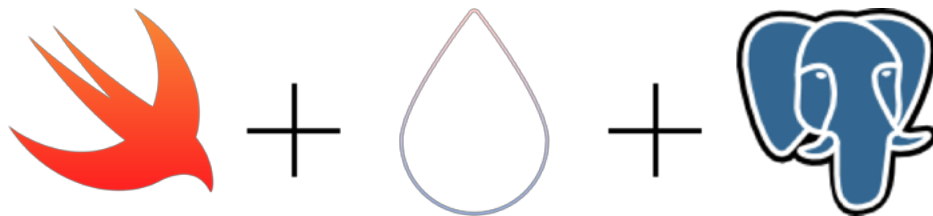


**Johann Kerr**[Follow](#)

Software Engineer @FlatironSchool, formerly Lead iOS & Web Instructor at //  
Jul 30 · 8 min read

# Persisting Data with Vapor 3 and PostgreSQL

A server-side Swift tutorial



At Flatiron School, our team culture encourages learning new tech and sharing knowledge. We constantly look for new technologies to work on and experiment with. We aren't currently using any Vapor/Swift internally, but as an iOS & Web developer, Server Side Swift continues to fascinate me, so I wanted to put together a tutorial on how to use PostgreSQL (an open source relational database) and Vapor 3 (a server side Swift framework) to store our data.

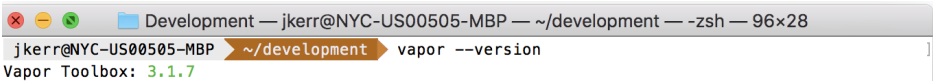
If you're not totally familiar with using SQL, have no fear: Vapor comes with an ORM (Object Relational Mapping Tool) called Fluent that makes working with databases very easy.

Before we dive in, let's take care of a little housekeeping. Ensure that you have the following:

- Vapor Toolbox: 3.1.7
- [Xcode 10 Beta](#)

To check your version of the Vapor Toolbox, open your terminal and type the following:

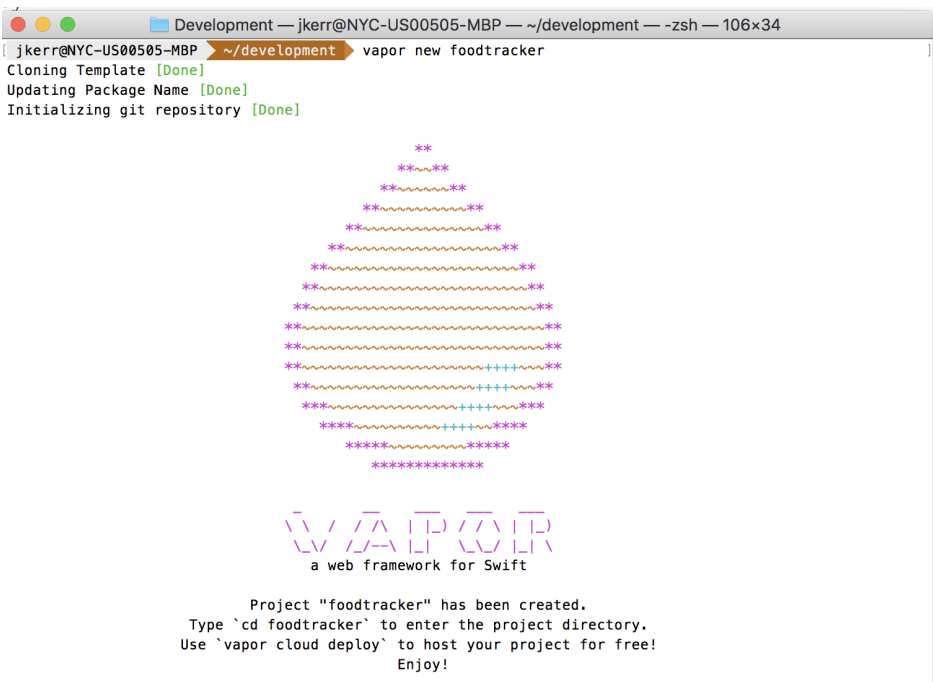
```
vapor --version
```



For this tutorial, we're going to be building a simple application to keep track of meals you enjoy.

## Step 1—Generate a new Vapor project

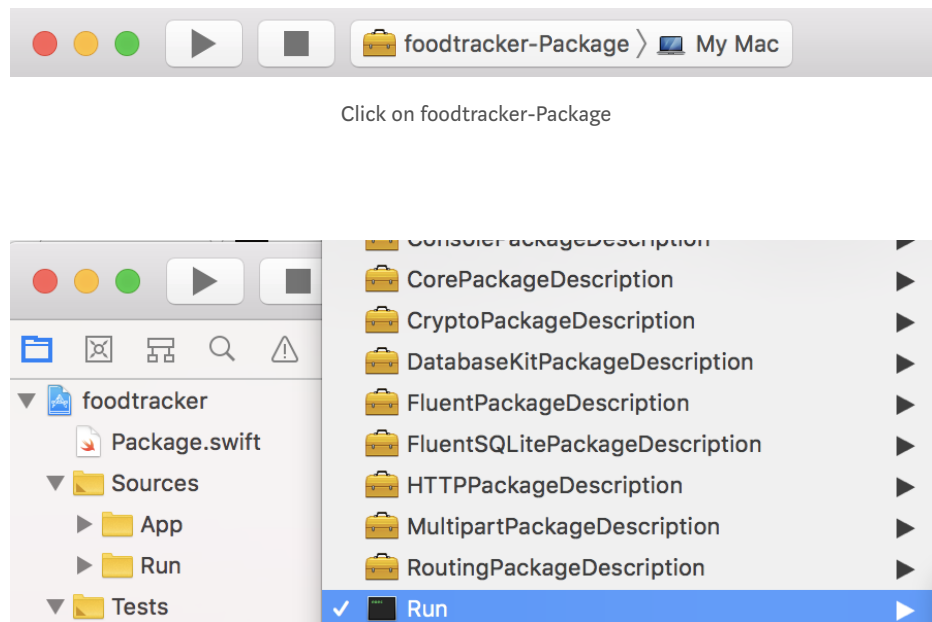
Inside of your terminal, enter `vapor new foodtracker`



## Step 2—Generate an Xcode project file

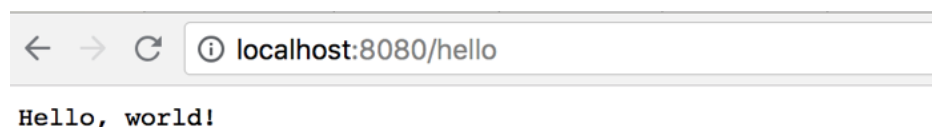
In terminal, type `cd foodtracker && vapor xcode -y`. The `-y` flag tells the vapor xcode generator to automatically open the `.xcodeproj` file.

Now that we have Xcode open, let's go ahead and start our server. First, we need to change our scheme from the `foodtracker-Package` package to `Run` :



Next, click the run button or hit `Cmd + R` to start up your server.

Once, it has finished building, navigate to `localhost:8080/hello`



Now that we have Vapor up and running, let's add PostgreSQL.

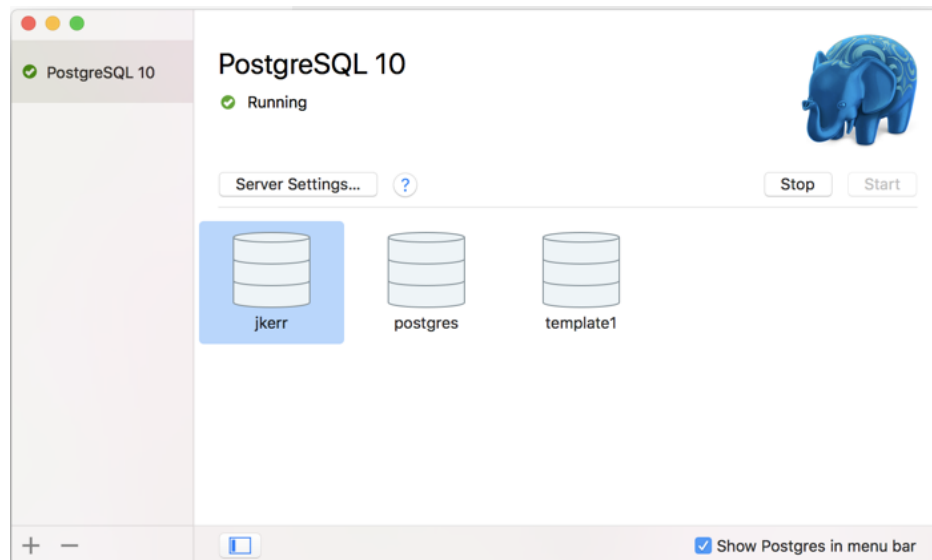
### Step 3—Install PostgreSQL

In terminal, type `brew install postgresql` .

Note: If you don't have Homebrew installed yet, you should! [Homebrew](#) is a package manager that helps developers install tools to their MacOS

devices.

Once `postgresql` is installed, let's install Postgres.app. This is a small program that makes running PostgreSQL on your machine easy. Once you've installed Postgres.app, let's run it.



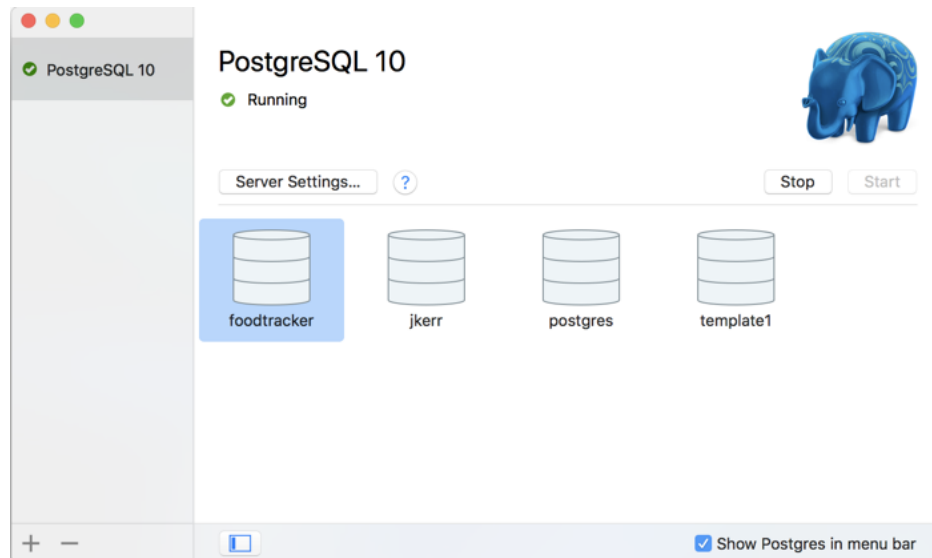
#### Step 4—Create database for foodtracker

In your terminal, enter `psql`. This will allow us to type SQL statements into our terminal to manage our database. In our terminal, type `CREATE DATABASE foodtracker;`.

```
jkerr@NYC-US00505-MBP ~/development/foodtracker master • psql
psql (10.4)
Type "help" for help.

jkerr=# CREATE DATABASE foodtracker;
CREATE DATABASE
jkerr=#
```

Once your database has been created you will notice a new database in Postgres.app:



## Step 5—Connect Vapor to PostgreSQL

To use PostgreSQL with Vapor, we will be using the [FluentPostgreSQL Provider](#). Firstly, what is a provider?

*The [Provider](#) protocol make it easy to integrate external services into your application. —[Vapor Documentation](#)*

To add a provider to your Vapor application, you first need to add the package to Vapor. To do this, we need to replace `fluent-sqlite` with `fluent-postgresql`. Add the following line to your `Package.swift`:

```
.package(url: "https://github.com/vapor/fluent-postgresql.git", from: "1.0.0")
```

Also replace the target `FluentSQLite` with `FluentPostgreSQL`. Your `Package.swift` should look like this:

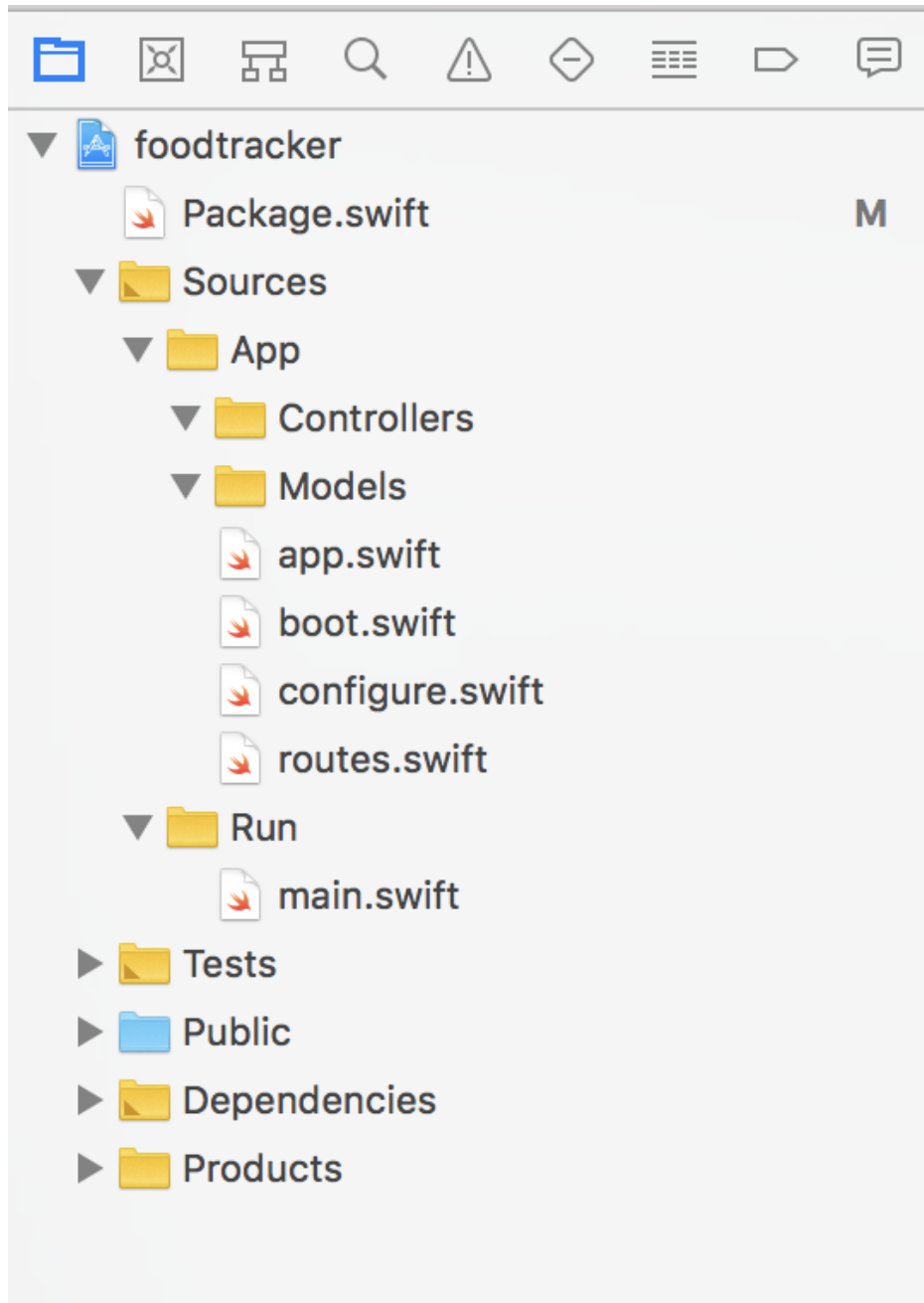
```
1 // swift-tools-version:4.0
2 import PackageDescription
3
4 let package = Package(
5     name: "foodtracker",
6     dependencies: [
7         // 🌊 A server-side Swift web framework.
8         .package(url: "https://github.com/vapor/vapor.git", from: "3.0.0"),
9         .package(url: "https://github.com/vapor/fluent-postgresql.git", from: "1.0.0")
10     ],
11     targets: [
12         .target(name: "App", dependencies: ["FluentPostgreSQL", "Vapor"]),
13         .target(name: "Run", dependencies: ["App"]),
14         .testTarget(name: "AppTests", dependencies: ["App"])
15     ]
16 )
17
18
```

To install our new package, return to your terminal and enter `vapor`  
`xcode -y` This will re-fetch the our packages and generate an Xcode  
project.

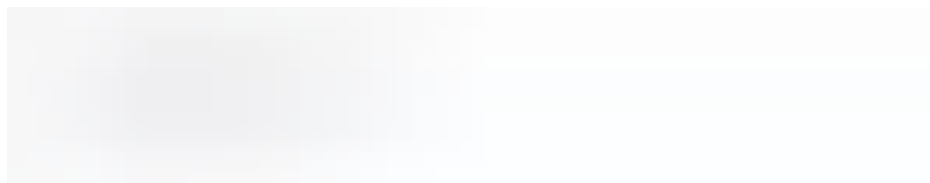
### Step 6—Configure Vapor to work with PostgreSQL

First, things first, we need to remove some of the old code that came  
with our project that is configured to use SQLite.

We need to delete `Todo.swift` and `TodoController.swift`. Your folder  
structure should look like this:



In `routes.swift`, we also need to remove the reference to `TodoController.swift` :



Now, we need to configure our application by changing the code in `configure.swift`.

First, let's change the import on Line 1 from `import FluentSQLite` to `import FluentPostgreSQL`. We also need to register our new provider. In the beginning of the `configure` method, we need to write the following:

```
try services.register(FluentPostgreSQLProvider())
```

Next, we need to create a `PostgreSQLDatabaseConfig` and initialize our `PostgreSQLDatabase` object using our configuration:

```
let config = PostgreSQLDatabaseConfig(hostname:
"localhost", port: 5432, username: "jkerr", database:
"foodtracker", password: nil, transport: .cleartext)

let postgres = PostgreSQLDatabase(config: config)
```

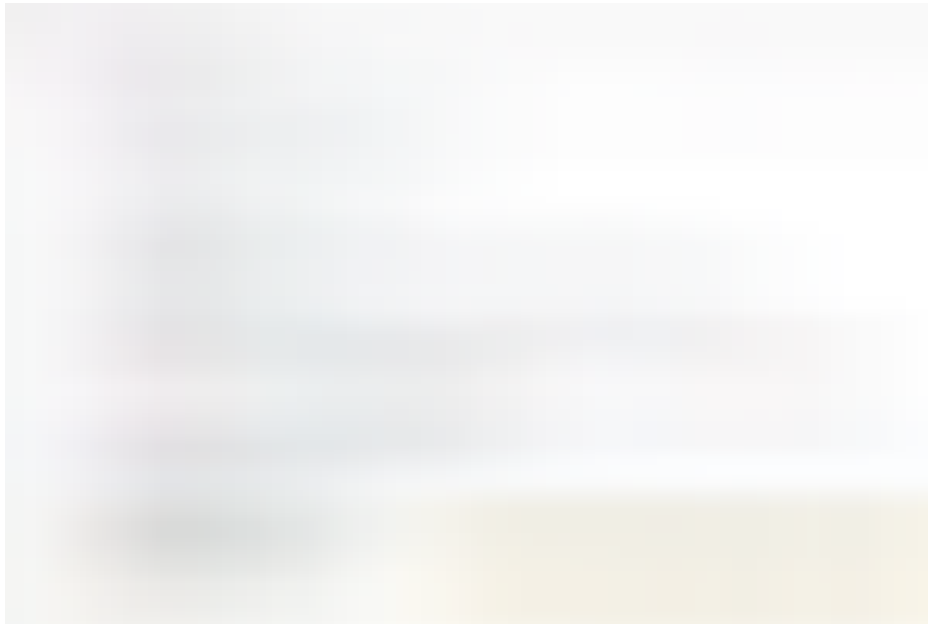
There are a couple of things to note here. The `hostname` points to our local computer and 5432 is the default port that PostgreSQL runs on. The database string here matches the database we just created.

Finally, let's add our new postgres database to our `DatabaseConfig` object:

```
databases.add(database: postgres, as: .psql)
```

Your code should now look like the following:





### Step 7—Add Vapor model

For this project, we’re going to create a `Meal` model to track food. Let’s create a `Meal.swift` file and add it to our models group.

Add the following code to your new file:

```
import Foundation
import FluentPostgreSQL

final class Meal: PostgreSQLModel {
    var id: Int?
}
```

If you have any experience with a SQL based database, the `id` property here should be familiar to you. In SQL databases, new records are identified uniquely using what is known as primary key. Imagine a school registration system where you store information on students. It is conceivable that two students could have the same name, for example “John Smith”. By storing a primary key like `id`, we can distinguish between one John and the next.

Let's add a new property to our meal class like `description` and an initializer. Our class should now look like this:

```
final class Meal: PostgreSQLModel {  
  var id: Int?  
  var description: String  
  
  init(description: String) {  
    self.description = description  
  }  
}
```

Finally, let's import Vapor at the top of this class and adopt a few more protocols to our class:

```
// Allows `Meal` to be used as a dynamic migration.  
extension Meal: Migration { }  
  
/// Allows `Meal` to be encoded to and decoded from HTTP  
messages.  
extension Meal: Content { }  
  
/// Allows `Meal` to be used as a dynamic parameter in  
route definitions.  
extension Meal: Parameter { }
```

Our final class should look like this:



In our database, we must have a table to match the model we just created. To do this we need to create a migration.

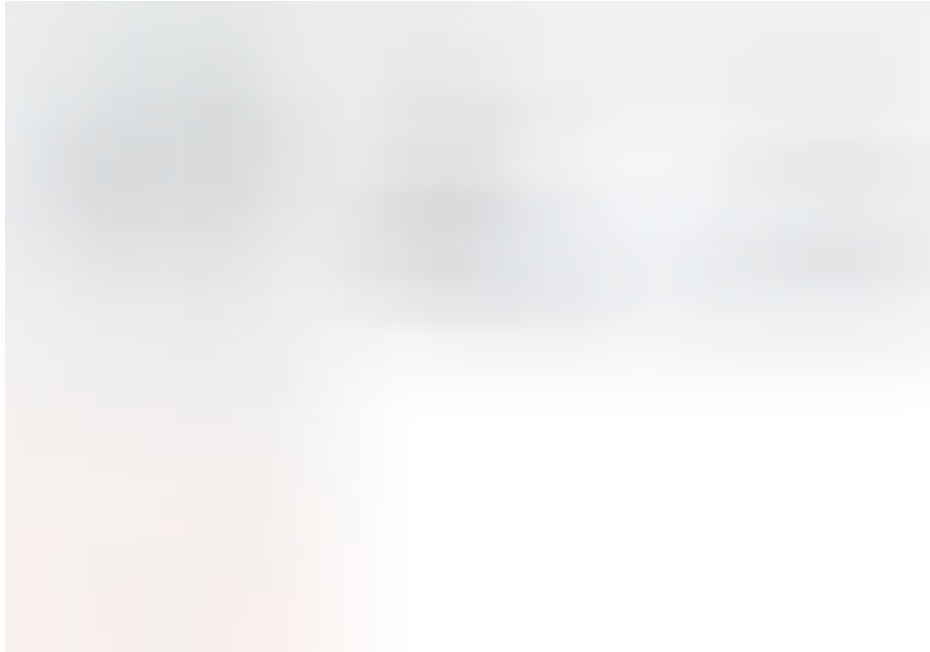
### Step 8—Create migration

Creating a migration with Fluent is simple. Navigate to the `configure.swift` file and after the creation of our `MigrationConfig` object, add the following line:

```
migrations.add(model: Meal.self, database: .psql)
```

Now let's run our project in Xcode.


In your terminal, we can see that our migrations have run, and our server is running. Now we want to be able to see the new table we have created. One of the easiest ways to do this is to download a PostgreSQL client like [Postico](#). This will allow us to directly connect to our database and use a GUI to interact with our tables.




Once we connect to our database, we should see the tables we created:



If we click into the Meal table, we can see an empty table with the property we created.



Let's go ahead and add a new record and save it.



Now, that our database and tables are working as expected we can create routes to allow us to interact with them.

### Step 9—Create a Meal Controller

Inside of our controller group, create a new file called `MealController.swift`.

Let's add the following code to our controller:

```
import Foundation
import Vapor

final class MealController {
    func index(_ req: Request) throws -> Future<[Meal]> {
        return Meal.query(on: req).all()
    }

    func create(_ req: Request) throws -> Future<Meal> {
        return try req.content.decode(Meal.self).flatMap {
            meal in
                return meal.save(on: req)
            }
    }
}
```

In the index method, we use one of Fluent's query methods to return all of the Meal objects stored in our database.

```
Meal.query(on: req).all()
```

Note that this method returns a `Future<Meal>`. Vapor utilizes the Future type to provide a reference to an object that isn't available yet.

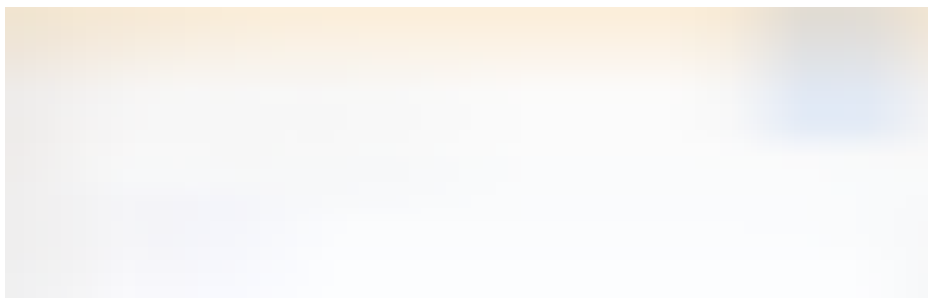
In the create method, we decode some JSON that is sent in the request to create a `Meal` object which we then save. Earlier in this tutorial, we adopted the `Content` protocol which gives us the ability to decode and encode Swift objects to JSON.

At the bottom of our routes method in the `routes.swift` file, add the following code:

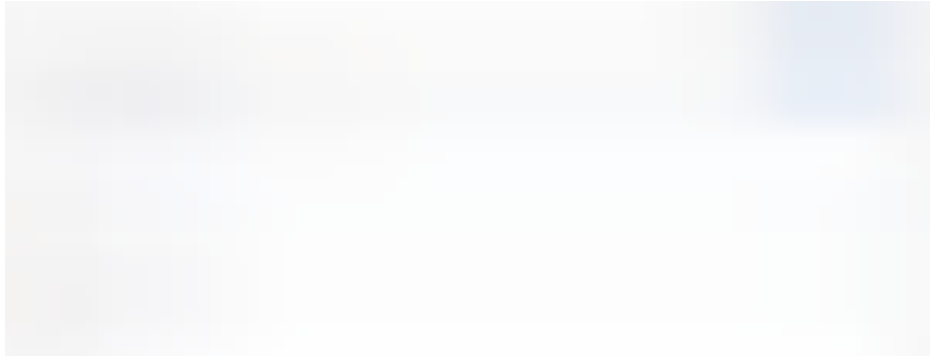
```
let mealController = MealController()
router.get("meals", use: mealController.index)
router.post("meals", use: mealController.create)
```

Let's run our program and test our routes. To test our routes, it's best to use a REST client like Postman. This will allow us to test GET and POST requests.

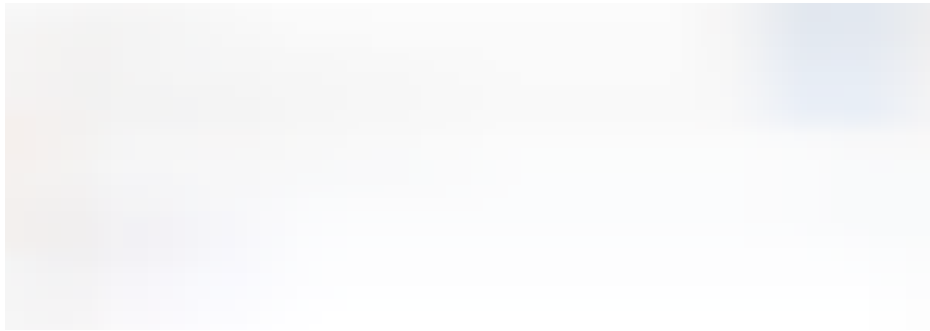
First, we'll try a GET request at `http://localhost:8004/meals`.



For the POST request, we will send some JSON data in the body of our request.



Note that if we retry our GET request we should now see two meal records.



Voila! We've now persisted data to our database and also retrieved that data back from the database. [Find the completed project here.](#)

. . .

To learn more about Flatiron School, visit the [website](#), follow us on [Facebook](#) and [Twitter](#), and visit us at [upcoming events](#) near you.

Flatiron School is a proud member of the [WeWork](#) family. Check out our sister technology blogs [WeWork Technology](#) and [Making Meetup](#).



