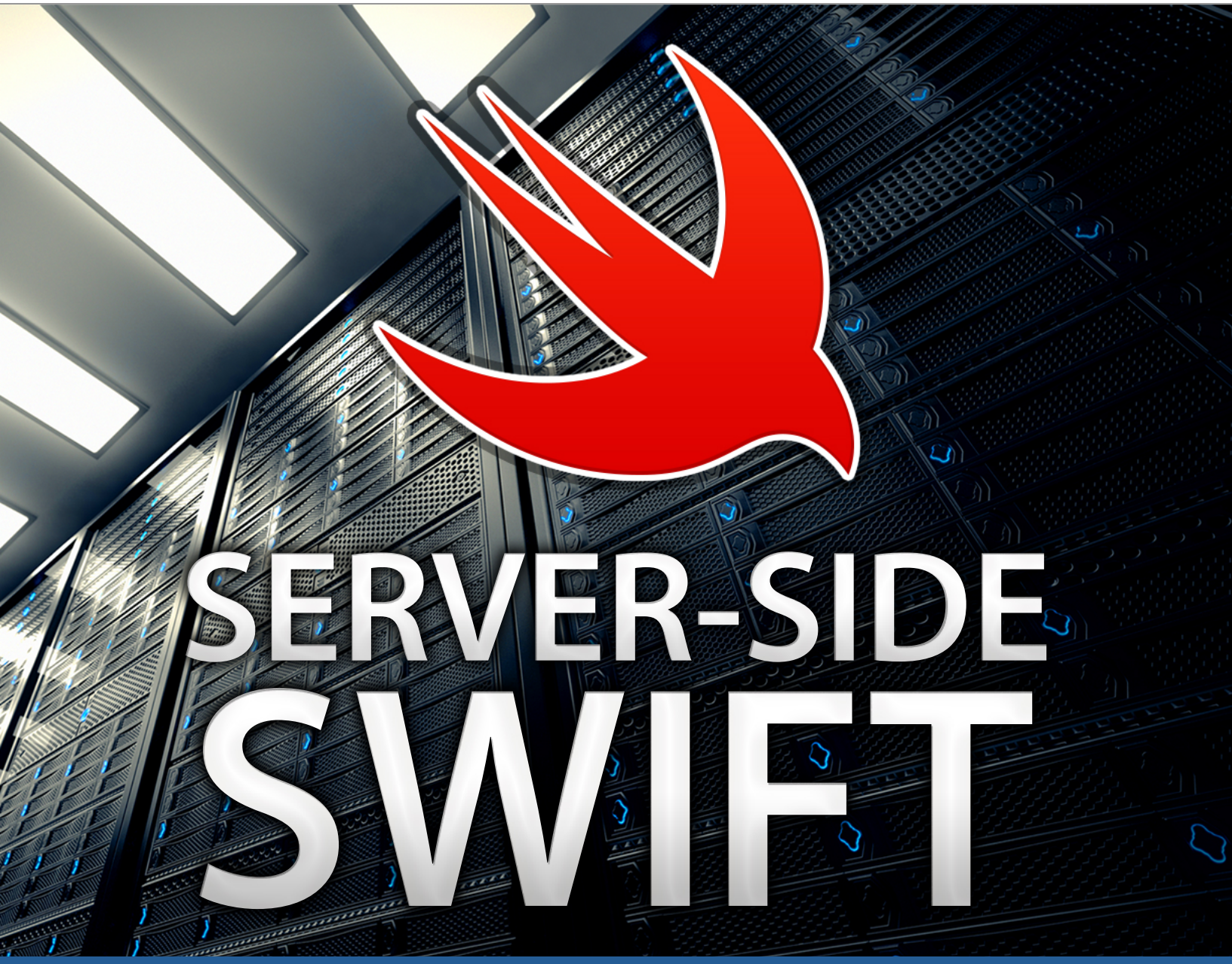
HACKING WITH SWIFT



SERVER-SIDE SWIFT

COMPLETE TUTORIAL COURSE

Learn to make web apps with real-wo
Swift projects

FREE SAMPLE

Paul Hudson

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 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
```

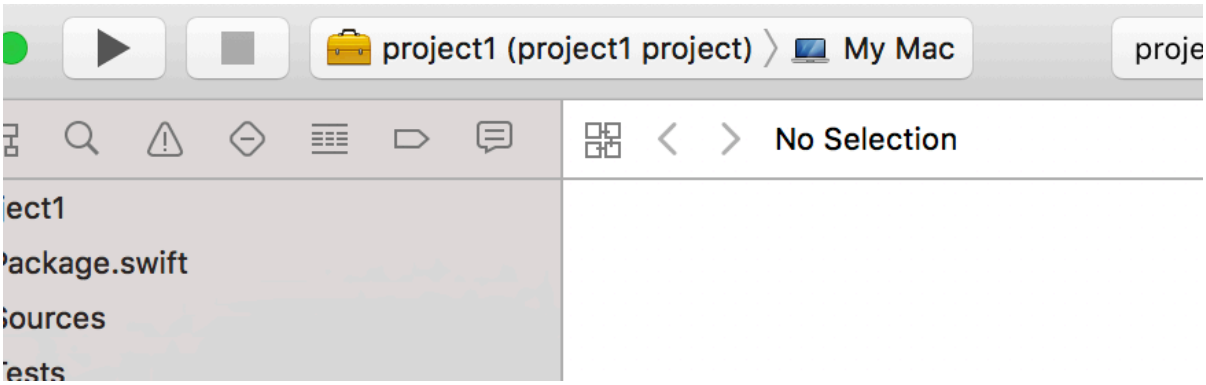
It might take a few seconds to complete, but when it finishes you'll see project1.xcodeproj in your current directory. You can open that in Xcode by running `open project1.xcodeproj` or by double-clicking it in Finder.

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

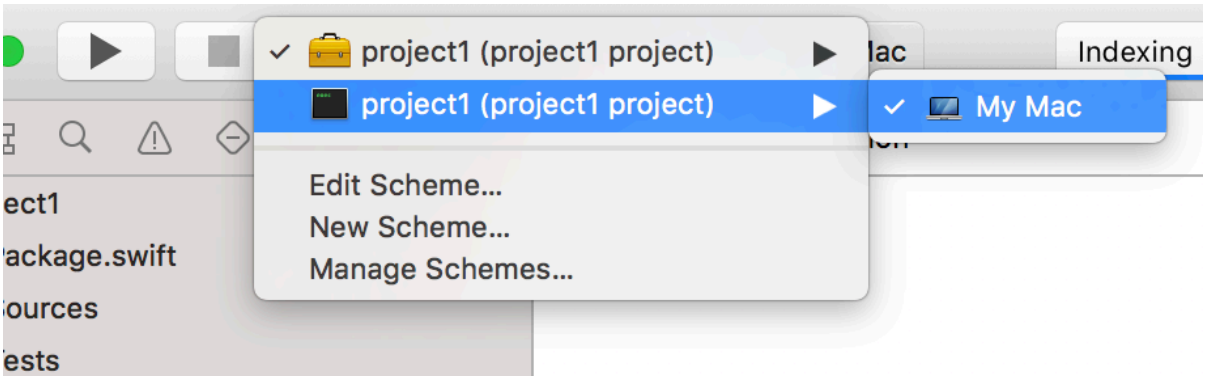
```
print("Hello, world!")
```

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 ("/team/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.

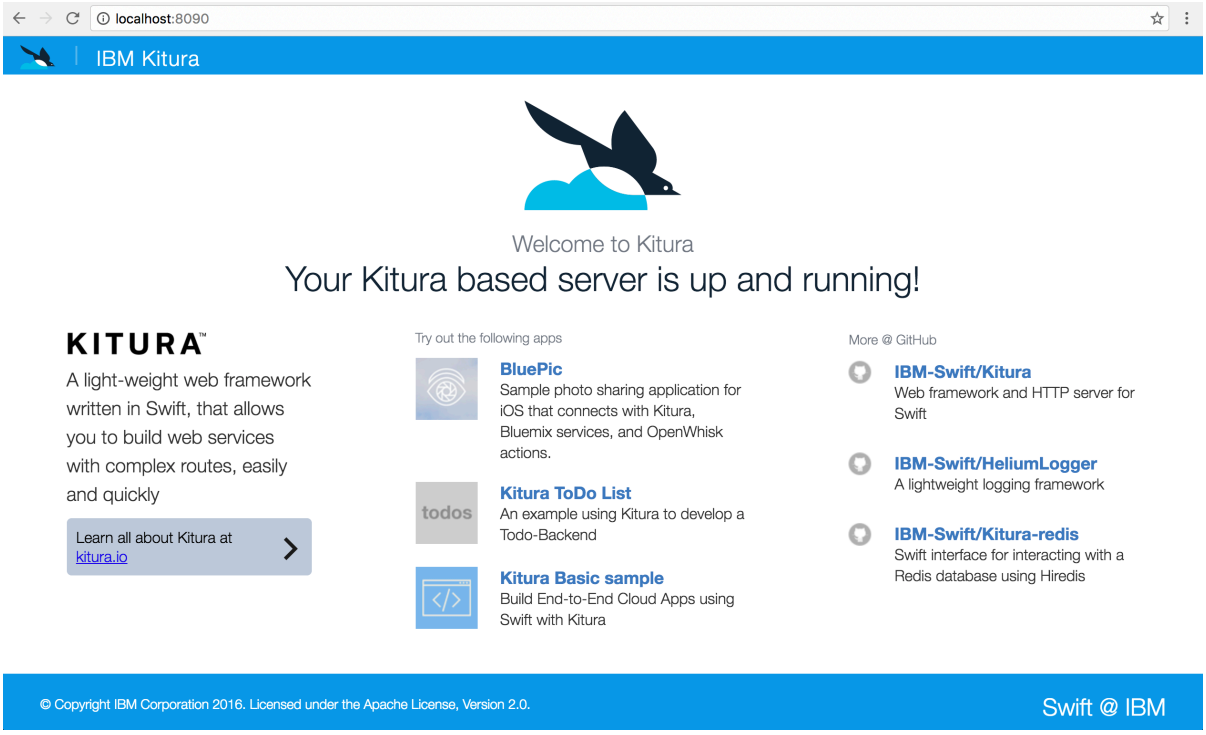
Build and run your code now. If you're using the command line in Docker or directly on your Mac you should run `swift build` followed by `.build/debug/project1`. If you're using Xcode, press the Play button.

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 called "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 &copy; 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!