HACKING WITH SWIFT

SERVER-SIDE SWIFT

VAPOR EDITION

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 Vapor toolbox, how to route users to different parts of your code depending on the URL they enter, 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 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 unless you ask for it: you create your project using the Vapor toolbox, which you should already have installed in the introduction.

The Vapor toolbox isn't strictly required, but it does help simplify tasks such as creating projects and deploying to real servers. Underneath the toolbox is the Swift package manager, which is responsible for installing third-party frameworks and their dependencies.

We're going to create a new project called "project1", then build it and run it straight from the command line. First, though, we need somewhere to put this project and all future projects. To make things easy I'm going to show you how to create a directory called "server" on your desktop, and you can then create all new projects in there going forward.

Launch the terminal app and run these three commands:

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

The first one changes to your desktop directory – if you're already there you should skip the

first command. The second creates a new directory called “server”, and the third changes into that new directory.

Now we can create our new project, so run these commands:

```
vapor new project1 --template=twostraws/vapor-clean
cd project1
```

That creates a new Vapor project called “project1” and changes into its directory. A couple of seconds will pass as the Vapor toolbox clones the default project repository, but eventually you’ll see a large ASCII Vapor logo on your screen – that means everything worked!

Note: I used **--template=twostraws/vapor-clean**, which is a template I created specifically for this book. At the time of writing, the official Vapor 3 templates include a lot of code that forms a sample app, which means you need to delete an awful lot of code before you can start writing your own. My template deletes all that code for you, and is effectively the smallest amount of code that sets up a working Vapor 3 environment.

Running those commands has given us a simple project. We’ll look at what it does shortly, but first let’s check that it all works. Run this command:

```
vapor build
```

That instructs Vapor to build the source code for our program, which in turn means it must fetch all the dependencies – that’s the code for Vapor itself in this instance.

We can also run the executable straight from the command line, using this command:

```
vapor run
```

If everything worked, you should see “Running project1…” and “Server starting on localhost: 8080”. That wasn’t so hard, was it?

These workers are there to respond to web requests, and now they are all started it means

Vapor is ready to serve web pages. Press Ctrl+C to terminate your Vapor server when you're finished admiring it.

Swift packages explained

Underneath the Vapor toolkit is the Swift Package Manager. This 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:

```
// swift-tools-version:4.0
import PackageDescription

let package = Package(
    name: "project1",
    dependencies: [
        // A server-side Swift web framework.
        .package(url: "https://github.com/vapor/
vapor.git", .upToNextMinor(from: "3.1.0")),
    ],
    targets: [
        .target(name: "App", dependencies: ["Vapor"]),
        .target(name: "Run", dependencies: ["App"]),
        .testTarget(name: "AppTests", dependencies: ["App"]),
    ]
)
```

The Swift package manager doesn’t have a neat way to add or modify dependencies; you

literally need to rewrite Package.swift's source code as needed.

For this project we're going to add a second dependency: the Leaf framework. This is Vapor's template engine, which is what allows it to render HTML efficiently.

Adding dependencies to Package.swift isn't a pleasant experience because often the Git repository name and the dependency name don't match. Don't worry, though – I'll be showing you exactly what to type for every project in this book.

First, add this line below the existing **.package(url:)** line:

```
.package(url: "https://github.com/vapor/
leaf.git", .upToNextMinor(from: "3.0.0"))
```

That tells the package manager that it must download code from the Leaf repository on GitHub.

Now find where it says **dependencies: ["Vapor"]** and change it to **dependencies: ["Vapor", "Leaf"]** – that tells the package manager that our Vapor app relies on Leaf to run.

Your final Package.swift file should look like this:

```
// swift-tools-version:4.0
import PackageDescription

let package = Package(
    name: "project1",
    dependencies: [
        // 💧 A server-side Swift web framework.
        .package(url: "https://github.com/vapor/
vapor.git", .upToNextMinor(from: "3.1.0")),
        .package(url: "https://github.com/vapor/
leaf.git", .upToNextMinor(from: "3.0.0"))
    ],
```

```
    targets: [
        .target(name: "App", dependencies: ["Vapor", "Leaf"]),
        .target(name: "Run", dependencies: ["App"]),
        .testTarget(name: "AppTests", dependencies: ["App"]),
    ]
)
```

Save your changes, then run **vapor update** from the command line to have it download the Leaf repository.

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

- "circle.yml" is a pre-built configuration file for users of CircleCI, which is a cloud-based continuous integration platform.
- "cloud.yml" is a configuration file for users who want to deploy their app to the Vapor Cloud service.
- "LICENSE" is where you should write your tool's license, if you have one. It's set to be MIT with a copyright of "YOUR NAME HERE", so you should probably change that.
- "Package.resolved" stores the precise version numbers that were fetched as a result of your Package.swift file. This allows you to get the same package versions on your server as you have on your local machine.
- "Public" is where you can serve any static content, such as images and JavaScript.
- "Resources" is where you'll store any template files used with Leaf.
- "Sources" is where your source code lives. This contains a Run directory that has a small amount of bootstrapping code you should leave alone, plus an App directory where your code will live.
- "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 directory called ".build" that contains all your dependencies plus your own compiled code, and a hidden file called ".gitignore", which configures your source control system to ignore the ".build" directory because it's fetched dynamically.

You can edit your Swift code using any editor you like, but if you're using a Mac I suggest you use Xcode so you can benefit from features like code completion.

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.
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.

With that in mind, we're going to ask Vapor to create an Xcode project from our current package so you can try it out. It might have already asked you when you updated your packages, but if not run this command now:

```
vapor xcode
```

It might take a few seconds to complete, but when it finishes Vapor should ask whether you want to open the Xcode project now – please press Y then return to open it now.

Once you're inside Xcode, you'll find the code for your app is nested inside project1 > Sources > App. We're not going to change the code just yet, because I want to make sure you can build with Xcode first,

You'll notice that pressing Cmd+R to run your app does nothing. 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, go to the Product menu and choose Scheme > Run. With that change, Cmd+R will now function as expected: you can build and run your code straight from Xcode. That small change told Xcode we want to build Vapor's app wrapper, which in turn launches our code.

Note: This might sound obvious, but for the avoidance of doubt you should select your Mac for the target, and not any of the iOS simulators or devices that you may have configured.

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, and I've added warnings when adding files to make sure you do it correctly.

A brief tour of Vapor projects

When a user enters yoursite.com into their web browser, Vapor 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 Vapor works and, more importantly, *why* it works that way.

Let's start with the absolute basics: Vapor 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, Vapor users work with port 8080 by default – you can use that without an admin password just fine.

As you might imagine, Vapor'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 right now we have just one because that's what the Vapor template gave us.

Let's start by looking in main.swift, which you'll find in the Run folder. This is loose code – code that's not inside a function – because it's run when your app is launched. You don't need to edit this code, but you should be able to see that it loads the default configuration, environment, and services for Vapor and passes them to a **configure()** function.

You'll notice that both **App** and **Application** are used, which is a bit confusing at first. **App** is the name of our part of the Vapor project, all grouped inside the Sources/App group in Xcode. **Application** is a Vapor class responsible for listening for and responding to requests from users.

The line **try App.boot(app)** calls a **boot()** function in our code, passing in Vapor's application. This is responsible for any early setup code you might need, but realistically you won't use it much if at all. In fact, if you look inside boot.swift you'll find it's empty – it's there in case you need to run some code before your main app runs.

On the other hand, in configure.swift you'll see code to set up an **EngineRouter** object using the Vapor application, which then calls the **routes()** function from routes.swift. It's that router

we care about, because that's where all the magic happens: a router is responsible for directing user requests to various routes (e.g. /hello, or /about/contact), and those routes are where Swift code is run to generate output.

Tip: You'll need to modify this **configure()** function throughout this book, but you should always add your code beneath the existing code – leave the **EngineRouter** alone.

To finish our brief tour of the Vapor template, open routes.swift for editing. This defines a **routes()** function that contains all the routes you want to manage. For most of this book we're going to put our code right inside that function because it's the easiest thing to do while you're learning, but towards the end I'll show you how to organize your projects more neatly so they can scale better.

You'll see there's one route in there already:

```
router.get("hello") { req in
    return "Hello, world!"
}
```

That will respond to requests for **http://localhost:8080/hello** with "Hello, world!" Try it out now – press Cmd+R to build and run your program, then open a web browser and point it at **http://localhost:8080/hello**.

Routing user requests

At this point we have a working Vapor project that launches a web server on port 8080, and a simple route that sends back “Hello, world” when the user requests “hello”.

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”.

Vapor’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 Vapor handle the rest. This is all handled inside the **routes()** function of routes.swift.

Let’s start with the example route that prints out “Hello, world!” message:

```
router.get("hello") { req in
    return "Hello, world!"
}
```

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

First: the **get()** 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. There are other methods, but they aren’t used much over the web.

Second, **"hello"**. This is the path we want to attach code to, which in this case is the root of our site plus “hello” after it. This is the page that gets served when someone goes to **http://localhost:8080/hello**. That “:8080” part is necessary because Vapor is running on that port rather than the standard port 80. We’ll write more interesting paths soon enough, don’t worry.

Third, **req in**. Each path has a closure bound to it, which is the code to execute when the route is matched. This closure is given a single parameter that 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.

You'll notice the route closure doesn't specify a return type, but clearly it returns the string "Hello, world!". Routes can actually return a variety of data types, and most of the time Vapor will do the hard work for you of figuring out how to convert your data into something your user can read.

One thing you *can't* see in that code is that the closure provided to the "hello" route can throw errors without catching them. Any errors thrown in this way will propagate upwards: you didn't catch them, so Vapor will catch them for you automatically and represent them to users as server errors.

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 routes.swift to this:

```
import Routing
import Vapor

/// Register your application's routes here.
///
/// [Learn More →](https://docs.vapor.codes/3.0/getting-
started/structure/#routesswift)
public func routes(_ router: Router) throws {
    router.get { req in
        return "Welcome to Million Hairs"
    }
```

```
    router.get("staff") { req in
        return "Meet our great team"
    }

    router.get("contact") { req in
        return "Get in touch with us"
    }
}
```

Notice that the first route doesn't have a path attached to it – I just used **router.get**. This means it will be used for the root route, i.e. what gets loaded when the user goes to **http://localhost:8080** with no further path specified.

All three of those routes are trivial, which is why we don't need to specify a return type from the route closures. When we starting adding more complex routes in a few minutes, you'll find you need to specify the return type explicitly.

Build and run your code then try a couple of routes to make sure everything is working, and let's move on to something more interesting!

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 { req -> String in
    let html = """
    <html>
    <body>
    <h1>Welcome to Million Hairs</h1>
    </body>
    </html>
    """
    return html
}
```

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 Vapor's template engine is called Leaf.

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.

It's the job of template engines such as Leaf to load your HTML code, customize it based on any dynamic data you want to pass in, then send it to Vapor for rendering. This is a much more efficient way of working than trying to write your HTML directly inside your Swift code.

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.

Let's try it out now. We're going to use the Leaf template engine, which was designed by the Vapor team to work seamlessly with the rest of the system. This is done in two steps: we register a Leaf provider as a service for whole our application, then render individual templates as needed.

First, the Leaf provider. This is done inside the configure.swift file, where you'll find the **configure()** function. First, add an import for the Leaf framework to the top of the file, like this:

```
import Leaf
```

Now add these two lines to the end of the **configure()** function:

```
try services.register(LeafProvider())
config.prefer(LeafRenderer.self, for: ViewRenderer.self)
```

The **services** variable here represents functionality we can draw on in the rest of our app. So, when you want to connect to databases, when you want to store user data in sessions, or - like here – when you want to render Leaf templates, you register them with Vapor's service manager so you can call on them later.

The **config** variable represents settings for our app. In this case, we're telling Vapor that when

we want to render templates, it should use Leaf.

With those in place, we can go ahead and start rendering templates using Leaf – and for that we need some HTML. Like I said, templates allow us to separate our Swift code from our HTML.

Leaf comes pre-configured to look for templates in a directory called "Resources/Views". I already provided this directory inside my Vapor Clean repository, so you should use that.

Warning: Both of these directories start with capital letters. Linux uses a case-sensitive filesystem, so if you try to use a directory called "views" you will have problems.

This Views directory is going to contain all our Leaf 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.leaf, 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 Leaf template when the root route is requested. Here's how that looks right now:

```
router.get { req in
    return "Welcome to Million Hairs"
}
```

Even though our Leaf HTML is just a string of text, we can't just send it back like that. Instead, we need to use a feature called *futures*. I'm not going to lie to you: futures are

complicated, and they will take you a little while to understand. Fortunately, we have a whole book in which to explore them, so we'll take it step by step.

Futures: the least you need to know

A future is a variable that might have a value right now or it might not, but definitely will at some point. Think of it like a container that you can pass from place to place, where the contents might arrive at any point in the future.

It's perfectly normal to find futures confusing, and you might even be having flashbacks to learning Swift's optionals. It's hard, at least at first, to imagine how Vapor is able to deal with things that don't actually have a value yet, but to be honest a lot of the time you don't need to worry about it – Vapor just does the work for you.

If you're struggling, try imagining the Queen of England going on a state visit to the US. She gets on a plane, and the plane departs for Washington DC, so the British prime minister calls the US president and says, "the queen is on her way."

Preparations for the queen's arrival can now begin, even though she's still several hours away. The chief usher can prepare a red carpet, a group of cars can arrive at the airport, the UK ambassador stationed in the US can prepare a speech, and so on – all things that aren't actually used immediately, but are prepared to be used as soon as the queen's plane lands.

Finally, the plane arrives in Washington, and all the other events can start happening: she gets into the car, walks along some red carpet, and listens to various speeches – the preparations for her arrival all get resolved once her arrival completes.

This is similar to how futures work: we're saying something will be ready at some point in the future, but in the meantime we want to carry on doing other work. You can even queue up work to be done on the result of the future, and as soon as its value becomes known that work will automatically happen – we can drive a car to collect the queen and start writing a speech even though she hasn't arrived yet, because we know at some point she *will* arrive.

All this matters because all of Vapors routes return futures. We've been using simple strings so

far, but rendering a Leaf template also uses futures: a rendered HTML view will arrive at some point, but it might not be immediately.

Now, for many templates it *will* be immediate, but that doesn't matter – using futures means that all views have the possibility of returning their content later on. This means you can ask Vapor to fetch content from a database, pass that over to Leaf even while that fetch is still taking place, and Leaf will render as much as it can. When it finds a value that isn't fully resolved yet – i.e., a future that hasn't yet completed and so doesn't have its real value in place yet – Leaf can intelligently pause until that data arrives.

Best of all, futures don't stop your web server from serving other requests. So, one part of your app might be waiting for a database call to complete and another part might be waiting for Leaf to render a template, but all other parts of your app can carry on working rather than everything grinding to halt.

Rendering futures with Leaf

As I said, when we render a view using Leaf we get handed back a future – there might be a view in there or there might not be. Helpfully, a lot of the time we don't actually *care*: we can actually pass that future view back to Vapor, and it will take care of waiting for completion on our behalf.

Needless to say this is quite brilliant: the very concept of futures can be hard enough to get to grips with, never mind having to write your own code to manage them. So, being able to hand them off to Vapor to deal with makes the whole thing almost easy – *almost*.

Open routes.swift and place these two imports at the top:

```
import Foundation
import Leaf
```

Now replace the existing **router.get** code with this:

```
router.get { req -> Future<View> in
```

```
    let context = [String: String]()
    return try req.view().render("home", context)
}
```

Notice how that sends back a **Future<View>**, which is a Leaf view that will be rendered at some point but not necessarily just yet.

Even though we're passing an empty dictionary right now, the second parameter to **render()** is important – that's how we send custom data to Leaf to get merged with the static HTML, known as its *context*.

We'll come back to Leaf contexts later, but first let's put in place another easy page: "contact". Create a new template file in the Resources/Views directory called contact.leaf, 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 the **routes()** function as follows:

```
router.get("contact") { req -> Future<View> in
    let context = [String: String]()
    return try req.view().render("contact", context)
}
```

Go ahead and run the updated code, and you should be able to visit both **http://localhost:8080** and **http://localhost:8080/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.leaf and contact.leaf, 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 Leaf has a special "embed" tag that embeds one template inside another. To use it, just specify the template filename you want to pull in, excluding its ".leaf" file extension. So, add this line to both home.leaf and contact.leaf, just before the **</body>** line:

```
#embed("footer")
```

Notice the curious syntax: a hash symbol, then a Leaf function, then a parameter inside parentheses. To finish up, create a footer.leaf file inside the Views subdirectory, giving it this content:

```
<p>Copyright &copy; 2018 Million Hairs Veterinary Clinic</p>
```

© is a HTML entity that gets replaced with the copyright symbol by web browsers.

Templates are stored separately from your Swift code, and while you're developing Vapor won't cache them. This means you can change a Leaf template and press refresh in your browser to see the updates without needing to rebuild your app.

Try loading either of the routes we made – you should now see the footer text on both of them.

Matching custom routes

You've seen how we can match specific routes such as "contact", but Vapor 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.

Vapor makes this process nice and easy, because you can pass multiple components to a route and mix both fixed strings and data types. In its most basic form, you can match multiple path components like this:

```
router.get("path", "to", "staff") { req in
    return "Meet our great team"
}
```

That route will be matched when the user requests "/path/to/staff". However, what we want to do is match unknown route components, and that's where specifying data types comes in – you can use **String.parameter** to match any string for that part of the path, or even **Int.parameter** if you only want to match numbers.

So, if we used **get("staff", String.parameter)** Vapor 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 for that string by calling the **parameters.next()** method of the request, passing in the data type you want to read. Each time you call this method, the parameter you get sent back is consumed – it gets removed from the list of remaining parameters. So, this means you need to call it only once for each parameter you expect, and do so in the order listed in your route path.

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 **embed** 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 **[String: String]** – an empty dictionary. You can pass any kind of context to Leaf as long as it can be encoded using **Codable**, which means you can use arrays, dictionaries, and even custom structs if you want.

When you're starting, using something simple like **[String: String]** is a great way to learn because they are easy to understand. However, for larger projects or any times when you need heterogenous values – i.e., some strings, some arrays, some dictionaries, and so on – you need to use a *struct* for passing in your values. Structs ensure you always fill each field correctly, which means if you change the struct later on you also need to update everywhere it's used.

These structs *don't* need to be available anywhere. In fact, if you're only using a context struct in one place it makes sense to organize your code so that the structs are placed inside your routes – you're making it clear to other coders (and your future self!) that this isn't designed to work elsewhere, and you avoid cluttering your code.

Vapor is able to use any kind of struct for rendering in Leaf, with only one catch: the struct – and all its contents – must conform to the **Codable** protocol. Fortunately, this usually as easy as writing **: Codable** after its definition, as you'll see.

To demonstrate this, what we're going to do is create a **StaffView** struct, 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 back no staff data. We're using a struct here because it allows us to pass a