doesn’t exist, so we’re going to check and unwrap the result, calling the **renderError()** method if something went wrong.

If the ID was set and if the back end responded with a valid story, we’re going to create the page context, assign our story to it, and hand it off to Stencil for rendering.

Add this **getStory()** method now:

```
func getStory(request: RouterRequest, response: RouterResponse,  
next: () -> Void) throws {  
    guard let id = request.parameters["id"] else {  
        renderError("Missing ID", response, next)  
        return  
    }  
  
    guard let story = get("/story/\\"(id)")?.dictionaryObject else {  
        renderError("Page not found", response, next)  
        return  
    }  
  
    var pageContext = context(for: request)  
    pageContext["title"] = story["title"] ?? ""  
    pageContext["story"] = story  
    try response.render("read", context: pageContext).end()  
}
```

As per usual, we need to connect that to a route separately, so add this line under the

`self.getHomePage` route:

```
router.get("/:category/:id/:slug", handler: self.getStory)
```

We haven't written the `read.stencil` template yet, but it's nice and easy. Create it now, and give it this content:

```
{% extends "master.stencil" %}

{% block body %}
<p class="lead">{{ story.strap|markdown }}</p>
<p>{{ story.content|markdown }}</p>
{% endblock %}
```

Build and run your code now, and try clicking one of the stories. All being well you should be taken to the `read` page, and see the Markdown content being rendered as HTML. For example, the “Hacking with macOS” link should now be live, and the iPhone 7 review should contain some bold text.

Now, you might look at think it works great, but there's a rather devilish little problem hiding in there. To demonstrate it, I'd like you to load this URL in your web browser: <http://localhost:8090/static/css/bootstrap.min.css> – that's the path to the Bootstrap CSS.

You should see a massive wall of CSS. This is “minified” CSS – compressed to take up as little space as possible for bandwidth reasons. Scroll to the very end of that text wall, and have a look at the very end. Here's what I see, right at the end: `/*# sourceMappingURL=bootstrap.min.css.map */`

`sourceMappingURL=bootstrap.min.css.map */`**Error: Page not found.**

Notice that “Error: Page not found” text? That's the error text I placed inside the `renderError()` method. If you decided to render a template, you'll see far more text in there. Even though we requested “/static/css/bootstrap.min.css”, we're somehow seeing that along side the output from `renderError()`. What gives?

As you know, Kitura routes are actually a pipeline. This means we can register `BodyParser()` as middleware, and have it work alongside other routes. That's why we call

**next()** – it means “continue onto the next part of our pipeline.”

What’s happening here is that we’re requesting the route “/static/css/bootstrap.min.css”, which exactly matches our route “/:category/:id/:slug”. We placed **StaticFileServer** first in the route, which means it will find and load bootstrap.min.css successfully, but then it will pass control on to the next route that matches, which is our page loading route. It looks at “/static/css/bootstrap.min.css” and thinks “css” is the story ID to load, so it tries and fails to fetch it from the back, then renders our error page.

The end result is that we see the content from bootstrap.min.css directly followed by the content of our error page, which is clearly not what we want. To fix this, we’re going to add another route immediately after the **StaticFileServer** that does nothing other than end the request without calling **next()**. This effectively terminates the pipeline.

**Warning:** It’s important you place this directly after the **StaticFileServer** route on the front end, otherwise it might not work as expected!

Add this route now:

```
router.all("/static/*") { req, res, next in try res.end() }
```

That means “if the route is /static followed by anything at all, end the request.” This comes *after* the **StaticFileServer**, which means the file will be loaded and served fully first. Run that code then try reloading <http://localhost:8090/static/css/bootstrap.min.css> – all being well you should no longer see any error text at the end.

## Browsing by category

Load <http://localhost:8090/> and take a look in the top-right corner of the navigation bar. Do you see that “Categories” button? It’s a drop-down menu, but right now clicking it shows nothing at all. The next stage in this project is to make that work, but it’s not quite as straightforward as you might think.

If you look back at the 0.sql file in your Working directory, you’ll see there’s a “categories” table containing things like “News”, “Reviews”, and “Tutorials”. We want those to appear in that popup button so that users can go to a page that shows them all the reviews, or all the tutorials. The problem is, how can the front-end know what categories are available?

There are three options available to us:

1. Automatically send all categories alongside any other response.
2. Create a “/categories” route on the back end, and request it each time a page is loaded.
3. Create a “/categories” route on the back end, and cache its results in the front-end.

The first option means running an extra SQL query for every request that’s made, and also transferring all the categories with every page. We only have a handful of categories right now, but if there were hundreds this would be quite wasteful.

The second option is worse: it means running the extra SQL query for every request, and also handling another network request.

The third option is both simplest and most efficient: we load the categories up front once, and cache them in the front end. If we ever find we need to reload the categories, we can make that happen – although if you think about it, categories are read countless times more often than they are written.

The back end code for this isn’t too hard: **SELECT** all the categories from the database, convert them to Swift native types, convert *that* to JSON, and send it back. However, I want to use this chance to demonstrate one of my favorite Swift methods, called **flatMap()**.

You’ve already met **map()** in project 5, when converted an array of URLs to an array of strings. The **flatMap()** method works similarly, except it automatically removes any nil

values once the mapping has taken place. Our SQL query will look like this:

```
SELECT `name` FROM `categories` ORDER BY `name`;
```

Normally we'd convert that to a Swift dictionary something like this:

```
var categoryDictionaries = [[String: Any]]()

for category in categories {
    var categoryDictionary = [String: Any]()
    categoryDictionary["name"] = category["name"]?.string
    parsedCategories.append(postDictionary)
}
```

That makes sense when you have lots of data to dig through, but all we have is “name” so can we perform the entire transformation in one line of code:

```
let categoryNames = categories.flatMap { $0["name"]?.string }
```

That's closure shorthand syntax again, where **\$0** is shorthand for “the current element in the array.” That will go over every item in the **categories** array, attempt to read its “name” value, then attempt to return its “string” property. If “name” doesn't exist, or can't be converted to a string, that will return nil, but **flatMap()** automatically removes any nil values it finds, so the end result is that **categoryNames** will be a string array containing the names of our categories.

Add this method to the **BackEnd** class now:

```
func getAllCategories(request: RouterRequest, response: RouterResponse, next: () -> Void) throws {
    defer { next() }

    let (db, connection) = try connectToDatabase()

    let query = "SELECT `name` FROM `categories` ORDER BY
```

```

`name`;
    let categories = try db.execute(query, [], connection)
    let categoryNames = categories.flatMap
    { $0["name"]?.string }

    response.status(.OK).send(json: JSON(categoryNames))
}

```

We need to connect that to a route, so add this underneath the other back end routes:

```
router.get("/categories", handler: self.getAllCategories)
```

Now for the front end: we want to load the categories only once, and keep them cached so we don't keep hitting the back end needlessly. To make that happen requires four small changes, starting with a property in the **FrontEnd** class to store the category names. Add this now, before the **lazy var router** line:

```
var categories = [String]()
```

Second, we need to deliver that array to our Stencil templates, so the categories appear in the drop-down menu correctly. We already have a **context(for:)** method in place, and Xcode has been warning us that **result** isn't being changed – it's time to silence that warning.

Change the method to this:

```
func context(for request: RouterRequest) -> [String: Any] {
    var result = [String: Any]()
    result["categories"] = categories
    return result
}
```

Third, we need to write a front-end method to fetch the category list from the back end. Thanks to the magic of our **get()** method this isn't hard at all: we'll attempt to fetch “/categories” from the back end, and if we get back a string error then we'll assign it to the **categories** property. Add this method:

```

func loadCategories() {
    if let remoteCategories = get("/categories")?.arrayObject
as? [String] {
        categories = remoteCategories
    }
}

```

Now for step four: making the front-end call **loadCategories()** only once, so that the categories are loaded as soon as it starts. Here's where my ulterior motive becomes clear: I want to teach you something new.

We've been using the **Kitura.addHTTPServer()** method in main.swift ever since project 1, but we've bothered using its return value – in fact, you might not even have noticed it *has* a return value. Well, it *does*: it returns a **HTTPServer** object, and we can use that to attach arbitrary code for when the server starts. In our case, we can tell Kitura to run “loadCategories” as soon as the front end is up and running.

Your current code should look like this:

```

let backend = BackEnd()
let frontend = FrontEnd()

Kitura.addHTTPServer(onPort: 8089, with: backend.router)
Kitura.addHTTPServer(onPort: 8090, with: frontend.router)

Kitura.run()

```

What we're going to do is catch the return value from the second **addHTTPServer()** call, then use it to attach a closure so that **loadCategories()** is called. Change the above code to this:

```

let backend = BackEnd()
let frontend = FrontEnd()

Kitura.addHTTPServer(onPort: 8089, with: backend.router)

let httpServer = Kitura.addHTTPServer(onPort: 8090, with: frontend.router)
httpServer.onReady { server in
    server.closures.append(backend.loadCategories)
}

```

```
let frontEndServer = Kitura.addHTTPServer(onPort: 8090, with:  
    frontend.router)  
  
frontEndServer.started { [unowned frontend] in  
    frontend.loadCategories()  
}  
  
Kitura.run()
```

That wasn't so hard, was it? Because **FrontEnd** is a class rather than a struct, we can be sure that **loadCategories** is being called on the same object reference that is serving routes in Kitura.

Build and run your code now, then try clicking the Categories button - all being well it should work. Better yet, it works by making only one request to the back end, so it's working *and* efficient. Of course, the links don't actually *work* yet, but I have to leave *something* for homework, right?

## Creating an admin section

The last part of this project is to create an admin section, so people can log in to create and edit stories. We're not going to add any authentication here – we already covered token-based authentication in project 7, and it would be identical here.

Instead, we're going to focus on what's new, because there's more than enough right there: we're going to submit JSON to the back end when articles are saved, and also code a page that lets us create new articles or edit existing ones. I realize that doesn't sound like hard work, but as you'll see there's a lot to it.

First, the easy stuff, mostly because it's always nice to get some easy wins: an “/admin” homepage. This is almost identical to the “/” on the front end: it will fetch the list of available stories from the back end then print them out. Add this method to the **FrontEnd** class:

```
func getAdminHome(request: RouterRequest, response:  
RouterResponse, next: () -> Void) throws {  
    defer { next() }  
    var pageContext = context(for: request)  
    pageContext["title"] = "Admin"  
    pageContext["stories"] = get("/stories")?.arrayObject  
    try response.render("admin_home", context: pageContext)  
}
```

The `admin_home.stencil` template needs to offer links to edit each article, or to create a new one. We're going to combine creation and editing into a single “/admin/edit” route, so we can point them at the same place. Add this template to your Views directory now:

```
{% extends "master.stencil" %}  
  
{% block body %}  
  
<p>Select a story to edit, or <a href="/admin/edit">click here  
to create a new one</a>.</p>  
  
<ul>
```

```

{%- for story in stories %}
<li><a href="/admin/edit/{{ story.id }}">{{ story.title }}</a></li>
{%- empty %}
<li><a href="/admin/edit">Create your first story</a></li>
{%- endfor %}
</ul>

{%- endblock %}

```

When it comes to adding a route to that page, I want to show you something new. So far we've been writing routes like this:

```
router.get("/:category/:id/:slug", handler: self.getStory)
```

That works fine when your routes are varied, but often you'll have groups of routes that share a base component. For example, in project 10 we had routes like “/projects/mine”, “/projects/all”, and “/projects/search” – they all have the same “projects” component at their core.

Kitura allows us to create *subrouters*, which are routers responsible for handling only part of a route hierarchy. We're going to use one here, so that all “/admin” routing is done in one place. Add this code just before the **return router** line in the front-end routes:

```

let adminRouter = Router()
adminRouter.get("/", handler: self.getAdminHome)
router.all("/admin", middleware: adminRouter)

```

Take a close look at that, because it does two interesting things. First, we're saying that the **self.getAdminHome** route is attached to “/”, but that our subrouter is attached to “/admin”. What this means is that the **self.getAdminHome** route is actually attached to “/admin”, because it's relative to the root of its parent. So, if you decide later that having “/admin” is too easy for script kiddies to guess, you could change it to something else and have all its child URLs update immediately.

Second, notice that the **adminRouter** is being attached to the main router as middleware

using the `all()` method. This means Kitura will automatically forward all requests there – the subrouting system is effectively invisible. This means you can organize your code into sensible components without users having to see anything different.

Now that we have an admin homepage, the next step is to add a page to let writers create or edit stories. Because these two tasks are effectively identical in terms of their workflow, we're going to handle them with a single route: “/edit/:id?”. Remember, that will be mounted on our admin subrouter, so it will be accessed using “/admin/edit/2” for example.

This is a route of two parts in the front end: we need a GET route to render the editing form, and a POST route to submit changes back to the user. The GET route is predictably easy: load the page context, add a title, and send it to Stencil to render.

One slight twist is that this route is actually “/edit/:id?” – it has an optional ID route parameter. So, this code needs to check whether that's set and, if it is, fill it story data from the back end using a `get()` call.

Add this method to **FrontEnd** now:

```
func getAdminEdit(request: RouterRequest, response: RouterResponse, next: () -> Void) throws {
    defer { next() }

    var pageContext = context(for: request)
    pageContext["title"] = "Edit"

    if let storyID = request.parameters["id"] {
        pageContext["story"] = get("/story/>\n(storyID)?.dictionaryObject"
    }

    try response.render("admin_edit", context: pageContext)
}
```

The `admin_edit.stencil` template is surprisingly long, because we have quite a few fields to

submit. We also need to cater for the fact that this form might be loading an existing story, so we're going to inject values from the story into the appropriate fields. Most of the time that's easy enough to do, like this:

```
<input type="text" name="title" class="form-control"  
value="{{ story.title }}" />
```

If **story** is set then its title will be placed in there; if it *isn't* set then it will be silently ignored.

The one place this is tricky is when handling categories. Our standard page context includes all categories, but if we're in editing mode then the story will also have its own category set, so we need to make sure it starts as the selected value in the **<select>** box. Fortunately, Stencil lets us do variable checks using an **{% if %}** tag, so we'll just add the **selected** attribute to whichever option should be selected, like this:

```
<select name="category" class="form-control">  
  {% for category in categories %}  
    {% if story.category == category %}  
      <option value="{{ category|lowercase }}"  
selected>{{ category }}</option>  
    {% else %}  
      <option value="{{ category|lowercase }}">{{ category }}</  
option>  
    {% endif %}  
  {% endfor %}  
</select>
```

Notice again that we're lowercasing categories when used as data, and keeping them with capital letters when being shown to the user.

Here's the full content for `admin_edit.stencil` – please add this to your Views directory now:

```
{% extends "master.stencil" %}  
  
{% block body %}
```

```

<form method="post">
    <div class="form-group">
        <label for="name">Title:</label>
        <input type="text" name="title" class="form-control"
value="{{ story.title }}" />
    </div>

    <div class="form-group">
        <label for="name">Strap:</label>
        <input type="text" name="strap" class="form-control"
value="{{ story.strap }}" />
    </div>

    <div class="form-group">
        <label for="content">Content:</label>
        <textarea name="content" rows="8" class="form-
control">{{ story.content }}</textarea>
    </div>

    <div class="form-group">
        <label for="category">Category:</label>
        <select name="category" class="form-control">
            {%
                for category in categories %}
                {% if story.category == category %}
                    <option value="{{ category|lowercase }}"
selected>{{ category }}</option>
                {% else %}
                    <option value="{{ category |
lowercase }}">{{ category }}</option>
                {% endif %}
                {% endfor %}
            </select>
    </div>

```

```

<div class="form-group">
    <label for="name">Slug:</label>
    <input type="text" name="slug" class="form-control"
value="{{ story.slug }}" />
</div>

<p><button type="submit" class="btn btn-primary">Submit</
button></p>
</form>

<p><a href="/admin">Cancel</a></p>

{ % endblock %}

```

Before we connect that to a route, we need to write the POST part of that route too. This is trickier, but only because not all the field values are required.

In the admin\_edit.stencil there are five fields to complete: title, strap, content, category, and slug. The first four are required, because a post wouldn't make sense without them. But the fifth one – slug – can be calculated by the system. In fact, unless it's something *special* then it's preferable to have the slug calculated by the system, because it will base it on the page title.

In previous projects we've used a `getPost()` function to extract URL-encoded fields that were posted. We provided it a list of fields that were required, and it made sure they were all present and had at least one non-whitespace character. We're going to be using `getPost()` again here, but we can't specify "slug" in there because the user might not provide a value.

If you thought *that* was confused, it's only just the beginning. We need to attempt to read "slug" from the raw POST values that we were given, but we still need to trim any whitespace and ensure that it's not empty. If it *is* empty, we're going to use the post's title instead.

Regardless of whether the user entered one or we're using the title, we need to do the actual work of converting it to a URL slug – converting it from "my awesome iPhone 7 review" to

“my-awesome-iphone-7-review”. Back at the beginning of this project, you added a dependency on my SwiftSlug package, which does exactly this job: it attempts to convert a string to a slug through a `convertedToSlug()` method on strings. However, it’s a *throwing* method, because it’s possible the conversion might fail or might end up being empty.

Putting that all together, we need to attempt to read the slug from raw POST fields, remove any whitespace, then check if it has any characters; if it doesn’t we’ll use the title instead. Regardless of what’s used, we need to try converting it to a valid slug, but if that fails we’ll just use whatever was entered because it’s better than nothing.

Translating all that into Swift becomes the following:

```
let slug =  
    rawPost["slug"]?.trimmingCharacters(in: .whitespacesAndNewlines)  
)  
  
if let unwrappedSlug = slug, unwrappedSlug.characters.count > 0  
{  
    fields["slug"] = unwrappedSlug  
} else {  
    fields["slug"] = fields["title"]!  
}  
  
fields["slug"] = (try? fields["slug"]!.convertedToSlug()) ??  
fields["slug"]!
```

The rest of the method is much easier: we call the `post()` method and pass it all the fields we have, then see what comes back. If it’s a success, we’ll redirect the user back to “/admin”, but if it *isn’t* a success we’ll do something new:

```
pageContext["story"] = rawPost  
try response.render("admin_edit", context: pageContext).end()
```

That re-renders the `admin_edit.stencil` template, passing in the same story that the user submitted. In practice, that means they’ll be back on the editing page with all their edits intact,

and they can correct their problem – a much nicer experience than just seeing an error page.

One last thing before I show you the code: this route is for editing articles as well as creating new articles, so we're going to post to two different back-end URLs. If we're editing we'll post to “/story/(storyID)”, but if we're creating a new article we'll use “/story/create” instead, like this:

```
let postResult: JSON?

if let storyID = request.parameters["id"] {
    postResult = post("/story/\(storyID)", fields: fields)
} else {
    postResult = post("/story/create", fields: fields)
}
```

OK, that's everything – go ahead and add this method to the **FrontEnd** class now:

```
func postAdminEdit(request: RouterRequest, response:
RouterResponse, next: () -> Void) throws {
    var pageContext = context(for: request)
    pageContext["title"] = "Edit"

    guard let values = request.body else { return }
    guard case .urlEncoded(let rawPost) = values else { return }

    if var fields = request.getPost(fields: ["title", "strap",
"content", "category"]) {
        let slug =
rawPost["slug"]?.trimmingCharacters(in: .whitespacesAndNewlines
)

        if let unwrappedSlug = slug,
unwrappedSlug.characters.count > 0 {
            fields["slug"] = unwrappedSlug
        } else {
```

```

        fields["slug"] = fields["title"]!
    }

    fields["slug"] = (try?
fields["slug"]!.convertedToSlug()) ?? fields["slug"]!

let postResult: JSON?

if let storyID = request.parameters["id"] {
    postResult = post("/story/\(storyID)", fields: fields)
} else {
    postResult = post("/story/create", fields: fields)
}

if let _ = postResult {
    try response.redirect("/admin")
    return
}
}

pageContext["story"] = rawPost
try response.render("admin_edit", context:
pageContext).end()
}

```

Now, that code won't build just yet, because we haven't written the `getPost()` method. You might be tempted to reach back to one of the earlier projects and copy it from there, but this time we're going to do something a little different.

Every time we've used `getPost()` so far, it's had this line near the start:

```
guard case .urlEncoded(let body) = values else { return nil }
```

That's ideal for handling URL-encoded forms like the one being submitted to "/admin/

edit/:id?”. However, our `post()` method sends data from the front end to the back end by posting JSON, not URL-encoded fields, which means our current `getPost()` method is insufficient – it will handle one kind of post but not another.

To resolve this, we’re going to create a new `getPost()` method that handles both URL-encoded values and JSON-encoded values. This takes a little bit of juggling, but the end result is much more useful. And following the theme for this project, we’re going to write this in an *organized* way – we’re going to make an extension on `RouterRequest`.

The “juggling” is mostly about handling HTML encoding. As you’ve seen in the past, URL-encoded values need to have their HTML encoding removed before they can be used. This *isn’t* true of JSON data, because it wasn’t submitted with URL encoding. In order to handle both, we’re going to declare two constants up front, like this:

```
let removeHTMLEncoding: Bool  
let submittedFields: [String: String]
```

The first will be true if we need to remove HTML encoding from each value, and the second will contain all the fields we need to parse. Then we’ll check what type of data we have: if we have URL-encoded form data then we can assign it straight to `submittedFields` because it’s already a `[String: String]` dictionary, and we can also mark `removeHTMLEncoding` as true. On the other hand, if we have JSON then we need to safely typecast its data to a dictionary, because it provides a `SwiftyJSON` object instead. We also need to set `removeHTMLEncoding` to be false, because it isn’t needed.

Here is that in code:

```
if case .urlEncoded(let body) = values {  
    submittedFields = body  
    removeHTMLEncoding = true  
} else if case .json(let body) = values {  
    // we need to convert the JSON before we can use it  
    guard let unwrapped = body.dictionaryObject as? [String:  
String] else { return nil }  
    submittedFields = unwrapped
```

```

        removeHTMLEncoding = false
    } else {
        // we received something else - run away!
        return nil
    }
}

```

Once we've homogenized our data into a **[String: String]** dictionary, most of the rest of the method is the same. The only difference is that we need to read the **removeHTMLEncoding** value when saving each field into the result, like this:

```

if removeHTMLEncoding {
    result[field] = value.removingHTMLEncoding()
} else {
    result[field] = value
}

```

That's how it all works, so it's time to piece all the code together. Please add a new file called RouterRequest+Post.swift. If you're using Xcode, that means right-clicking on the project11 group and choosing New File > Swift File; if you're not using Xcode, just create the file in your Sources directory.

Either way, give it this content:

```

import Foundation
import Kitura

extension RouterRequest {
    func getPost(fields: [String]) -> [String: String]? {
        // make sure we have some values to parse
        guard let values = self.body else { return nil }

        let removeHTMLEncoding: Bool
        let submittedFields: [String: String]

        if case .urlEncoded(let body) = values {

```

```

    // we're URL-encoded
    submittedFields = body
    removeHTMLEncoding = true
} else if case .json(let body) = values {
    // we're JSON-encoded; convert it to a dictionary
    guard let unwrapped = body.dictionaryObject as?
[String: String] else { return nil }
    submittedFields = unwrapped
    removeHTMLEncoding = false
} else {
    // we're something else; bail out
    return nil
}

// prepare our list of finished fields
var result = [String: String]()

for field in fields {
    // if this field exists, removing any whitespace
    if let value =
submittedFields[field]?.trimmingCharacters(in: .whitespacesAndNewlines) {
        // and it has some characters
        if value.characters.count > 0 {
            // add it to our result, removing encoding as
needed
            if removeHTMLEncoding {
                result[field] = value.removingHTMLEncoding()
            } else {
                result[field] = value
            }
            continue
        }
    }
}

```

```

        return nil
    }

    return result
}
}

```

That code *still* won't build, because we're using the `removingHTMLEncoding()` method and we haven't added that yet. Fortunately, that *hasn't* changed, but we *are* going to put it into its own file to keep our project organized. Repeat the process from above to create a new file, this time calling it `String+HTMLEncoding.swift`, then give it this content:

```

import Foundation

extension String {
    func removingHTMLEncoding() -> String {
        let result = self.replacingOccurrences(of: "+", with: "")
    }
}

```

Hopefully your code should compile cleanly again, so all that's left to do on the front end is add routes for our editing page. Add these two lines directly after the `self.getAdminHome` route in the `FrontEnd` routes list:

```

adminRouter.get("/edit/:id?", handler: self.getAdminEdit)
adminRouter.post("/edit/:id?", handler: self.postAdminEdit)

```

The front end is now feature complete and done; all that's left is to create a back-end route to handle article editing and creation. As you've seen, this is the same route: if we're creating a new one then we post to "/story/create", and if we're editing an existing one then we post to "/story/1" or whatever its ID number is. So, the route will be POST to "/story/:id".

This final method needs to:

1. Check that “id” is set in the route parameters. It will either be a number or the string “create”, but it does need to be there.
2. Use **getPost()** to ensure that all five values are present, otherwise send back that it’s a bad request.
3. Connect to MySQL and look up the category they sent. We need to convert “News” into “2”, for example. If the category doesn’t exist, it’s a bad request.
4. Execute either a **UPDATE** or an **INSERT INTO** query, depending on whether we are editing or creating an article.
5. Send back a success or failure message depending on the result of the query.

Two parts of that are worthy of further discussion before I show you the full code.

First, converting categories from their name, e.g. “News”, into their ID, e.g. “2”. This requires executing an SQL query, then reading the first row that came back, then reading the first field in the first row, and using that value. If the query fails, or if no rows come back, or if the field doesn’t exist in the first row, then it fails. That’s a fair chunk of code to write when all you care about is a single value, which is why in my own code I nearly always write a helper method to make it more pleasant.

So, we’re going to write that helper method first. It’s called **singleQuery()**, and it’s going to be an extension on the **Database** class. This runs a **SELECT** query that reads one value from one row, and it returns it. If there are more rows or more fields, they all get ignored – it just returns the first field in the first row.

Create a new file called Database+SingleQuery.swift, then give it this content:

```
import Foundation
import MySQL

extension Database {
    func singleQuery(_ query: String, _ values: [NodeRepresentable] = [], _ connection: Connection? = nil) -> Node? {
```

```

        do {
            return try self.execute(query, values,
connection).first?.first?.value
        } catch {
            return nil
        }
    }
}

```

As you can see, it uses optional chaining so that if any part of the code fails it will return nil. In short, when you call that, you either get back your value or you get nil, no matter what happens inside.

The second thing that is worthy of further discussion is handling the difference between an **INSERT INTO** query and an **UPDATE** query. I'm sure you hate duplicating code as much as I do, so I've solved this in a way that duplicates nothing.

First, we're going to create a **query** string constant, like this:

```
let query: String
```

That's going to hold one of the two queries, depending what path we taken.

Second, we're going to create an array of ordered fields, which is our **fields** dictionary except placed into a precise order so it lines up with all the question marks we're going to use in our queries:

```
var orderedFields = [fields["title"]!, fields["strap"]!,
fields["content"]!, fields["category"]!, fields["slug"]!]
```

There are five values there, which is perfect for creating a new story. But if we're *updating* a story, we need *six* values – we need to add the ID of the story we're updating. To make that work, we're going to check whether the story ID is set to “create”, and if it is craft an SQL query with five question marks, one for each field. If it *isn't* “create”, then we're going to craft a query with *six* question marks, and append the story ID to the **orderedFields** array, like

this:

```
if storyID == "create" {
    query = "INSERT INTO `posts` (`title`, `strap`, `content`,
`category`, `slug`, `date`) VALUES (?, ?, ?, ?, ?, ?, NOW());"
} else {
    query = "UPDATE `posts` SET `title` = ?, `strap` = ?,
`content` = ?, `category` = ?, `slug` = ? WHERE `id` = ?;"
    orderedFields.append(storyID)
}
```

Using this approach, both **query** and **orderedFields** can be used interchangeably because they are both valid no matter what path was taken.

Add this final method to the **BackEnd** class now:

```
func postStory(request: RouterRequest, response:
RouterResponse, next: () -> Void) throws {
    defer { next() }

    // make sure we have a valid story ID, or "create"
    guard let storyID = request.parameters["id"] else {
        response.status(.badRequest).send("Missing story ID.")
        return
    }

    // all five fields are required here
    guard var fields = request.getPost(fields: ["title",
"strap", "content", "category", "slug"]) else {
        response.status(.badRequest).send("Missing required
fields.")
        return
    }

    // connect to MySQL
```

```

let (db, connection) = try connectToDatabase()
let categoryQuery = "SELECT `id` FROM `categories` WHERE
`name` = ?"

// find the category ID for the category they want to use
if let categoryID = db.singleQuery(categoryQuery,
[fields["category"]!], connection)?.int {
    fields["category"] = String(categoryID)
} else {
    // it doesn't exist - bail out!
    response.status(.badRequest).send("Unknown category.")
    return
}

// craft an SQL query and field list
let query: String
var orderedFields = [fields["title"]!, fields["strap"]!,
fields["content"]!, fields["category"]!, fields["slug"]!]

if storyID == "create" {
    query = "INSERT INTO `posts` (`title`, `strap`,
`content`, `category`, `slug`, `date`) VALUES (?, ?, ?, ?, ?, ?,
NOW());"
} else {
    query = "UPDATE `posts` SET `title` = ?, `strap` = ?,
`content` = ?, `category` = ?, `slug` = ? WHERE `id` = ?;"
    orderedFields.append(storyID)
}

do {
    // run the query
    _ = try db.execute(query, orderedFields, connection)

    // if we're still here it means it worked; send back the

```

```
all clear

    let result = [ "status": "ok" ]
    response.status(.OK).send(json: JSON(result))

} catch {
    // something went wrong – send back an error
    response.status(.notFound).send("Unknown story ID.")
    return
}

}
```

Finally, just add the route for that method, and the back end is complete. Add this after the GET route for “/story/:id”:

```
router.post("/story/:id", handler: self.postStory)
```

Finished! Go ahead and give it a try – being able to add or edit posts from the same route is a helpful time saver.

## Wrap up

I told you this was going to be a big project, but I hope you can see why: we’re now sending and receiving content between two Kitura servers, we have Stencil filters for Markdown and link formatting, we handle JSON parsing in POST requests, there’s a subrouter for admin pages, and more.

More importantly, I hope this project gave you an idea of how to organize your code more effectively. Putting things in main.swift really is fine while you’re learning, but you’re past that point now and it’s time to start thinking about maintainability.

And of course, let’s not forget that you have another server-side Swift project under your belt. There’s lots you can do to improve it – just see the homework section below! – but at the same time it’s given you another chance to practice your skills on a real-world project, and maybe, just maybe, you’re feeling a bit better about SQL.

## Homework

This project is so big it’s hard to know where to start, but I think a good place would be to make the “/category/:categoryname” path work. This is triggered by using one of the options in the top-right drop-down menu, and should return all articles that belong to a specific category.

Second, have a think about the way slugs are handled. I’ve resolved everything in the front end, but what happens if the user enters a slug that has already been taken by a different article? See if you can move the slug generation and validation to the back-end, which would allow you to ensure that new slugs are always unique.

Finally, look back at the code you wrote in project 5, Meme Machine. That let users upload pictures to the server, including resizing them to generate thumbnails. If you’re looking for a real challenge, add an image upload feature to this CMS so that writers can embed images directly into their articles. Once you’ve made image uploads work, let users copy the URL to an image to their clipboard so they can use it in their Markdown. In case you were wondering, the Markdown syntax for embedding an image is as follows:

```
![Alt text for image](/path/to/image.jpg)
```

# **Chapter 12**

## Testing

## Setting up

This last technique project is going to cover testing using the XCTest framework. This might seem like a strange choice to end up, but it's an important one.

Obviously, yes, having tests in your app is always a good thing, but with server-side Swift it's actually easier than ever to write tests – far easier than when you're working with iOS or macOS. A number of our projects were API-based: we wrote back-ends that deliver JSON for consumption elsewhere. This is very different to the kind of apps you normally craft on iOS, where testing is more complicated. On iOS, you need to write tests for things like “if the user taps button A and selects option B in the alert controller...” and that's *hard*, even with the UI test recording feature in Xcode.

In server-side Swift it's quite different: we can write code to query an API just like a real end user would, and we evaluate the results directly. This means writing calls to our back ends that check they send data, writing calls that send new data to save, writing calls that send *bad* data to generate errors, and more. It's all testable, if you take the time to think it through.

There are a number of ways to write tests for web code, but I want to show you what is by far the easiest. Project 11, the one you just finished, was neatly split up into front-end and back-end code, with the front-end providing useful helper methods like `get()` and `post()`. In this project we're going to write tests to hijack those calls: we're going to start a back end instance serving real data, then start a dummy front-end instance to send and receive data. We can then use XCTest to evaluate the responses from the server to make sure they match our expectations.

I should add, there are other, more advanced ways of testing APIs. For example, in a more advanced app, it wouldn't be uncommon to uncouple the data model from the back end, so you could write tests for the model layer without having to make requests.

Anyway, we're going with the simple-but-effective option here, just to help you started. Take a copy of your project11 folder, rename it project12, and let's get started!

## Bootstrapping XCTest

All the projects we've created so far have had a Tests directory, but it has always been empty. This is a strange quirk of the Swift package manager: if you create a framework package it adds some default code in the Tests folder to help get you started, but in executable packages you just get an empty folder. Even more annoyingly, the skeleton code needs to be structured in a particular way, otherwise it won't work correctly.

To save you some time, I created a skeleton test set up for you by creating a dummy framework and chopping out the bits that aren't needed. I've included it into the project files with this book, so please look in there now for project12-files, and copy its entire Tests folder into your project directory, overwriting your empty Tests folder.

**Tip:** You'll notice my files are called "project11Tests" even though this is clearly project 12. Remember, we're writing tests for project 11's code!

To check that everything is working OK, run the command "swift test" from your project directory. This will build the project code, then build and run the test code. If you're using macOS, you will need to pass the same build flags you were using during project 11, e.g. "swift test -Xlinker -L/usr/local/lib".

When the test finishes, you'll see a report: how many tests were run, and how many failed. With this skeleton test bed, you should see 1 and 0 respectively. Let's dive in with some real tests: open the Tests directory and then look for "project11Tests" inside it. Inside *that* you'll see project11Tests.swift, which is where all our tests will live, so open that in your preferred editor. Here's what you'll find:

```
import Kitura
import SwiftyJSON
import XCTest
@testable import project11

class project11Tests: XCTestCase {
    override func setUp() {
}
```

```

override func tearDown() {
}

func testHello() {
    XCTAssertEqual("Hello", "Hello", "Hello should equal
Hello")
}

static var allTests : [(String, (project11Tests) -> () throws -> Void)] {
    return [
        ("testHello", testHello),
    ]
}
}

```

There are five parts of interest in there.

1. The **@testable import project11** line is Swift’s magic way of importing a project for testing, even though it means breaking all sorts of protection access so that our tests can poke around in the project11 classes freely.
2. The **setUp()** method is called once before each test. We’re going to use that to launch a fully reset back end, ready for testing.
3. The **tearDown()** method is called after each test has finished. We’re going to use that to stop Kitura, so we can restart it fresh for the next test.
4. The **testHello()** method is a test that will be run by the “swift test” command. It starts with the word “test”, accepts no parameters, and returns no values – that’s required.
5. The **allTests** variable is how XCTest knows which tests to run. Whenever we write a new test, you need to add it to the array just like you see “testHello” is done.

Now, you might wonder why we bother starting and stopping everything between each test, and what “fully reset” means. The answer is that testing is only useful if the tests are isolated from each other: even if you write five tests that do similar things, you need to assume they

could be run in any order, so you shouldn't ever rely on the results of previous tests.

As for “fully reset”, we’re going to insert a new route into the project11 back end that wipes the “posts” table, then call that during the **setUp()** method. In fact, let’s do that now – open BackEnd.swift in your project12 copy of project11, then add this method:

```
func reloadData(request: RouterRequest, response: RouterResponse, next: () -> Void) throws {
    let (db, connection) = try connectToDatabase()
    try db.execute("DELETE FROM `posts`;", [], connection)
    try response.status(.OK).end()
}
```

Now add this route directly above the **return router** line in the list of back-end routes:

```
router.get("/admin/_topsekrit_/reset", handler: reloadData)
```

Now, obviously that’s *not* the kind of thing you want lying around in your code, and certainly not unless you’ve added token-based authentication to restrict access to “/admin”. However, it’s important to have some way of resetting your environment during tests, and it’s fine for the purposes of this project.

Now that we have a simple reset hack built in, we can add some test code to project11Tests.swift. First, we need a property that contains an instance of our **FrontEnd** class, so we have a simple way of making requests to the back end. Add this property now:

```
var frontEnd: FrontEnd!
```

Next, we’re going to need to create test stories in several places, so rather than repeat the code it’s easier to add a dedicated method. All this needs to do is create a dictionary of data, just like we’re get from creating an article in our admin system. Add this method now:

```
func createStory() -> [String: String] {
    let title = "Hello, world"
    let strap = "This is a strap"
```

```

let content = "This is some content"
let category = "Reviews"
let slug = "hello-world"

return ["title": title, "strap": strap, "content": content,
"category": category, "slug": slug]
}

```

Finally, we're going to fill in the `setUp()` and `tearDown()` methods with some real code. The `setUp()` method is going to create a `BackEnd()` object, attach it to a server, then start Kitura running. We've been using `Kitura.run()` to do that in our previous projects, but here we're going to use `Kitura.start()` – it does the same thing, but it returns immediately rather than waiting until the server terminates. This means our code can carry on running, which is obviously important in these tests!

Once the back end is up and running, we'll create a `FrontEnd` object and assign it to our `frontEnd` property, so we have a simple way of making calls to the back end. Finally, we'll tell that `frontEnd` property to call our secret reset route on the back end, forcing the “posts” table to be clearer.

Here's the new code for `setUp()`:

```

override func setUp() {
    let backend = BackEnd()
    Kitura.addHTTPServer(onPort: 8089, with: backend.router)
    Kitura.start()

    frontEnd = FrontEnd()
    _ = frontEnd.get("/admin/__topsekrit__/reset")
}

```

The code for `tearDown()` is much easier: it just needs to call `Kitura.stop()` to stop all running servers. Add this now:

```
override func tearDown() {
```

```
    Kitura.stop()
}
```

That's all our set up written, so it's time to add some real tests...

# Testing our routes

We have one test right now, which is this:

```
func testHello() {  
    XCTAssertEqual("Hello", "Hello", "Hello should equal Hello")  
}
```

That runs an XCTest function called **XCTAssertEqual()**: it checks whether the first parameter (“Hello”) is equal to the second parameter (also “Hello”). If the two parameters are different XCTest considers the test to be a failure, and it prints out whatever you passed in parameter 3 – in our case it’s what we *expected* to be the case.

Let’s replace that with a real test. Before each test is run, our **setUp()** method runs our secret data wiping route, which means the “posts” table should be empty by default. That’s perfect for a test, so let’s start there.

For this first first, **testNoStories()**, we need to call **frontEnd.get()** with the route “/stories” to fetch all the stories in the system. We can then use **arrayObject** to convert it to an array, and check that the array’s **count** property is 0. If fetching “/stories” fails for some reason, we can call a different XCTest function: **XCTFail()**, which is a forced failure.

Add this method now:

```
func testNoStories() {  
    if let result = frontEnd.get("/stories")?.arrayObject {  
        // we got data back – make sure the array is empty  
        XCTAssertEqual(result.count, 0, "There should be no  
        stories")  
    } else {  
        // fetch failed for some reason  
        XCTFail("Fetching stories failed")  
    }  
}
```

Make sure you modify the **allTests** variable to include the **testNoStories()** test, like

this:

```
( "testNoStories" , testNoStories ) ,
```

Now go ahead and run “swift build”, remembering to include linker parameters if you’re on macOS. All being well you should see the new test run and pass. If you’re curious what a failed test looks like, try changing the **XCTAssertEqual()** line to this:

```
XCTAssertEqual(result.count, 10, "There should be no stories")
```

Because **result.count** will be 0, comparing to 10 will create a failing test. Make sure you change the 10 back to 0 before continuing, otherwise the rest of this chapter will be most confusing!

Next, let’s write a test to ensure that a non-existent story fails to load. Add this:

```
func testNonexistentStory() {  
    let result = frontEnd.get("/story/  
fizzbuzz")?.dictionaryObject  
    XCTAssertNil(result, "Nonexistent stories should not load")  
}
```

**XCTAssertNil()** is another new XCTest function: it considers a test to have passed if the first parameter (**result**) is nil. Again, add that test to your **allTests** variable, like this:

```
( "testNonexistentStory" , testNonexistentStory ) ,
```

Once that’s done, re-run “swift build” to give it a try.

Let’s try something more complicated: creating a story. This is more complicated because not only do we need to post values to the server and check the result, but to be *really* sure that it’s worked we need to fetch the “/stories” route again and make sure it has precisely 1 item in the resulting array. This route will use our **createStory()** method to create an example story.

Add this method now:

```

func testCreateStory() {
    let story = createStory()
    let createResult = frontEnd.post("/story/create", fields:
        story)

    XCTAssertNotNil(createResult, "Creating a story failed")

    if let result = frontEnd.get("/stories")?.arrayObject {
        XCTAssertEqual(result.count, 1, "There should be 1
story")
    } else {
        XCTFail("Fetching stories failed")
    }
}

```

Yes, that's yet another XCTest function: **XCTAssertNotNil()**. It's the counterpart of **XCTAssertNil()**, and considers a test to have passed if the first parameter is anything except nil.

It's always important to create tests for *invalid* data as well as *valid* data, so here's a test that attempts to create a story with missing fields:

```

func testCreateBadStory() {
    let createResult = frontEnd.post("/story/create", fields:
        ["title": "Meh"])
    XCTAssertNil(createResult, "Creating an invalid story should
fail")
}

```

As you can see, it expects **createResult** to be nil, because the server should refuse to create the story.

Remember, each test needs to be totally isolated from the others. So, if you want to test *updating* a story, you need to first *create* a story, then retrieve it, then update it, because the database will start out completely blank.

Here's how that looks in a test:

```
func testUpdateStory() {
    // first, create a story
    var story = createStory()
    let _ = frontEnd.post("/story/create", fields: story)

    // next, fetch that story from the back end
    if let result = frontEnd.get("/stories")?.arrayObject?.first
        as? [String: Any] {
        // figure out what ID we need to post back
        let id = result["id"]

        // make a small change
        story["title"] = "Modified"

        // then post it back
        let updateResult = frontEnd.post("/story/\(id)", fields:
            story)

        // ensure the result was valid
        XCTAssertNotNil(updateResult, "Updating a story failed")
    } else {
        XCTFail("Fetching stories failed")
    }
}
```

Finally, here's a test that fails:

```
func testUpdateNonexistentStory() {
    let story = createStory()
    let updateResult = frontEnd.post("/story/fizzbuzz", fields:
        story)
    XCTAssertNil(updateResult, "Updating a nonexistent story")
```

```
should not work")
}
```

Take a look at that: it's attempting to update a story with the ID "fizzbuzz" even though the database is empty. We're testing using **XCTAssertNil()**, meaning that we expect to get nil back because the server refused the update. But as you'll see, the test fails because **updateResult** comes back with a success - the server claims everything went just fine.

Yes, that's a bug. Yes, it's intentional. It's almost like I planned ahead...

Now it's over to you: your tests have found a bug in project 11: the server doesn't actually check that an ID exists when updating an article. Can you fix that in project 11 to make your tests pass? Good luck!

## Wrap up

Testing isn't glamorous or fun, but it *is* essential, and it's surprisingly easy to do in server-side Swift because so much of our work is passing JSON to and from various places. You don't have to go full-on into test-driven development to get the benefit from testing: start small, do your best, and build upwards.

Once you finally manage to get good test coverage, you'll find it makes making changes live far easier – and far less stressful. As long as your tests test what you *think* they test, you'll be in good shape if they all pass.