PRACTICAL

SERVER SIDE SWIFT

THIRD EDITION

BY TIBOR BÖDECS

SAMPLE

PRACTICAL

SERVER SIDE SWIFT

BY TIBOR BÖDECS

THIRD EDITION
VERSION 1.5.0
PUBLISHED BY TIBOR BÖDECS
3RD OF APRIL 2023

SAMPLE

CHAPTER 3

GETTING STARTED WITH SWIFTHTML

In this chapter we're going to build our first website using the SwiftHtml library. We're going to generate HTML code through Swift by creating template files using a Domain Specific Language (DSL). You'll learn about how to connect SwiftHtml with Vapor and how to render HTML by using context variables to provide additional template data. You'll learn about the syntax of SwiftHtml, how to iterate through objects, how to check optional variables, and how to extend a base template and provide a reusable framework for our website. We'll build a simple blog layout with a post list and detail pages.

RENDERING TEMPLATES USING SWIFTHTML

The SwiftHtml library has multiple library products. The core library is called **SwiftSgml**, which is an abstract interface for the other SGML-based libraries. The **SwiftHtml** product contains most of the necessary HTML tags, the **SwiftSvg** library can help you to dynamically build SVG tags, the **SwiftSitemap** product can generate a sitemap.xml file and **SwiftRss** dependency can help you with proper RSS feed generation.

You can use **SwiftHtml** to generate dynamic HTML pages for a front-end website. Using a DSL-based approach has its benefits: first of all, you don't have to write HTML code by hand, but you can use Swift and take advantage of the compiler to catch errors. Separating the template layer and using a DSL is always a good thing, and you can reuse the template files and keep away the view layer from the rest of your business logic.

Using **SwiftHtml** is relatively simple. If you're familiar with the HTML standard it's going to be very straightforward to work with this small utility library. **SwiftHtml** tries to follow the standards as much as possible, so hopefully, it's going to feel quite natural to build your templates. It also gives you type safety, so you won't be able to misspell a tag or misplace a closing tag.

Let's continue with the Vapor toolbox-based example and alter the contents of the package. We need to add the **SwiftHtml** package as a dependency to our **Package.swift** file.

/// FILE: Package.swift

```
// swift-tools-version:5.7
import PackageDescription

let package = Package(
    name: "myProject",
    platforms: [
       .macOS(.v12)
    ],
    dependencies: [
        .package(
            url: "https://github.com/vapor/vapor",
            from: "4.70.0"
```

```
        ),
        .package(
            url: "https://github.com/binarybirds/swift-html",
            from: "1.7.0"
        ),
    ],
    targets: [
        .target(name: "App", dependencies: [
            .product(name: "Vapor", package: "vapor"),
            .product(name: "SwiftHtml", package: "swift-html"),
            .product(name: "SwiftSvg", package: "swift-html"),
        ]),
        .executableTarget(name: "Run", dependencies: ["App"]),
        .testTarget(name: "AppTests", dependencies: [
            .target(name: "App"),
            .product(name: "XCTVapor", package: "vapor"),
        ])
    ]
)
```

Now you should run the **swift package update** command again, or wait until Xcode fetches the new package dependencies. When the process is completed, we should be ready to render HTML files with just a few simple lines in the **routes.swift** file:

```
/// FILE: Sources/App/routes.swift

import Vapor
import SwiftHtml

func routes(_ app: Application) throws {
    app.get { req async in
        "It works!"
    }

    app.routes.get("hello") { req -> Response in
        let doc = Document(.html) {
            Html {
                Head {
                    Title("Hello, World!")
                }
                Body {
                    H1("Hello, World!")
                }
            }
        }
        let body = DocumentRenderer(
            minify: false,
            indent: 4
        )
        .render(doc)
        return Response(
            status: .ok,
            headers: [
                "Content-Type": "text/html; charset=utf-8"
            ],
            body: .init(string: body)
        )
    }
}
```

Alter the configuration file by enabling the **FileMiddleware**, so Vapor can serve public files from a directory, please note that you might have to create this directory later on.

```
/// FILE: Sources/App/configure.swift

import Vapor

public func configure(_ app: Application) throws {
```

```
    app.middleware.use(
        FileMiddleware(
            publicDirectory: app.directory.publicDirectory
        )
    )

    try routes(app)
}
```

If you don't have a Public directory under your project folder, please create one, since we're going to place our assets there later on. This is also a good time to create other folders we'll use during this chapter. We're going to create two modules inside the App directory.

Note: *A* ***Module*** *is a common interface that can boot a collection of components required to implement a particular function of the application. For example, in a CRUDS module, there's a page to show the starting index of existing records (S), then one to create a new record \ (C), one to update an existing record (U), and one to save the updated record... well, you get the idea. All the code for a given functionality would usually be together in one module. I'm not saying that it would be in one file: on the contrary, I'm saying that the sub-directories and Swift files comprising the functionality would be under one directory. The examples here are* ***Sources/App/Modules/Blog*** *and* ***Sources/App/Modules/Web****. (See the tree structure below.)*

Use the commands:

```
cd ~/myProject
mkdir -p Public
mkdir -p Public/css
mkdir -p Public/img
mkdir -p Public/img/posts
mkdir -p Public/js
mkdir -p Resources
mkdir -p Sources/App/Controllers
mkdir -p Sources/App/Template
mkdir -p Sources/App/Models
mkdir -p Sources/App/Modules
mkdir -p Sources/App/Modules/Web
mkdir -p Sources/App/Modules/Web/Controllers
mkdir -p Sources/App/Modules/Web/Templates
mkdir -p Sources/App/Modules/Web/Templates/Html
mkdir -p Sources/App/Modules/Web/Templates/Contexts
mkdir -p Sources/App/Modules/Blog
mkdir -p Sources/App/Modules/Blog/Controllers
mkdir -p Sources/App/Modules/Blog/Templates
mkdir -p Sources/App/Modules/Blog/Templates/Contexts
mkdir -p Sources/App/Modules/Blog/Templates/Html
mkdir -p Sources/App/Middlewares
```

Your directory structure should look like this now:

```
.
├── Dockerfile
├── Package.resolved
├── Package.swift
├── Public
| ├── css
| ├── img
| | └── posts
```

```
| └── js
├── Resources
├── Sources
| ├── App
| | ├── Controllers
| | ├── Middlewares
| | ├── Migrations
| | ├── Models
| | ├── Modules
| | | ├── Blog
| | | | ├── Controllers
| | | | └── Templates
| | | |     ├── Contexts
| | | |     └── Html
| | | └── Web
| | |     ├── Controllers
| | |     └── Templates
| | |         ├── Contexts
| | |         └── Html
| | ├── Template
| | ├── configure.swift
| | └── routes.swift
| └── Run
|     └── main.swift
├── Tests
| └── AppTests
|     └── AppTests.swift
└── docker-compose.yml
```

A **middleware** is a function that will be executed every time before the request handler. So in our case, if the browser asks for a file such as a stylesheet, a script, or an image, the **FileMiddleware** can look it up in the **public directory**. If the file exists, the content will be returned as a response. This is great for serving static assets, but please note that everything inside the configured directory will be publicly available through the server, so don't place sensitive data there.

In the next part of this example, we're simply using a request handler block and the built-in **DocumentRenderer** from SwiftHtml to return a **Response** with the necessary headers. A response object is something that represents an HTTP response. It has a status code, some header information, and maybe a body. In our case, we simply set the proper content-type header for our HTML string output, and we can use a 200 status code to indicate that the response was OK. The **DocumentRenderer** simply turns our HTML DSL structure into plain text; you can also minify your output, or set the indent size if you want.

Although Vapor has an abstract view layer that we could use to render our template files, we want to have a bit more type safety, so we're going to create our own template renderer. First of all, we're going to need a reusable template protocol:

```
/// FILE: Sources/App/Template/TemplateRepresentable.swift

import Vapor
import SwiftSgml

public protocol TemplateRepresentable {
```

```
    @TagBuilder
    func render(_ req: Request) -> Tag
}
```

This interface has only one method that can return a **Tag** object; the method itself is called **render** and it'll receive the current **Request** object so we'll be able to access it inside our template files. The next step is to create the actual renderer, which is going to be very similar to the method that we've already had in our configuration file.

```
/// FILE: Sources/App/Template/TemplateRenderer.swift

import Vapor
import SwiftHtml

public struct TemplateRenderer {

    var req: Request

    init(_ req: Request) {
        self.req = req
    }

    public func renderHtml(
        _ template: TemplateRepresentable,
        minify: Bool = false,
        indent: Int = 4
    ) -> Response {
        let doc = Document(.html) {
            template.render(req)
        }
        let body = DocumentRenderer(
            minify: minify,
            indent: indent
        )
        .render(doc)
        return Response(
            status: .ok,
            headers: [
                "Content-Type": "text/html; charset=utf-8"
            ],
            body: .init(string: body)
        )
    }
}
```

The **TemplateRenderer** can render an HTML template, which is a **TemplateRepresentable** object, and we're also going to be able to set additional minification and indentation options when calling the **renderHtml** method. This method returns with a **Response** object and it's using the same principles as we've seen before. The TemplateRenderer has an internal init method: we won't create this struct here, but instead, we're going to extend the Request object to get an instance of the renderer.

```
/// FILE: Sources/App/Template/Request+Template.swift

import Vapor

public extension Request {
    var templates: TemplateRenderer { .init(self) }
}
```

Now if we go back to our router file, we can create a new template and render it using the **req.templates** extension.

```
/// FILE: Sources/App/routes.swift
```

```swift
import Vapor
import SwiftHtml

struct MyTemplate: TemplateRepresentable {
    let title: String

    func render(_ req: Request) -> Tag {
        Html {
            Head {
                Title(title)
            }
            Body {
                H1(title)
            }
        }
    }
}

func routes(_ app: Application) throws {
    app.get { req async in
        "It works!"
    }

    app.routes.get("hello") { req -> Response in
        req.templates.renderHtml(
            MyTemplate(title: "Hello, World!")
        )
    }
}
```

As you can see, the **MyTemplate** struct conforms to the **TemplateRepresentable** protocol; we've also introduced a new contextual variable called **title**. We can pass this context to our template when we initialize it, and we can access the title in the render method. The request object is also available in the render method, but at this time, we won't use it.

Finally we can call the **req.templates.renderHtml** method using our template instance to return an HTML response.

TEMPLATES AND CONTEXTS

So far, we're good with the template renderer, so now let's tackle one other issue by creating a reusable index template that's going to be the base of every web page that we're going to render later on. Since we're going to use a modular approach to build everything, this is why we created the **Modules** folder with a **Web** module inside of it.

We're going to place all of our templates inside the **Templates/Html** directory; every template will have an associated context object, and we're going to store those inside a **Templates/Contexts** directory.

```swift
/// FILE: Sources/App/Modules/Web/Templates/Html/WebIndexTemplate.swift

import Vapor
import SwiftHtml

public struct WebIndexTemplate: TemplateRepresentable {

    public var context: WebIndexContext
```

```swift
    public init(_ context: WebIndexContext) {
        self.context = context
    }

    @TagBuilder
    public func render(_ req: Request) -> Tag {
        Html {
            Head {
                Meta()
                    .charset("utf-8")
                Meta()
                    .name(.viewport)
                    .content("width=device-width, initial-scale=1")

                Link(rel: .shortcutIcon)
                    .href("/img/favicon.ico")
                    .type("image/x-icon")
                Link(rel: .stylesheet)
                    .href("https://cdn.jsdelivr.net/gh/feathercms/feather-core@1.0.0-
beta.44/feather.min.css")
                Link(rel: .stylesheet)
                    .href("/css/web.css")

                Title(context.title)
            }
            Body {
                Main {
                    Section {
                        H1(context.message)
                    }
                    .class("wrapper")
                }
            }
        }
        .lang("en-US")
    }
}
```

This file is our index HTML template. If you're familiar with SwiftUI, you should notice that the render method uses a result builder to create the necessary structure. The syntax itself is very simple: every single HTML tag is available as a Tag subclass, so the naming convention is the same. You can add attributes through modifiers and the entire tree will be rendered using the DocumentRenderer that we've introduced a bit earlier.

Before we move forward, we should talk a bit about CSS. If you don't know much about HTML & CSS, you should take a look at this HTML Tutorial. This book will focus more on Swift, but since the templates will contain lots of HTML code you should have some basic understanding of the fundamentals of frontend development, including both the Hypertext Markup Language and Cascading Style Sheets.

We're going to use an external stylesheet through a Content Delivery Network (CDN) system. A CDN allows us to load external resources much faster. The external Feather CSS file is part of a CMS system, that contains some basic components that we can use to style our document. If you take a look at our website with this extra stylesheet imported, you should see that it works nice both in light and dark mode. If you want to know more about the underlying components, please take a look at the docs.

After the external CSS, we're also going to add one more extra line to a local CSS file reference. We're going to place our local style overrides into the **Public/css/web.css** file. In the **Public/css** folder, create a **web.css** file. A CSS is a stylesheet that describes the visual style of an HTML document. You can learn more about this format using the W3Schools

website. Our **web.css** file will be quite empty for now, since the external stylesheet gives us pretty much everything we need to display a nice-looking, but still really basic website.

```
touch Public/css/web.css
```

We're also going to define the context object to use its properties as variables inside our template file. The context that's used for this template is called **WebIndexContext** and it's a relatively simple struct.

```
/// FILE: Sources/App/Modules/Web/Templates/Contexts/WebIndexContext.swift

public struct WebIndexContext {

    public let title: String
    public let message: String

    public init(
        title: String,
        message: String
    ) {
        self.title = title
        self.message = message
    }
}
```

The final step is to alter our routes a little bit and return the rendered template as an HTML response. You can render a template by using the **req.templates.renderHtml** method; we just have to initialize our template with a given context.

```
// FILE: Sources/App/routes.swift

import Vapor
import SwiftHtml

func routes(_ app: Application) throws {

    app.routes.get { req -> Response in
        req.templates.renderHtml(
            WebIndexTemplate(
                WebIndexContext(
                    title: "Home",
                    message: "Hi there, welcome to my page!"
                )
            )
        )
    }
}
```

We can say that a template context is a **type-safe data** representation of everything that we need to render our template file. Of course, the request object is also available inside the render method and we can use it to get dynamic path components or the currently logged-in user, but let's talk about these kinds of things later on.

Remember, the render method will convert your template file into an HTML string and set some extra headers. The **Content-Type** will be set to **text/html**, so your browser can render the page as an HTML website. Run the app using the command line or Xcode; but if you're using Xcode, definitely **don't forget to set a custom working directory** or the server won't find your templates and public files. Check the previous chapter if you don't know how to set up a custom working directory.

If your project isn't on your local machine, test the app using **curl**. The ampersand at the end of the Swift command will run the compile in the background so that you can use **curl** in the foreground.

```
swift run &
curl localhost:8080
```

The response you expect is:

```
<!DOCTYPE html>
<html lang="en-US">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="shortcut icon" href="/img/favicon.ico" type="image/x-icon">
        <link rel="stylesheet" href="https://cdn.jsdelivr.net/gh/feathercms/feather-core@1.0.0-beta.44/feather.min.css">
        <link rel="stylesheet" href="/css/web.css">
        <title>Home</title>
    </head>
    <body>
        <main>
            <section class="wrapper">
                <h1>Hi there, welcome to my page!</h1>
            </section>
        </main>
    </body>
</html>
```

To stop the project, bring the job to the foreground, the use **CTRL+C**.

```
$ fg %1
# press CTRL+C to stop the process
```

TEMPLATE HIERARCHY

Splitting up templates is going to be essential if you're planning to build a multi-page website. We can create reusable parts that you can **share** and render them later on inside other template files. In the following example, we're going to create three separate pages. First, we have to update the index template, since that's going to be reused for the entire website.

```
/// FILE: Sources/App/Modules/Web/Templates/Html/WebIndexTemplate.swift

import Vapor
import SwiftHtml
import SwiftSvg

extension Svg {

    static func menuIcon() -> Svg {
        Svg {
            Line(x1: 3, y1: 12, x2: 21, y2: 12)
            Line(x1: 3, y1: 6, x2: 21, y2: 6)
            Line(x1: 3, y1: 18, x2: 21, y2: 18)
        }
        .width(24)
        .height(24)
```

```
            .viewBox(minX: 0, minY: 0, width: 24, height: 24)
            .fill("none")
            .stroke("currentColor")
            .strokeWidth(2)
            .strokeLinecap("round")
            .strokeLinejoin("round")
        }
    }

public struct WebIndexTemplate: TemplateRepresentable {

    public var context: WebIndexContext
    var body: Tag

    public init(
        _ context: WebIndexContext,
        @TagBuilder _ builder: () -> Tag
    ) {
        self.context = context
        self.body = builder()
    }

    @TagBuilder
    public func render(
        _ req: Request
    ) -> Tag {
        Html {
            Head {
                Meta()
                    .charset("utf-8")
                Meta()
                    .name(.viewport)
                    .content("width=device-width, initial-scale=1")

                Link(rel: .shortcutIcon)
                    .href("/img/favicon.ico")
                    .type("image/x-icon")
                Link(rel: .stylesheet)
                    .href("https://cdn.jsdelivr.net/gh/feathercms/feather-core@1.0.0-
beta.44/feather.min.css")
                Link(rel: .stylesheet)
                    .href("/css/web.css")

                Title(context.title)
            }
            Body {
                Header {
                    Div {
                        A {
                            Img(src: "/img/logo.png", alt: "Logo")
                        }
                        .id("site-logo")
                        .href("/")

                        Nav {
                            Input()
                                .type(.checkbox)
                                .id("primary-menu-button")
                                .name("menu-button")
                                .class("menu-button")
                            Label {
                                Svg.menuIcon()
                            }
                            .for("primary-menu-button")
                            Div {
                                A("Home")
                                    .href("/")
                                    .class("selected", req.url.path == "/")
                                A("Blog")
                                    .href("/blog/")
                                    .class("selected", req.url.path == "/blog/")
                                A("About")
                                    .href("#")
                                    .onClick("javascript:about();")
```

```
                            }
                            .class("menu-items")
                        }
                        .id("primary-menu")
                    }
                    .id("navigation")
                }

                Main {
                    body
                }

                Footer {
                    Section {
                        P {
                            Text("This site is powered by ")
                            A("Swift")
                                .href("https://swift.org")
                                .target(.blank)
                            Text(" & ")
                            A("Vapor")
                                .href("https://vapor.codes")
                                .target(.blank)
                            Text(".")
                        }
                        P("myPage &copy; 2020-2022")
                    }
                }

                Script()
                    .type(.javascript)
                    .src("/js/web.js")

            }
        }
        .lang("en-US")
    }
}
```

One major change: here's the new builder parameter that you can pass for the template file. It's marked with the **@TagBuilder** result builder, so this means that you can build an additional HTML structure when calling the init method, and the final tag of that result will be used inside the main section of the index template. It's not that complicated when you see it in use; you can simply create a new custom body tag for your index template through this builder attribute.

The template structure itself is similar to our previous version, but we've added a new **header** section with a **logo**, plus some navigation links that'll help us to transition between the sub-pages. We're using the **SwiftSvg** library from the SwiftHtml package to render an inline SVG to represent our navigation menu icon. It's a standard hamburger menu element.

The good news is that you can create even smaller chunks as functions, so for example the entire navigation can be a standalone template file or just a new method inside the index template. Just play around with this a bit and try to make it fit your needs.

In our case, this index template will be reused across multiple pages, so we don't have to copy and paste all the generic Swift HTML code that would be the same everywhere. We're going to fill the body placeholder with some actual tag defined in other templates, plus replace the title variable using the context. Please make sure that you remove the **message** variable from the **WebIndexContext** struct since we don't need that anymore.

```swift
/// FILE: Sources/App/Modules/Web/Templates/Contexts/WebIndexContext.swift

public struct WebIndexContext {

    public let title: String

    public init(
        title: String
    ) {
        self.title = title
    }
}
```

We have to move the message in the **routes.swift** file into the tag builder.

```swift
/// FILE: Sources/App/routes.swift


import Vapor
import SwiftHtml

func routes(_ app: Application) throws {

    app.routes.get { req -> Response in
        req.templates.renderHtml(
            WebIndexTemplate(
                WebIndexContext(
                    title: "Home"
                )
            ) {
                P("Hi there, welcome to my page!")
            }
        )
    }
}
```

We're going to download the site logo and the favicon from the GitHub repository using the following snippet:

```
SRC="raw.githubusercontent.com/tib/practical-server-side-swift/main/Assets"
DST="$HOME/myProject/Public/img/"
curl https://$SRC/favicon.ico -o $DST/favicon.ico
curl https://$SRC/logo.png -o $DST/logo.png
```

As the last component of our index template, we're going to embed some basic javascript files from the **Public/js** directory. Please create an empty **web.js** file; no worries, we'll use this real soon.

```
touch Public/js/web.js
```

Now you should try to run the application.

THE HOME PAGE TEMPLATE

The home page will be really simple, but first, we're going to create a new **WebHomeContext** struct to represent the data that we'd like to render later on inside our home template.

```swift
/// FILE: Sources/App/Modules/Web/Templates/Contexts/WebHomeContext.swift

struct WebHomeContext {
    let title: String
    let message: String
}
```

Now we can define our **WebHomeTemplate** file. The tricky part is that we're going to render a **WebIndexTemplate** with a custom body tag and we're going to feed the index template's context with the title from the home template context.

```swift
/// FILE: Sources/App/Modules/Web/Templates/Html/WebHomeTemplate.swift

import Vapor
import SwiftHtml

struct WebHomeTemplate: TemplateRepresentable {

    var context: WebHomeContext

    init(
        _ context: WebHomeContext
    ) {
        self.context = context
    }

    @TagBuilder
    func render(
        _ req: Request
    ) -> Tag {
        WebIndexTemplate(
            .init(
                title: context.title
            )
        ) {
            Div {
                Section {
                    H1(context.title)
                    P(context.message)
                }
                .class("lead")
            }
            .id("home")
            .class("container")
        }
        .render(req)
    }
}
```

It's time to render the entire page. We don't have to set a **body** parameter anymore using the context variable in the request handler since it's already defined in the home template. This is a major difference between variables and evaluated blocks. We can say in general that variables are usually coming from Swift, and blocks will be defined using templates.

It's possible to create a chain of templates, so for example index ▸ page ▸ welcome. Multi-level templates are fine, if you follow the same pattern from above you can create a nice hierarchy for your views, but you can also go the other way around. So for example, you can create a LeadTemplate with a title and message context, and render that template inside of a **<div>** instead of manually placing the same code there again and again. Try to experiment with this now, but later on, I'll show you examples.

MODULE CONTROLLERS

What makes a **module**? I already mentioned that a Vapor app can have models, controllers, migration scripts, and many more. A module is something that holds together these components plus our template and context files. Our very first module is called **Web** because it's responsible for rendering the main pages of our website.

Until now, we've placed everything inside the configure or routes files, but that's not a very good approach to separate things. We'll move the entire template render logic from these files by using a dedicated **WebFrontendController** object. You can put this controller into a file with the same name, under a **Controllers** directory inside the **Web** module. Usually, most of the structs and classes have their own dedicated Swift files, you should follow this convention later on too.

Instead of using request handler completion blocks, you can also create a function that has the same signature as the block had, and we can connect this method to the route as a pointer to handle incoming requests. First, this is our new controller.

```
/// FILE: Sources/App/Modules/Web/Controllers/WebFrontendController.swift

import Vapor

struct WebFrontendController {

    func homeView(
        req: Request
    ) throws -> Response {
        req.templates.renderHtml(
            WebHomeTemplate(
                .init(
                    title: "Home",
                    message: "Hi there, welcome to my page."
                )
            )
        )
    }
}
```

The next thing that we should do is to make the connection between the router and the controller. We're not going to simply put everything into the routes or the config file; instead, we'll have a standalone **Router**. If you have lots of routes it's a good idea to split them up into collections by using the **RouteCollection** protocol. This protocol has a **boot** function that you have to implement and register the routes using the routes object instead of the app.

You can use the same get method on the routes object just like we did before. There are helper functions defined on the **RoutesBuilder** that are available for all the HTTP methods (get, post, put, delete, etc.). You can also group routes by path components or middleware. A route group can be used to connect endpoints under the same namespace with similar functions.

You could also enter a specific path component as the first parameter, but in our case, we'll simply connect our homeView method from the WebFrontendController to the main endpoint.

```
/// FILE: Sources/App/Modules/Web/WebRouter.swift
```

SAMPLE

```swift
import Vapor

struct WebRouter: RouteCollection {

    let frontendController = WebFrontendController()

    func boot(
        routes: RoutesBuilder
    ) throws {
        routes.get(use: frontendController.homeView)
    }
}
```

Now we have to boot the router inside the configuration method. This is a nice approach since you can have multiple routers and register as many as you want. The boot method needs a route builder, so we can pass the **app.routes** property, and that'll just work fine.

```swift
/// FILE: Sources/App/configure.swift

import Vapor

public func configure(
    _ app: Application
) throws {

    app.middleware.use(
        FileMiddleware(
            publicDirectory: app.directory.publicDirectory
        )
    )

    let router = WebRouter()
    try router.boot(routes: app.routes)
}
```

You don't need the **routes.swift** file anymore because it was replaced by **WebRouter.swift**. Delete it like this:

```
rm Sources/App/routes.swift
```

Run the application and you should see a nice little home page rendered by using the two template files combined. Don't go to the blog page yet: we're going to do that one next.

RENDERING SUB-TEMPLATES

I mentioned that you can render a template inside a template, so let me show you an example of how to do that. We're going to use quite a lot of links later on, so it makes sense to create a **WebLinkContext** object with a **label** and **URL** property.

```swift
/// FILE: Sources/App/Modules/Web/Templates/Contexts/WebLinkContext.swift

public struct WebLinkContext {

    public let label: String
    public let url: String

    public init(
        label: String,
```

SAMPLE

```
        url: String
    ) {
        self.label = label
        self.url = url
    }
}
```

With a corresponding **WebLinkTemplate**, we can render our WebLinkContext objects; of course, we could add more properties, such as style classes or a boolean value to determine if the link is a blank link or not, but for the sake of simplicity let's just start with a label & URL. It's a good experiment for you if you'd like to add more options.

```
/// FILE: Sources/App/Modules/Web/Templates/Html/WebLinkTemplate.swift

import Vapor
import SwiftHtml

struct WebLinkTemplate: TemplateRepresentable {

    var context: WebLinkContext

    init(
        _ context: WebLinkContext
    ) {
        self.context = context
    }

    @TagBuilder
    func render(
        _ req: Request
    ) -> Tag {
        A(context.label)
            .href(context.url)
    }
}
```

We should also alter the **WebHomeContext** struct, so we can take advantage of the newly created link context. We're also going to drop in a new icon property to make our home page just a bit prettier.

```
/// FILE: Sources/App/Modules/Web/Templates/Contexts/WebHomeContext.swift

struct WebHomeContext {
    let icon: String
    let title: String
    let message: String
    let paragraphs: [String]
    let link: WebLinkContext
}
```

We have to upgrade our home page template to represent the changes that we made earlier.

```
/// FILE: Sources/App/Modules/Web/Templates/Html/WebHomeTemplate.swift

import Vapor
import SwiftHtml

struct WebHomeTemplate: TemplateRepresentable {

    var context: WebHomeContext

    init(
```

SAMPLE

```swift
        _ context: WebHomeContext
    ) {
        self.context = context
    }

    @TagBuilder
    func render(
        _ req: Request
    ) -> Tag {
        WebIndexTemplate(
            .init(title: context.title)
        ) {
            Div {
                Section {
                    P(context.icon)
                    H1(context.title)
                    P(context.message)
                }
                .class("lead")

                for paragraph in context.paragraphs {
                    P(paragraph)
                }

                WebLinkTemplate(context.link).render(req)
            }
            .id("home")
            .class("container")
        }
        .render(req)
    }
}
```

As you can see, we can use the **WebLinkTemplate** with the link context (that's part of the home context) and use the render method on the template to return a tag. The returned **Tag** object is just like any other tag that we can create by hand, so it's safe to embed one template inside of another.

Please note that we can still use a regular **for** loop (also it's possible to use **if-else**) inside the template file. This is great because we can iterate through paragraph values and render them by using the **P** tag.

```swift
/// FILE: Sources/App/Modules/Frontend/Controllers/FrontendController.swift

import Vapor

struct WebFrontendController {

    func homeView(req: Request) throws -> Response {
        let ctx = WebHomeContext(
            icon: "👋",
            title: "Home",
            message: "Hi there, welcome to my page.",
            paragraphs: [
                "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
                "Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.",
                "Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris.",
                "Nisi ut aliquip ex ea commodo consequat.",
            ],
            link: .init(
                label: "Read my blog →",
                url: "/blog/"
            )
        )

        return req.templates.renderHtml(
            WebHomeTemplate(ctx)
```

SAMPLE

```
        )
    }
}
```

Finally, we have to modify the frontend controller, and of course, we can use some lorem ipsum text to display some random text inside the body. As you can see, using template hierarchies is quite simple with SwiftHtml, since you can use a **@TagBuilder** to provide additional content for a template, or you can simply **render** a template inside another.

THE BLOG LIST

Since we're building an app using a modular architecture, we can't simply put blog-related stuff into the Web module. The web module is somewhat special in our case since it provides us with the main elements to render our website. It contains the index template the web stylesheet and javascript files too.

That's why we created the module called **Blog**. Every single module will follow the same pattern as we created before. This means that we're going to have dedicated routers and controllers. Before we dive in we're going to create a **BlogPost** struct to represent our articles. Make a new Swift file under the **Sources/App/Modules/Blog** directory.

```
/// FILE: Sources/App/Modules/Blog/BlogPost.swift

import Foundation

struct BlogPost: Codable {
    let title: String
    let slug: String
    let image: String
    let excerpt: String
    let date: Date
    let category: String?
    let content: String
}
```

Title is the title of the blog post. We're going to use the **slug** field to have a nice SEO-friendly URL for the posts. I've prepared some images that you can grab from the source materials. Place them under the **Public/img/posts** directory. The easiest way is to enter the commands below into your AWS terminal. You can use the same commands on a Mac in a terminal window.

```
SRC="raw.githubusercontent.com/tib/practical-server-side-swift/main/Assets"
DST="$HOME/myProject/Public/img/posts"
curl https://$SRC/01.jpg -o $DST/01.jpg
curl https://$SRC/02.jpg -o $DST/02.jpg
curl https://$SRC/03.jpg -o $DST/03.jpg
curl https://$SRC/04.jpg -o $DST/04.jpg
curl https://$SRC/05.jpg -o $DST/05.jpg
curl https://$SRC/06.jpg -o $DST/06.jpg
curl https://$SRC/07.jpg -o $DST/07.jpg
curl https://$SRC/08.jpg -o $DST/08.jpg
curl https://$SRC/09.jpg -o $DST/09.jpg
curl https://$SRC/10.jpg -o $DST/10.jpg
```

We're going to store the name of these under the **image** field. **Excerpt** is going to be displayed on the list, and post **date** is the publish date of a given post. **Category** is an optional string that we're going to use as a category to group posts together. **Content** is going to be displayed on the post detail pages.

How do we store these blog posts? Well, for now, we're going to generate some random data using the **BlogFrontendController** to simplify things. In the next chapter, we're going to use an SQLite database, and later on, we're going to migrate to PostgreSQL storage.

We're going to create a few sample posts by simply using the **stride** method and map the indexes to **BlogPost** types. To uniquely identify every blog post with a **slug**, we just use a standard dashed version of the title which will also contain the index value. We'll generate random **date** values from the past 60 days for the sample posts. There will be a total of 9 random posts. Finally, everything gets sorted by date, all of this happens inside of a **posts** variable.

```
/// FILE: Sources/App/Modules/Blog/Controllers/BlogFrontendController.swift

import Vapor

struct BlogFrontendController {

    var posts: [BlogPost] = {
        stride(from: 1, to: 9, by: 1).map { index in
            BlogPost(
                title: "Sample post #\(index)",
                slug: "sample-post-\(index)",
                image: "/img/posts/\(String(format: "%02d", index + 1)).jpg",
                excerpt: "Lorem ipsum",
                date: Date().addingTimeInterval(-Double.random(in: 0...(86400 * 60))),
                category: Bool.random() ? "Sample category" : nil,
                content: "Lorem ipsum dolor sit amet."
            )
        }.sorted() { $0.date > $1.date }
    }()
}
```

The **BlogFrontendController** is responsible for handling all the blog-related routes that are being publicly available on the web. That's why it's called a frontend controller. We'll use the same logic later on to create other types of content channels, such as admin controllers and API controllers.

Now for our blog posts page, we're going to need a new **BlogPostsContext** struct that we can use to render a page.

```
/// FILE: Sources/App/Modules/Blog/Templates/Contexts/BlogPostsContext.swift

struct BlogPostsContext {
    let icon: String
    let title: String
    let message: String
    let posts: [BlogPost]
}
```

We should add a new template called **BlogPostsTemplate** to the project. This file goes under the **Blog/Templates/Html** directory. we're going to iterate through the blog posts in this view and render the available post data.

```swift
/// FILE: Sources/App/Modules/Blog/Templates/Html/BlogPostsTemplate.swift

import Vapor
import SwiftHtml

struct BlogPostsTemplate: TemplateRepresentable {

    var context: BlogPostsContext

    init(
        _ context: BlogPostsContext
    ) {
        self.context = context
    }

    @TagBuilder
    func render(
        _ req: Request
    ) -> Tag {
        WebIndexTemplate(
            .init(title: context.title)
        ) {
            Div {
                Section {
                    P(context.icon)
                    H1(context.title)
                    P(context.message)
                }
                .class("lead")

                Div {
                    for post in context.posts {
                        Article {
                            A {
                                Img(src: post.image, alt: post.title)
                                H2(post.title)
                                P(post.excerpt)
                            }
                            .href("/\(post.slug)/")
                        }
                    }
                }
                .class("grid-221")
            }
            .id("blog")
        }
        .render(req)
    }
}
```

I already mentioned this, but the nice thing about using the third-party Feather CSS framework is that we get most of the components out of the box. For example, our list will be responsive, because we're using the **grid-221** class.

This means that the grid will use a 2 column layout on desktop and tablet devices and it'll feature a single column on mobile devices. We have to tweak the standard heading elements for our posts when we display them in the list; we're going to add one small change to our **web.css** file.

```css
/* FILE: Public/css/web.css */
```

```
#blog h2 {
    margin: 0.5rem 0;
}
```

Now we can set up a request handler for this template inside the controller. Don't remove anything; just add the **func blogView** at the end.

```swift
/// FILE: Sources/App/Modules/Blog/Controllers/BlogFrontendController.swift

import Vapor

struct BlogFrontendController {

    var posts: [BlogPost] = {
        stride(from: 1, to: 9, by: 1).map { index in
            BlogPost(
                title: "Sample post #\(index)",
                slug: "sample-post-\(index)",
                image: "/img/posts/\(String(format: "%02d", index + 1)).jpg",
                excerpt: "Lorem ipsum",
                date: Date().addingTimeInterval(-Double.random(in: 0...(86400 * 60))),
                category: Bool.random() ? "Sample category" : nil,
                content: "Lorem ipsum dolor sit amet."
            )
        }.sorted() { $0.date > $1.date }
    }()

    func blogView(
        req: Request
    ) throws -> Response {
        let ctx = BlogPostsContext(
            icon: "🔥",
            title: "Blog",
            message: "Hot news and stories about everything.",
            posts: posts
        )
        return req.templates.renderHtml(
            BlogPostsTemplate(ctx)
        )
    }
}
```

The request handler is very similar to the one that we made for the **home** template, except that now we use an **array of posts** as part of the context. We'll also have to create a router object for the blog module along with the controller. The only route that we're going to register is going to be the list view for the blog. This goes inside the blog module directory saved as **BlogRouter.swift**.

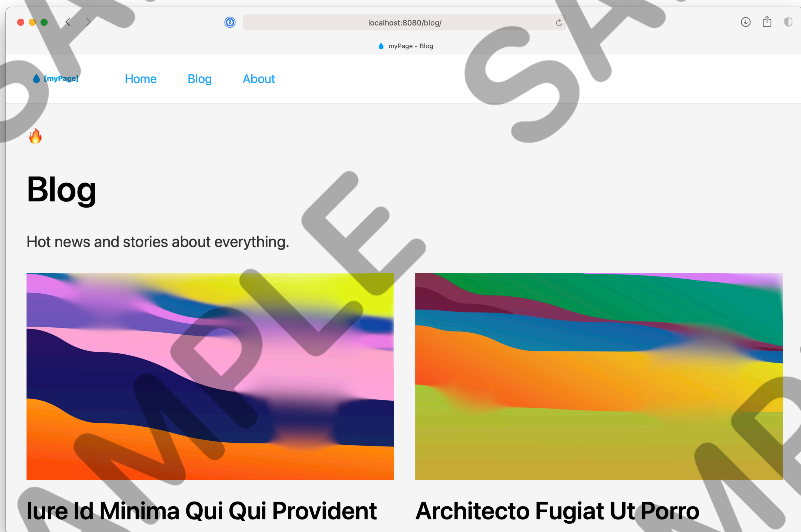
```swift
/// FILE: Sources/App/Modules/Blog/BlogRouter.swift

import Vapor

struct BlogRouter: RouteCollection {

    let controller = BlogFrontendController()

    func boot(
        routes: RoutesBuilder
    ) throws {
        routes.get("blog", use: controller.blogView)
    }
}
```

SAMPLE

As a final step, we have to register this newly created **BlogRouter** inside the config file. We can simply initiate a new object and put it into the **routers** array. This way Vapor can boot both the frontend and the blog router and register all the necessary route handlers.

```swift
/// FILE: Sources/App/configure.swift

import Vapor

public func configure(
    _ app: Application
) throws {

    app.middleware.use(
        FileMiddleware(
            publicDirectory: app.directory.publicDirectory
        )
    )

    let routers: [RouteCollection] = [
        WebRouter(),
        BlogRouter(),
    ]
    for router in routers {
        try router.boot(routes: app.routes)
    }
}
```

Run the application and navigate to the **/blog/** page, you should see the list of posts.



THE POST ENTRY PAGES

SAMPLE

The very last thing in this chapter that we're going to accomplish is that we implement a search engine optimization (SEO) friendly routing for the blog post detail pages. This means that we're going to use a unique slug as the path of the URL to see the detail page for every single article. We'll start by creating a new **BlogPostTemplate** file in the Templates folder.

```
/// FILE: Sources/App/Modules/Blog/Templates/Html/BlogPostTemplate.swift

import Vapor
import SwiftHtml

struct BlogPostTemplate: TemplateRepresentable {

    var context: BlogPostContext

    init(
        _ context: BlogPostContext
    ) {
        self.context = context
    }

    var dateFormatter: DateFormatter = {
        let formatter = DateFormatter()
        formatter.dateStyle = .long
        formatter.timeStyle = .short
        return formatter
    }()

    @TagBuilder
    func render(
        _ req: Request
    ) -> Tag {
        WebIndexTemplate(
            .init(title: context.post.title)
        ) {
            Div {
                Section {
                    P(dateFormatter.string(from: context.post.date))
                    H1(context.post.title)
                    P(context.post.excerpt)
                }
                .class(["lead", "container"])

                Img(src: context.post.image, alt: context.post.title)

                Article {
                    Text(context.post.content)
                }
                .class("container")
            }
            .id("post")
        }
        .render(req)
    }
}
```

The date is a special variable: since it's stored as a **Date** value, we can format it and print it as a human-friendly representation with the help of a custom date formatter. The good news is that template files are Swift files, so it's really easy to share a global date formatter to use the same format, but this time a local variable will do just fine.

Apart from the date output, the snippet above follows pretty much the same logic as we had in the blog template. The context that we used for it (**BlogPostContext**) contains a simple post variable.

```
/// FILE: Sources/App/Modules/Blog/Templates/Contexts/BlogPostContext.swift
```

```swift
struct BlogPostContext {
    let post: BlogPost
}
```

In our controller, we have to find the first element that has a matching slug with the current path of our URL string. If there's no match, we can simply redirect the browser to the home screen, but if there's an article that has the given path, we can render it using the view system. Add **postView** to the end of **BlogFrontendController**.

```swift
/// FILE: Sources/App/Modules/Blog/Controllers/BlogFrontendController.swift

import Vapor

struct BlogFrontendController {

    var posts: [BlogPost] = {
        stride(from: 1, to: 9, by: 1).map { index in
            BlogPost(
                title: "Sample post #\(index)",
                slug: "sample-post-\(index)",
                image: "/img/posts/\(String(format: "%02d", index + 1)).jpg",
                excerpt: "Lorem ipsum",
                date: Date().addingTimeInterval(-Double.random(in: 0...(86400 * 60))),
                category: Bool.random() ? "Sample category" : nil,
                content: "Lorem ipsum dolor sit amet."
            )
        }.sorted() { $0.date > $1.date }
    }()

    func blogView(
        req: Request
    ) throws -> Response {
        let ctx = BlogPostsContext(
            icon: "🔥",
            title: "Blog",
            message: "Hot news and stories about everything.",
            posts: posts
        )
        return req.templates.renderHtml(
            BlogPostsTemplate(ctx)
        )
    }

    func postView(
        req: Request
    ) throws -> Response {
        let slug = req.url.path.trimmingCharacters(
            in: .init(charactersIn: "/")
        )
        guard let post = posts.first(where: { $0.slug == slug }) else {
            return req.redirect(to: "/")
        }
        let ctx = BlogPostContext(post: post)
        return req.templates.renderHtml(
            BlogPostTemplate(ctx)
        )
    }
}
```

You can access the path of the URL via the **req.url.path** property. We need to trim it first since we don't care about trailing and leading slashes; next, we can filter our blog posts to see if there are any that match the given route.

SAMPLE

This time we'll **redirect** to the home page if there was no match using a future response. Otherwise, we'll display the post using the view renderer. Since the **redirect** method also returns a **Response**, it's safe to return with it.

ROUTES AND PATH COMPONENTS

So, we were able to create our controller method; now the only question is: how do we connect the handler to every single route that can have a possible matching path?

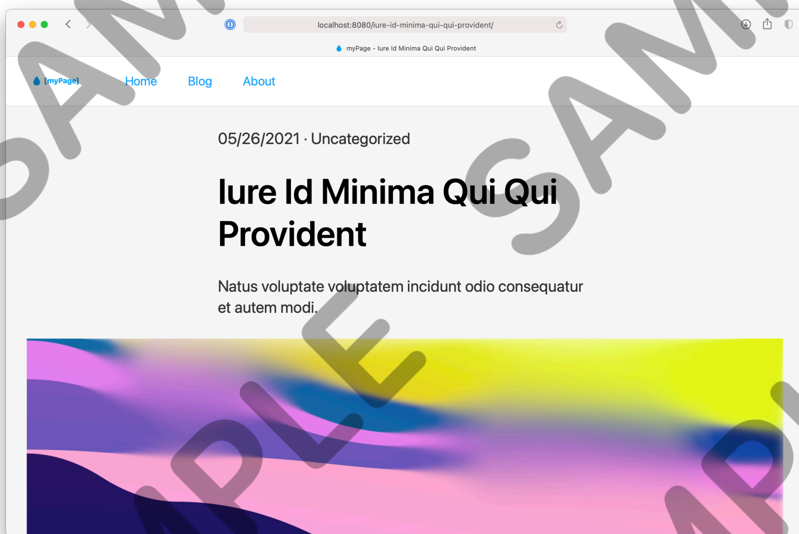
You need to know that you can catch all the routes using a route handler and the **.anything** path component. There's also a **.catchall** case, the only difference between the two of them is that anything (*) is just a single match for a path component, but the catch-all (**) case will catch everything after the first **/** character including other sub-paths such as **/foo/bar/**.

In our case the **.anything** pattern will be enough, this is how we can use it:

```
/// FILE: Sources/App/Modules/Blog/BlogRouter.swift

import Vapor

struct BlogRouter: RouteCollection {

    let controller = BlogFrontendController()

    func boot(
        routes: RoutesBuilder
    ) throws {
        routes.get("blog", use: controller.blogView)
        routes.get(.anything, use: controller.postView)
    }
}
```

Build and run the application using the command line or Xcode. In your browser window click on one of the blog posts and hopefully you should be able to read the full article.

SAMPLE



From an SEO perspective, this approach is nice because of the clean URLs. That's one of the most important factors during ranking. As a practice you can extend the index template with some additional meta information; to support rich previews or, as an alternative, you can move out the lead section and build a custom template for it.

CUSTOM MIDDLEWARES

Now if you enter the blog URL, notice that it'll work with a / suffix and without a trailing slash character. This means that we can access every single URL using two versions of the same path (e.g. **/blog/** vs **/blog**). We can change this behavior, if needed, by hooking into the "responder chain".

As I mentioned before, middlewares can hook into requests and alter their behavior. we're going to place our custom middlewares into a **Middlewares** folder under the App/Middleware folder we already created, and add a new **ExtendPathMiddleware.swift** file to it with the following contents.

In a modern Vapor application, a middleware should conform to the **AsyncMiddleware** protocol. This protocol uses the brand new async/await feature that's available from Swift 5.5. Of course, there's an older Middleware protocol that returns with an EventLoopFuture, but it's clear to see that we should avoid that because it's way more complicated to work with futures and promises. Let's just say for now that a function with an async signature is something that you have to wait for; so for example, the next parameter is an **AsyncResponder**, and this is why we have to put the await keyword before the method

SAMPLE

when we call it. You can read more about async / await and concurrency on the official Swift website.

```
/// FILE: Sources/App/Middlewares/ExtendPathMiddleware.swift

import Vapor

struct ExtendPathMiddleware: AsyncMiddleware {

    func respond(
        to req: Request,
        chainingTo next: AsyncResponder
    ) async throws -> Response {
        if !req.url.path.hasSuffix("/") && !req.url.path.contains(".") {
            return req.redirect(
                to: req.url.path + "/",
                redirectType: .permanent
            )
        }
        return try await next.respond(
            to: req
        )
    }
}
```

When a request arrives at the server, we're going to check if the **path** of the request URL has a forward slash suffix when it doesn't contain an extension (dot character). If not, we can simply redirect the client to a path that we would like to see, using a **.permanent** redirection type. If the path contains a trailing slash, we can use the **Responder** object and "pass the chain" to the next responder.

Think of this as a chain of request handlers that will be called one after another. Every member of the chain can alter the request object, extend it with additional information (e.g. authentication) or terminate the execution by sending a response. The final element in your chain is usually the request handler that you register with the **.get, .post**, etc. methods on the app or router instance.

Now that we defined a middleware, we still have to register it so it can be part of the chain. We can do this in the **configure.swift** file using the **middleware** property on our application.

```
/// FILE: Sources/App/configure.swift

import Vapor

public func configure(
    _ app: Application
) throws {

    app.middleware.use(
        FileMiddleware(
            publicDirectory: app.directory.publicDirectory
        )
    )

    app.middleware.use(ExtendPathMiddleware())

    let routers: [RouteCollection] = [
        WebRouter(),
        BlogRouter(),
    ]
    for router in routers {
        try router.boot(routes: app.routes)
    }
```

```
}
```

In Vapor, it's relatively easy to alter the responder chain through middlewares. You can use middleware for many things, and in this example, we were only scratching the surface. You need to keep in mind that this little path extension middleware is only good for **GET** requests. In a real-world server application, you might want to check the request method and perform additional checking if you want to use such a middleware.

What about the last menu item? Let's use that empty **web.js** file that we created at the beginning of the tutorial. We're going to simply display an alert, but of course, you can use this template to spice up the website with some fancy animations.

```
/* FILE: Public/js/web.js */

function about() {
    alert("myPage\n\nversion 1.0.0");
}
```

That's the about menu, nothing serious for now, but I hope that this example gives you a basic idea about how to import and use javascript files. You can use **jQuery** or anything else to make your life better, but in this book, we're only going to write **Vanilla JavaScript**.

SUMMARY

This chapter was all about getting started with Vapor and the view templates. SwiftHtml is real easy to start with: the most difficult part is when you have to create the connection between the library and Vapor. Using a DSL to write type-safe HTML code is nice since the compiler can catch your errors at build time and you'll make fewer mistakes. We've seen how you can create modules to separate individual components in your application. Modules are really powerful code organization tools; using standalone Routers and Controllers helps us to maintain clean code everywhere. We've also learned about the fundamentals of routing and played around a little bit with an async middleware. In the next chapter, we'll focus on persisting blog entries into a local SQLite database using Fluent.

SAMPLE SAMPLE SAMPLE SAMPLE SAMPLE SAMPLE