




Want to make macOS apps? I wrote a whole book showing you how to transfer skills the easy way.

**CLICK
HERE**

NEW! **Master Swift design patterns with my latest book!** >> (/store/hacking-with-macos?source=macos-1) (/store/swift-design-patterns)

Get started with Vapor 3 for free

Paul Hudson (/about) April 2nd 2018  @twostraws
(<https://twitter.com/twostraws>)



With the release of Swift 4.1 (</articles/50/whats-new-in-swift-4-1>), work on Vapor 3 is finally wrapping up and the team are busy tagging all the Vapor repositories for the 3.0 release.

So, now is the perfect time to dive in to Vapor 3 and see how it works. I already wrote a book teaching Vapor 3 (available to buy here (<https://gum.co/server-side-swift-vapor>)), but here I'm going to walk you through building a new project completely from scratch.

Note: This tutorial is focused on building a real project. Although I'll be explaining what all the code does, if you want more depth you should really check out my Vapor 3 book.

Part One: Getting started with a basic server

To get started, open a macOS terminal window and run this command:

```
brew install vapor/tap/vapor
```

That will install the Vapor 3 toolbox, which is responsible for creating, building, and testing projects. If you don't have Homebrew installed on your Mac already, you'll need to install that first to get the "brew" command:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homeb
```

Note: installing Homebrew will ask for your password.

Now we can go ahead and create a new project for Vapor 3. In this example we're going to build a simple web chat called WaffleSpot: users can enter a username and a message into a web form, which we'll then save to an SQLite database. They'll also

be able to read messages posted by others, and just for fun we'll also serve up our data as both HTML and JSON so other apps can read our data.

First, change to your Desktop folder if you weren't already there:

```
cd Desktop
```

Now we're going to use the Vapor toolbox to create our new project. You need to tell it what template it should use, and for this project we're going to use a template I built called Vapor-Clean.

This template is available on GitHub at <https://github.com/twostraws/vapor-clean> (<https://github.com/twostraws/vapor-clean>) and does nothing special. In fact, the whole point of the template is that it does as little as possible – it's the absolute least amount of code required to get a working Vapor server.

Run this command now:

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

That generates a new project called WaffleSpot and changes into the directory. By default these projects are pure Swift code – there is no Xcode project attached – but before we ask the Vapor toolbox to generate the Xcode project we need to make some changes to the Package.swift file.

Package.swift is responsible for declaring which dependencies we want to use. The Vapor Clean template does the absolute minimum required to get Vapor 3 up and running, so it doesn't include Vapor's Fluent framework for database access or its Leaf framework for template rendering.

So, open Package.swift now and change it to this:

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

let package = Package(
    name: "WaffleSpot",
    dependencies: [
        .package(url: "https://github.com/vapor/vapor.git", from: "3.0"),
        .package(url: "https://github.com/vapor/leaf.git", from: "3.0"),
        .package(url: "https://github.com/vapor/fluent-sqlite.git", from: "3.0"),
    ],
    targets: [
        .target(name: "App", dependencies: ["Vapor", "FluentSQLite", "Leaf"]),
        .target(name: "Run", dependencies: ["App"]),
        .testTarget(name: "AppTests", dependencies: ["App"]),
    ]
)
```

That adds both Fluent and Leaf repositories on GitHub, and adds them both as dependencies for our project.

Those are all the dependencies we need for this project, so the next step is to ask the Vapor toolkit to generate an Xcode project so we have a comfortable working environment. Run this command in your terminal window:

```
vapor xcode
```

That will fetch all the Vapor dependencies from GitHub, then generate the Xcode project. When it finishes press “y” to open Xcode with the project.

When Xcode has launched, you’ll find you can press Cmd+B to build your project, but Cmd+R to run won’t do anything. This is because Vapor ships with a number of targets, but the one you want – “Run” – won’t be selected by default. To fix that, go to

the Product menu and choose Scheme > Run.

Note: At the risk of sounding patronizing, make sure you select My Mac for your target device, as opposed to any iPhone or iPad.

With that change, press Cmd+R to build and run your code. It will take 30 seconds or so the first time, but eventually your Xcode log will say “Server starting on `http://localhost:8080 (http://localhost:8080)`”, which means that Vapor is ready to serve requests.

My Vapor Clean code includes one trivial route just to make sure Vapor is working correctly. Routes are how Vapor servers send content externally - it’s the “hello” part in **`http://localhost:8080/hello (http://localhost:8080/hello)`**. Try loading `http://localhost:8080/hello (http://localhost:8080/hello)` in your web browser now, and you should see “Hello, world!” printed out to show that your Vapor server is working correctly.

Part Two: Returning a message

We’re building a simple chat server where users can post messages freely. To make that work, we’re going to create a new `Message` struct that holds one message posted by a user: a unique identifier for the message, the name of the user that wrote it, the message content itself, and the time it was posted. The unique identifier is helpful in case we ever need to delete a specific message – it lets us identify every message individually, even if they were posted with the same username and content.

In Xcode, press Cmd+N to create a new file, choose Swift File, then click Next. Name it `Message.swift`, and make sure you have App selected for both Group and Targets. This will ensure the file is added to the correct part of your project.

The `Message.swift` file that Xcode creates for you will just have a single line of code: `import Foundation`. We need to add some more code below that to define our `Message` struct, so please do that now:

```
struct Message {  
    var id: UUID?  
    var username: String  
    var content: String  
    var date: Date  
}
```

The `id` property uses a Foundation type called `UUID`, which is a Universally Unique Identifier. If you allow the system to generate these you can be sure they will always be unique, no matter how many you generate. I've declared it as an *optional* `UUID` because later on we're going to ask Vapor to generate them for us.

You can store more information if you want, but that's enough for now. However, before we're done with this struct we're going to make it conform to a Vapor protocol called `Content`. This is used for any kind of data that you want to send and receive, and in our app we want to receive new messages from users and send back messages in our web pages.

So, first add this `import` line:

```
import Vapor
```

Now modify the struct to add the `Content` protocol:

```
struct Message: Content {  
    var id: UUID?  
    var username: String  
    var content: String  
    var date: Date  
}
```

As you can see we haven't had to change anything else in the struct – just adding the `Content` protocol is enough for Vapor.

Now that we have a `Message` data type, we can create a new route that uses it. By default Vapor routes are stored in `routes.swift` inside the Sources > App group, and while you're learning it's easy to continue putting routes in there. This is a simple enough app, so we'll do just that, so open `routes.swift` now.

Note: As your app grows beyond non-trivial size, you'll learn that putting routes into separate controllers helps keep your code more organized and less tightly coupled. This is covered in detail in my book, but for this trivial app we'll just put routes into `routes.swift`.

Inside `routes.swift` you should see this:

```
public func routes(_ router: Router) throws {  
    router.get("hello") { req in  
        return "Hello, world!"  
    }  
}
```

The call to `router.get()` is our existing test route: it attaches code to `"/hello"` on our server, and returns a string. The `get()` part refers to the HTTP GET method, which is used when reading data from the server.

In that code, the `req` coming in to our route closure stores the user's request. Yes, they requested the "hello" route, but they might also have attached form data, query parameters, and more – all information that Vapor makes available to us inside that `req` parameter.

For now we want to check that our new `Message` struct is working as intended, and that Vapor is happy to use it as content. So, we're going to create a new "send" route that creates a test `Message` instance with example data and returns it directly to Vapor.

Replace the existing “hello” route with this:

```
router.get("send") { req -> Message in
    let msg = Message(id: UUID(), username: "@twostraws", content: "He
    return msg
}
```

There are three important parts to that code:

1. I’ve modified the closure signature so that it accepts a request and explicitly returns a `Message` instance. Swift can sometimes figure out the return type for you, but it’s usually best to be explicit.
2. The new `Message` instance is created with a random identifier generated by a new `UUID` instance, and the current date and time.
3. We return the new `Message` directly from our route. Because it conforms to the `Content` protocol, Vapor will automatically convert it to JSON and send that back.

Build and run your code now, then visit <http://localhost:8080/send> (<http://localhost:8080/send>) in your browser – you should see some JSON coming back containing our example data, with a random UUID. Each time you refresh the page you should see a completely different UUID – they really are guaranteed to be unique.

Part Three: Rendering a HTML form

Our little app isn’t much use if we only ever write the same message again and again, so the next step in our project is to add a web form so that users can enter their own username and message content.

Although Vapor is perfectly capable of rendering HTML directly from your Swift code, it really isn't pleasant – you'd need to recompile your project every time you made a small change, and you'd also clutter up your code in no time at all.

The best place to put your HTML is in separate files, known as *templates*. These can be complete HTML pages, but more often than not they contain placeholders for content you can inject in Swift. For example, your HTML might contain all the styling for your page, then have a gap for where the user's profile data should be. Trying to hard-code user profile data into HTML doesn't make sense, so instead you can write Swift code to generate that data and pass that into the template to be rendered.

Although Vapor is capable of using other templating languages, its main language is called Leaf. Leaf lets you write as much HTML as you want, then intersperse small amounts of layout functionality as needed.

Let's start with a simple template that shows a message posting form:

```
<html>
<body>
<h3>WaffleSpot</h3>
<form method="get" action="/send">
<p>Username: <input type="text" name="username" /></p>
<p><textarea name="content" rows="10" cols="40" placeholder="Write you
<p><input type="submit" /></p>
</form>
</body>
</html>
```

Save that to a file called “home.leaf” inside the Resources/Views directory of your WaffleSpot project. As you can see that's just pure HTML – there's no Leaf code in there, and certainly no Swift code either.

The HTML contains a single `<form>` that points to our “send” route, and includes two places for the user to enter data: a username field and a content field. When they click Submit that data will be sent to our “send” route for us to work with – we’ll come to that bit later on.

By default, Vapor doesn’t know about the existence of Leaf, so in order to start rendering templates we need to configure our app to use Leaf for all template rendering.

Alongside `routes.swift` in the Sources > App group you’ll also see `configure.swift`. This is where one-time configuration for your server is performed, and you should see a comment in there saying `// Configure the rest of your application here`. That’s where we need to register a Leaf provider, which will ask Vapor to render templates using Leaf whenever we need it.

Add this `import` line near the top of `configure.swift`:

```
import Leaf
```

Now add these two lines of code below the `// Configure the rest of your application here` comment in `configure.swift`:

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

Leaf knows to look in Resources/Views by default, so that’s all the code we need to configure Leaf templates inside our project.

So, we have some HTML we want to render, and we have configured Vapor to render Leaf templates. The last step here is to add a new route that will render `home.leaf` when requested. The Vapor Clean template provided a “hello” route, and we already

replaced that with a “send” route. But this time we’re going to attach a route to the home page – the page people see when they visit `http://localhost:8080/` (`http://localhost:8080/`) directly.

To create this homepage route – a root route, as it were – just don’t provide any parameters to `router.get()`, like this:

```
router.get { req in
    return "This is the homepage"
}
```

Of course, we need to return more than just a simple string – we need to have Leaf render our template.

To make this happen you need to learn three new things:

1. Now that we’ve registered Leaf as our default template engine, Vapor lets us use `req.view()` to create an instance of the Leaf template renderer. Vapor will automatically re-use these for us, maximizing performance.
2. If you try to render a template that doesn’t exist Vapor will throw an error. So, we need to call our rendering code using `try`. You don’t need to worry about catching the error here, though – all route closures are automatically throwing, so Vapor will catch the error for us.
3. Loading a template and executing any Leaf code takes time, and Vapor is designed to be as quick as possible. So, when you ask Vapor to render a template it will return a `Future<View>` instead – a promise that the work will complete at some point in the future, but isn’t actually done just yet.

The concept of futures is quite tricky when you’re just learning, and I devote quite some space in my book to making it as clear as possible. However, here we don’t need to worry about it: all you need to know is that rendering a template returns a `View` object, but Vapor will wrap that inside a future to make `Future<View>` because the actual rendering work hasn’t happened yet.

All this matters because we need to specify a return value from our closure. We used `Message` previously to return a message obvious, but this time we need to use `Future<View>` to tell Vapor that a rendered template will be coming at some point. Vapor understands that the rendering work hasn't actually taken place, so will automatically wait for the future to complete before sending its contents back to the user's web browser.

After all that, our new route takes just three lines of code:

```
router.get { req -> Future<View> in
    return try req.view().render("home")
}
```

As a reminder:

1. We don't specify an route to `get()` because we want to attach this to the homepage.
2. `req` is the user's request, telling us any information they submitted.
3. We return a `Future<View>` to represent the Leaf template that will be rendered soon; Vapor will automatically wait for that future to complete and send the finished HTML back to the web browser.
4. As we have registered a `LeafProvider` inside `configure.swift`, we can use `req.view()` to have Vapor create a Leaf renderer that we can use to render templates.
5. Once we have that, we can call the `render()` method and ask it to render `home.leaf`. The `.leaf` file extension is left off.

Build and run your code now, then visit `http://localhost:8080/` (`http://localhost:8080/`) in your web browser. All being well you should see your simple web form – progress!

Part Four: Connecting to a database

If you click the Submit button, you'll go back to the same "send" route we wrote earlier – it creates an example `Message` instance and sends it back as JSON.

What we *really* want to happen is for the user's form data to be converted to a message and to be saved in a database for others to read. But before we can do that, we need to set up the database that will store our data – otherwise we'll have nowhere to put our data.

There are lots of ways to create and manipulate databases, but when you're starting out the easiest is to use an SQLite in-memory database – a database that is automatically destroyed when your server stops running.

SQLite is easily the most commonly installed database in the world – it's on every iPhone ever made, and every Android device ever made, for example. It comes pre-installed on macOS, and we already requested the `FluentSQLite` dependency when configuring our `Package.swift` file, so it's ready to be used in our project.

Configuring a database for our project requires three steps:

1. Creating an SQLite database.
2. Modifying our `Message` struct so that it can be used with SQLite.
3. Asking Vapor to configure the database for messages when our app launches.

Let's start with the first step: creating an SQLite database. To make things easier to follow, I'm this down into four smaller steps so I can explain what everything does.

First, we need to add an import for `FluentSQLite` so that we can connect our SQLite database inside `configure.swift`. So, add this line to the top of `configure.swift`:

```
import FluentSQLite
```

Second, we need to register an instance of `FluentSQLiteProvider` with Vapor's app services. This sets up the bare bones of SQLite functionality, and allows us to connect in-memory databases and file-based databases.

Add this line to `configure.swift` now, below the previous Leaf configuration code:

```
try services.register(FluentSQLiteProvider())
```

Third, we need to create an `SQLiteDatabase` instance pointing at wherever we want to store our data. In this example app we're going to use an in-memory database so that our data gets wiped when Vapor stops running.

Add this line below the previous one.

```
let sqlite = try SQLiteDatabase(storage: .memory)
```

Finally, we need to create an instance of Vapor's `DatabaseConfig` struct, adding our SQLite database using the built-in `.sqlite` identifier. Identifiers are there to let you register multiple databases simultaneously, but in this app we only want one so we're going to add our `sqlite` database there and then register the whole `DatabaseConfig` with Vapor's services system.

Add this code below the previous two lines:

```
var databases = DatabaseConfig()
databases.add(database: sqlite, as: .sqlite)
services.register(databases)
```

Our Vapor app is now able to read and write to an in-memory SQLite database, so that's our first step complete.

Just having an SQLite database available to us isn't enough to make this app work – we need to configure our `Message` model so that Vapor is able to convert it into data that SQLite can store.

This is the second of our three steps, and again I want to break it down into three smaller steps so I can explain what everything does. All these small steps are done in `Message.swift`, so please open that now.

First, add `import FluentSQLite` alongside the existing `import` statements at the top of the file. This lets us use Fluent's SQLite wrapper with our struct.

Second, add `SQLiteUUIDModel` to the list of protocol conformances for the `Message` struct. A model is any Swift type that should be saved to a database, and this specific protocol is for SQLite database models that have a `UUID` as their identifier.

Finally, add `Migration` to the list of protocol conformances for the `Message` struct. This protocol allows Vapor to configure our database to store instances of this type. In SQL, one database can have many tables – for example, a `BookStore` database might have a table for books, a table for customers, and a table for purchases. Adding `Migration` here allows this struct to be converted to a `messages` table.

And that's it – those are all the changes we need to make to `Message` to have them work with Vapor. What you're seeing here is Fluent – Vapor's remarkably powerful database abstraction layer. Its job is to allow us to translate Swift data types such as `Message` into database data, without us having to write lots of code by hand.

Your final `Message.swift` file should look like this:

```
import FluentSQLite
import Foundation
import Vapor

struct Message: Content, SQLiteUUIDModel, Migration {
    var id: UUID?
    var username: String
    var content: String
    var date: Date
}
```

That's the second of three steps completed, so there's one last step we need to complete before our database setup code is finished. Making our `Message` struct conform to the `Migration` protocol means that it is *able* to be created in the database when Vapor starts, but to actually make that work happen we need to add three extra lines of code to `configure.swift`.

So, open `configure.swift`, and add these three directly below the previous configuration code we wrote:

```
var migrationConfig = MigrationConfig()
migrationConfig.add(model: Message.self, database: .sqlite)
services.register(migrationConfig)
```

That attaches our `Message` struct to the same `.sqlite` database identifier we registered earlier. As a result, when our app launches Fluent will automatically make sure a `messages` table exists and is able to hold the four properties that our `Message` struct contains.

After all that work, I'm afraid you won't see much visible difference when you build and run your code. In fact, the only difference you'll see is in Xcode's debug log output – it should now show this:

```
[ INFO ] Migrating 'sqlite' database (FluentProvider.swift:28)
[ INFO ] Preparing migration 'Message' (MigrationContainer.swift:50)
[ INFO ] Migrations complete (FluentProvider.swift:32)
```

Fortunately, this was the hardest part of our tutorial – it's surprisingly easy from here on!

Part Five: Saving messages

At this point in our project we have created our basic Vapor server, designed a `Message` struct, rendered a HTML form, and configured a database where we can save and load user messages.

Now it's time for the most important part of this project: when the user submits their form with information we want to save it to our database so that later on we can show a list of messages that have been posted.

We already have a “send” route that looks like this:

```
router.get("send") { req -> Message in
    let msg = Message(id: UUID(), username: "@twostraws", content: "He
    return msg
}
```

That was fine just for testing, but before we dive into the database code we need to make one important change: `router.get("send")` needs to become `router.post("send")`. The HTTP GET method is designed to read data from a server, and shouldn't make changes to content. On the other hand, POST is designed to send changes – when your web browser shows you that “Are you sure you want to resubmit this form?” message, it's because the form uses POST and you might accidentally be writing data twice.

So, change the “send” route to this:

```
router.post("send") { req -> Message in
    let msg = Message(id: UUID(), username: "@twostraws", content: "He
    return msg
}
```

If you remember, our `home.leaf` also used the HTTP GET method for sending its form data. Look for this line:

```
<form method="get" action="/send">
```

Again, that's fine for when you're just reading data, but now that we're preparing to write data we need to use POST here too. Change the line to this and make sure you save the changed file:

```
<form method="post" action="/send">
```

Now that we've told Vapor this route will make changes to our application, it's time to read the user's data they submitted in the form.

Each route closure gets given the user's request as its parameter – that's the `req` constant we've had all along. We can use that request to decode the form content by passing in the name of the form elements we want to read.

Vapor provides us with two ways of decoding form content: the `decode()` method extracts the element you ask for asynchronously, which is useful if you expect to receive lots of data; and the `syncDecode()` method does the same work just *synchronously* – it returns the data immediately.

This is another example of Vapor's use of futures. I mentioned them earlier when talking about rendering templates, but Vapor uses them a *lot* as you'll see. For this app using `syncDecode()` is a good idea: it keeps our code nice and easy, and works perfectly fine for the small amount of data we expect.

So, start by deleting the contents of the "send" route, like this:

```
router.post("send") { req -> Message in  
  
}
```

Xcode will complain, but that's OK – we're going to fill it again.

The first code to write in there will use `syncDecode()` to pull out first the username string then the content string. You need to be explicit with your types here, because you might want to extract integers, UUIDs, or other data types.

So, add these two lines of code to the “send” route:

```
let username: String = try req.content.syncGet(at: "username")
let content: String = try req.content.syncGet(at: "content")
```

Second, we need to create a `Message` struct from those two. This time we're going to use `nil` for the identifier, because `Fluent` will fill that in for us automatically.

Add this line below the previous two:

```
let msg = Message(id: nil, username: username, content: content, date:
```

Now for the best part: we need to convert that `Message` instance into something `SQLite` can store, save it to the database, then return back the saved object with an automatically created `UUID` as its identifier.

That might sound hard, but really it's just one line of code. Add this below the previous three:

```
return msg.save(on: req)
```

Now, Xcode will *still* complain at this point. This is because our “send” closure is defined like this:

```
router.post("send") { req -> Message in
```

It says we're going to return a `Message` instance, but that is no longer true. When you call `save()` on a `Message` what you get back is a `Future<Message>` – a promise that your work will be done, but isn't necessarily completed just yet.

Just like with Leaf templates, we don't really care that the work isn't done yet – we'll let Vapor figure that out. So, rather than returning a `Message` we actually want to return a `Future<Message>`, like this:

```
router.post("send") { req -> Future<Message> in
```

All being well your finished route should look like this:

```
router.post("send") { req -> Future<Message> in
    let username: String = try req.content.syncGet(at: "username")
    let content: String = try req.content.syncGet(at: "content")
    let msg = Message(id: nil, username: username, content: content, d
    return msg.save(on: req)
}
```

Build and run your code, then try writing some values in the form and submitting them – you should see some JSON coming back, with the `id` and `date` fields filled in appropriately.

Part Six: Reading the waffle

If our web app was designed to let users scream into a void, it would be done. However, really we want to let folks read what others have posted, which means we need a way to read messages that have been posted already.

Our root route – that’s the route that renders when users visit `http://localhost:8080` (`http://localhost:8080`) directly – renders our `home.leaf` template right now, and it’s going to continue doing that. However, we need to provide it with some dynamic content: we want to read all the messages that have been saved, and show them inside the Leaf template.

It’s not hard to send data to a Leaf template: when you call `render()` you can pass in a dictionary of data, or pass in a struct that conforms to `Codable`, like this:

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

It’s even surprisingly easy to fetch the array of messages to load. Thanks to Fluent, it’s just one line of code:

```
let messages = Message.query(on: req).all()
```

However, where things get tricky is how we bridge the two. You see, fetching messages from the database is another example of how Vapor uses futures – the data will come back at some point, but probably isn’t ready yet.

We’ve dealt with futures twice already: once when rendering Leaf templates, and again when saving messages. However, both times we’ve been able to ignore the fact that futures were involved – we returned a `Future<View>` from Leaf, and a `Future<Message>` from Fluent, and let Vapor wait for the future to complete on our behalf.

This time we can’t take that approach: our database call needs to finish and have sent us the list of messages before we can start rendering the Leaf view, because the Leaf view will show those messages in our HTML.

Instead, we need to use a method called `flatMap()`, which is designed to attach work to a future. This work will only start happening when the future completes, and will be sent whatever data was inside the future – in our case the list of messages from the database.

Tip: Everyone who learns Vapor finds this part difficult. If you find it difficult too it's *not* because you're not smart enough or you're too junior for Vapor – everyone finds it's difficult at first, so give yourself a break!

When you call `flatMap()` you're telling Vapor you want to attach some work that will run when the future completes. In this case, we want to convert the array of messages that Fluent gives us into a rendered Leaf view.

I'll give you the full code in a moment, but first I'd like to focus on the call to `flatMap()`. Here's how it looks:

```
return Message.query(on: req).all().flatMap(to: View.self) { messages
```

That will fetch all the messages in our database, and when that completes and the messages are available it will prepare to convert those messages into a Leaf View.

The `messages` in part is important: because this code only runs when the work has actually happened (the future is finished), that `messages` parameter will be a real array of `Message` instances that we can pass to our Leaf template.

Inside that `flatMap()` it's our job to render the Leaf template just like we did before, except now we can pass in some dynamic context – some data that we want Leaf to display that *isn't* hard-coded into the HTML.

Like I said earlier, you can use dictionaries here if you want, but in serious apps you'll probably want to use `Codable` structs instead so you get better type safety.

Here, though, just creating a dictionary with one key (“messages”) and one value (our array of messages from Fluent) is enough. Modify the root route to this:

```
router.get { req -> Future<View> in
    return Message.query(on: req).all().flatMap(to: View.self) { messa
        let context = ["messages": messages]
        return try req.view().render("home", context)
    }
}
```

That will send our array of messages to the `home.leaf` template, but don't try to run your code just yet. You see, even though we're sending the messages array to Leaf, our Leaf template is just pure HTML – it doesn't have anywhere to show those messages.

This is where we need to add some Leaf code to `home.leaf`. Leaf's templating language has a handful of useful instructions that add some display logic to our HTML. You don't ever try to calculate things here – your Swift code should do all that. Instead, Leaf is just here to adjust the way things are presented.

To complete this step in our project we need to use two types of Leaf tags: a `#for()` tag that loops over arrays, and a simple `#()` tag that prints out values. Using these we can loop over all the messages we passed in, and print out each value.

When we rendered our view, we passed in context from Swift like this:

```
let context = ["messages": messages]
```

That used "messages" as the name for the array inside our context, which means we'll be able to use the same name inside our Leaf code.

Open `home.leaf`, then add this new code directly after the `</form>` line:

```
<h2>Messages from others...</h2>

#for(message in messages) {
    <p>At #(message.date), #(message.username) wrote: "#(message.conte
}
```

That loops over the `messages` array we passed in, using syntax that's almost identical to Swift. Then, for each message it finds inside the array, it prints out the date, username, and content.

Notice how Leaf lets us mix HTML alongside its own instructions – it makes for a really clear, clean way to render pages.

Save your changed `home.leaf` file then try running the project again. Remember, SQLite in-memory databases get wiped each time we restart the Vapor server, so you'll need to post a message then return to the homepage before you see anything.

Part Seven: Adding some polish

At this point you should be able to post messages, then return to `http://localhost:8080` (`http://localhost:8080`) to see them listed. It works, but it's not ideal – with a few small changes we can make a much nicer experience.

Specifically, we're going to make four changes:

1. Message times are being shown in a weird format – “544373868.28156” is meaningless to most people, and we can do better.
2. Posting a message shows you the JSON of the message you wrote, rather than taking you straight back to the homepage.
3. New messages always appear at the bottom, which makes the chat annoying to use.

4. Didn't I promise we'd also make our app serve up all messages as JSON so that other apps could read our data?

Fixing all four of those only takes five minutes, but – *gulp!* – you will need to touch futures again to solve the second one.

Let's start with the first problem: times are being shown as "544373868.28156" rather than something human-readable like "15:48". What you're seeing here is Foundation's internal way of storing data – the number of seconds that have elapsed since January 1st 2001.

This format is fine for internal use, but when working with humans it's much nicer to show something like "15:48" or "3:48pm". This annoyed me so much I actually wrote a custom Leaf tag to fix it, and the Vapor team kindly merged it into the Leaf project so that everyone can use it.

The tag is called `#date()` and take two parameters: the `Date` value you want to work with, and the format you want to print out. The format is written in the same you would use with a regular `DateFormatter`.

So, to make our output look a lot nicer we're going to modify `home.leaf` to use the `#date` tag. If you prefer using the 24-hour clock (15:38), use this:

```
<p>At #date(message.date, "HH:mm"), #(message.username) wrote: "#(mess
```

If you prefer using the 12-hour clock (3:48pm), then use this:

```
<p>At #date(message.date, "h:mma"), #(message.username) wrote: "#(mess
```

Either way, that's our first problem solved.

The second problem is that when you post a new message, all you see is the JSON of the message you wrote – you need to navigate back to `http://localhost:8080` (`http://localhost:8080`) by hand. What we really want to do is redirect users back to

the root route when their message has been saved, so they can see it alongside others.

Vapor has a simple way of redirecting users from one page to another:

`req.redirect()`. You pass it the path they should be forwarded to, and Vapor does the rest.

However, there's a catch: saving to the database returns a future, so we need to wait for that future to complete before we redirect the user. If we don't wait, it's possible the user might return to the homepage before their message has been saved, and they'll wonder why it's missing.

If you've been paying attention, you might be thinking "aha! I'll use `flatMap()` to chain the work!" However, futures have one more surprise complexity waiting for you, and this one might bend your brain a little. Again: these things are complicated, so if you find them difficult it's just a sign your brain is working correctly.

As I said earlier, loading data from a database returns a future, and when that future completes its work it will contain your data. Rendering a Leaf template also returns a future, and when *that* future completes its work it will contain a rendered `View` that can be displayed in a browser.

We wrote some code earlier that put those two operations together: we fetched data from a database, then rendered it to a view – we combined two futures together. Vapor lets us combine futures together in two different ways:

1. We could get a future future – a double future. This is similar to “optional optional” in Swift (e.g. `Int??`), in that it's something that's possible but you mostly don't want it.
2. We could collapse the two futures together into a single future that completes only when the two things inside both complete.

That second option is what `flatMap()` is for. When you call `flatMap()` on a future it means “when this future completes I'm going to run some code that will also return a future, but please turn this whole thing into a regular future rather than one of those nasty future futures.”

This behavior is where the “flat” part of the name `flatMap()` comes from: it flattens two futures into one. The problem here is that `req.redirect()` *doesn't* return a future – it just returns a simple type called `Response`. So, we can't call `flatMap()` – it's job is to convert future futures into regular futures.

Instead, we need a method that is almost identical, but is used when the work you want to attach to your future just returns a value immediately. This method is `map()`, and apart from its name it's used just like `flatMap()`.

Tip: Both `map()` and `flatMap()` are designed to attach work to be run when a future completes. You can't use the wrong one by mistake – Xcode will refuse to compile. The simple rule is this: if the work you're attaching returns a future, you should use `flatMap()`, otherwise use `map()`.

So, to fix the second problem – to redirect users back to the homepage once their message has been saved successfully – we need to modify the “send” route to this:

```
router.post("send") { req -> Future<Response> in
    let username: String = try req.content.syncGet(at: "username")
    let content: String = try req.content.syncGet(at: "content")
    let msg = Message(id: nil, username: username, content: content, d

    return msg.save(on: req).map(to: Response.self) { _ in
        return req.redirect(to: "/")
    }
}
```

That now returns a `Future<Response>`, and uses `map()` to run some code when the saving future completes. The `_ in` part of the closure is there because Vapor will still give us the message that got saved, but we don't need it – we're just going to redirect to the homepage.

There are two problems left, and they take only a minute to solve each.

First, new messages always appear at the bottom, which makes the chat annoying to use. To fix this we need to modify the code in the root route to include a call to Fluent's `sort()` method. You give this a Swift keypath pointing to the property that should be sorted, then tell it whether to sort ascending or descending.

Find the existing `Message.query()` line in the root route and change it to this:

```
return try Message.query(on: req).sort(\Message.date, .descending).all
```

Note: you need to use `try` with this code, because sorting has the potential to fail.

Finally, I said right at the beginning of this project that we'd also create a simple API endpoint so that other apps could read our message list. This is utterly trivial in Fluent, and you already know everything you need:

1. Create a GET route on "list".
2. Use `Message.query()` to read the messages table, sort it by date descending, then use `all()` to get all rows.
3. Make the route send back a `Future<[Message]>` – an array of all our messages.

Vapor will take care of converting the array to JSON for us, which means the entire route is just three lines of Swift:

```
router.get("list") { req -> Future<[Message]> in
    return try Message.query(on: req).sort(\Message.date, .descending)
}
```

Build and run your code one last time so you can try it out. Remember to add some messages first – all being well you should get redirected back to the homepage after each message is saved.

When you have a few messages in place, and you've admired the date formatting and the descending sort, try going to <http://localhost:8080/list> (<http://localhost:8080/list>) to see the whole message array shown as JSON.

Well done!

Part Eight: Wrap up

This has been a fairly rapid run through building an example Vapor project from scratch. Hopefully you've learned quite a lot:

1. Creating a basic Vapor server.
2. Making Vapor return JSON for your structs.
3. Rendering a HTML form using Leaf.
4. Connecting to an in-memory SQLite database.
5. Parsing the form and saving it to the database.
6. Reading data back out and sending it as Leaf context.

Plus sorting, date formatting, redirecting, and probably more about futures than you might have liked.

This tutorial has just been a brief taste of what Vapor can do. If you want more projects like this one, teaching advanced routing, sessions, authentication, MySQL, Leaf, and more, you should check out my book: *Server-Side Swift* (<https://gumroad.com/l/server-side-swift-vapor>).

Tweet




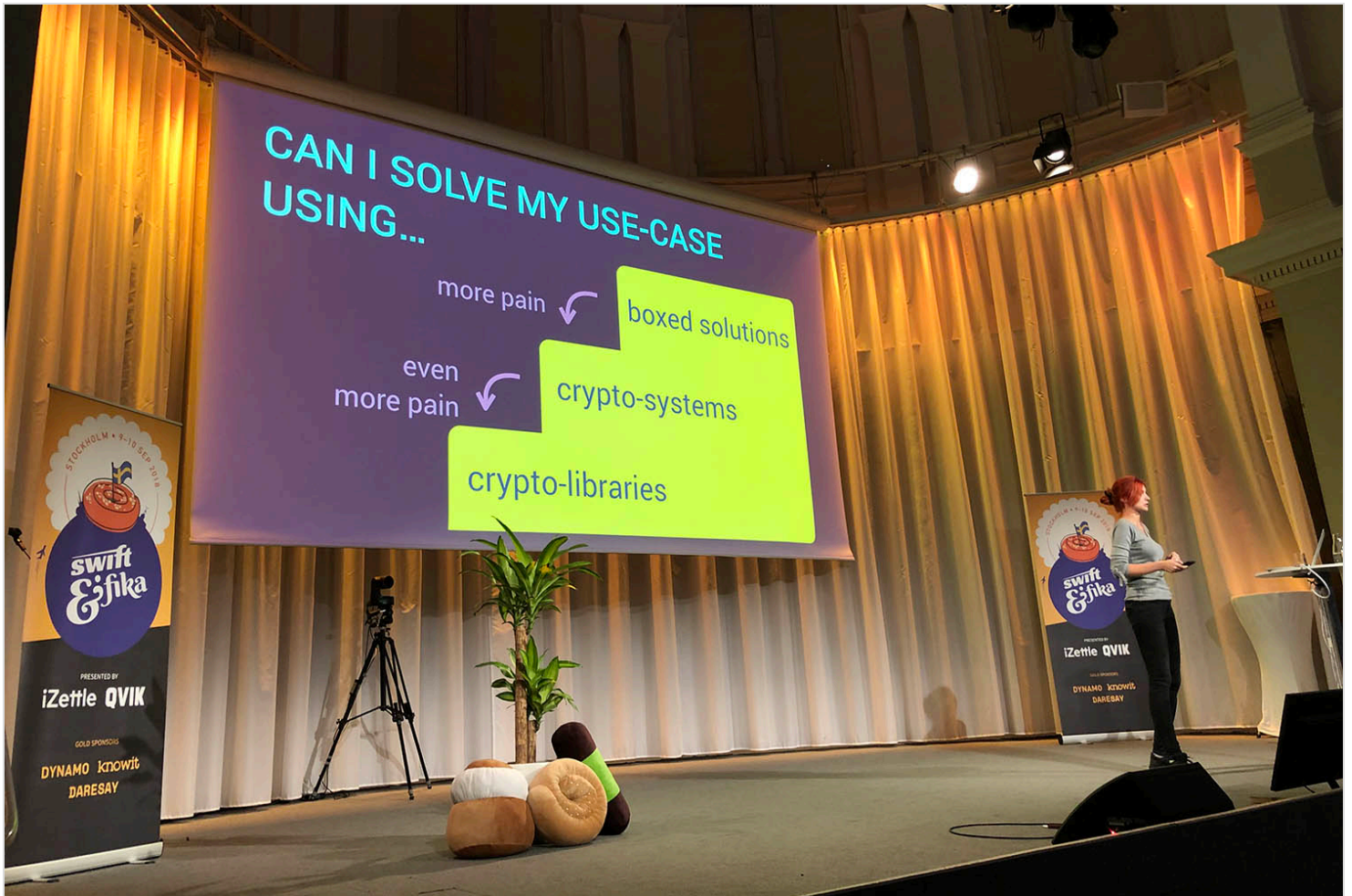
@twostraws (<https://twitter.com/twostraws>)

About the author

Paul Hudson is the creator of Hacking with Swift, the most comprehensive series of Swift books (</store>) in the world. He's also the editor of Swift Developer News (</articles>), the maintainer of the Swift Knowledge Base (</example-code>), and Mario Kart world champion. OK, so that last part isn't true. If you're curious you can learn more here (</about>).

More articles (</articles>)

 **RSS feed (</articles/rss>)**



Conference report: Swift & Fika

(/articles/127/conference-report-swift-fika)



What's new in Swift 5.0

(/articles/126/whats-new-in-swift-5-0)



Conference report: iOSDevUK 8

(/articles/125/conference-report-iosdevuk-8)



Swift Knowledge Base updates for Swift 4.2

(/articles/124/swift-knowledge-base-updates-for-swift-4-2)



(<https://gumroad.com/l/natural-swift>)

Was this page useful? Let me know!



Click here to visit the Hacking with Swift store >> (/store)



@twostraws

(<https://twitter.com/twostraws>)



paul@hackingwithswift.com

(<mailto:paul@hackingwithswift.com>)

Swift, the Swift logo, Xcode, Instruments, Cocoa Touch, Touch ID, AirDrop, iBeacon, iPhone, iPad, Safari, App Store, watchOS, tvOS, Mac and macOS are trademarks of Apple Inc., registered in the U.S. and other countries. Pulp Fiction is copyright © 1994 Miramax Films. Hacking with Swift ©2018 Paul Hudson.

[Privacy Policy \(/privacy\)](/privacy)