Build a Galaxy Backend

Create a Hummingbird + Fluent application.

Create a simple web application using the Hummingbird template.

15mins

Estimated Time

Section 1

Create your project

Clone the Hummingbird template, configure a project and review the contents of the generated project.

Clone template

No Preview

```
> git clone https://github.com/hummingbird-project/template
```

Step 1

Clone the Hummingbird template GitHub project

Step 2

Create your project, using the template configure script. Press return on each question to use the default value.

Step 3

Add the `hummingbird-fluent` and `fluent-sqlite-driver` dependencies.

Like with Vapor, you can use different Fluent Drivers as your backing storage.

Section 2

Add Fluent

With your Package.swift set up, lets add Fluent to your project.

Step 1

Open `Sources/App/Application+build.swift`.

Sources/App/Application+build.swift

```swift
import Hummingbird
import Logging
import FluentSQLiteDriver
import Foundation
import HummingbirdFluent

/// Application arguments protocol. We use a protocol so we can call
/// `buildApplication` inside Tests as well as in the App executable.
/// Any variables added here also have to be added to `App` in App.swift
```

Add the Fluent dependencies, and modify the `AppArguments` to contain two new variables.

Step 2

Open `Sources/App/App.swift`

This contains an App type conforming to `AsyncParsableCommand` with three options, the `hostname` and `port` are used to define the server bind address, `logLevel` sets the level of logging required. Finally the `run()` function which calls `buildApplication(_:)` to create an `Application` and then runs it using `runService()`. You can find out more about the argument parser library here.

Step 3

Add the new app arguments with default values.

Step 4

Open `Sources/App/Application+build.swift` again.

We can now add Fluent to our application's lifecycle.

Step 5

First, create a Fluent object and add the SQLite driver to Fluent.

Depending on the `inMemoryDatabase` boolean, this application can run completely in-memory. This is useful for testing, as it loses all data when the application is re-launched.

```swift
/// `TestArguments` in AppTest.swift
public protocol AppArguments {
    var inMemoryDatabase: Bool { get }
    var migrate: Bool { get }
    var hostname: String { get }
    var port: Int { get }
    var logLevel: Logger.Level? { get }
}

// Request context used by application
typealias AppRequestContext = BasicRequestContext

///  Build application
/// - Parameter arguments: application arguments
public func buildApplication(_ arguments: some AppArguments) async throw
    let environment = Environment()
    let logger = {
        var logger = Logger(label: "TodosFluent")
        logger.logLevel =
        arguments.logLevel ??
        environment.get("LOG_LEVEL").flatMap { Logger.Level(rawValue: $0
            .info
        return logger
    }()
    let router = buildRouter()
    let app = Application(
        router: router,
        configuration: .init(
            address: .hostname(arguments.hostname, port: arguments.port)
            serverName: "TodosFluent"
        ),
        logger: logger
    )
    return app
}

/// Build router
func buildRouter() -> Router<AppRequestContext> {
    let router = Router(context: AppRequestContext.self)
    // Add middleware
    router.addMiddleware {
        // logging middleware
        LogRequestsMiddleware(.info)
    }
    // Add default endpoint
```

Step 6

Next, we’ll use Fluent as a persistence mechanism for the Persist framework. This step is **optional** for this tutorial.

This allows it to integrate with Hummingbird’s ecosystem, including the Auth framework.

Step 7

Finally, both Fluent and the FluentPersistDriver are added to swift-service-lifecycle.

Section 3

Add Galaxy API

Add your database models and routes to edit them.

Step 1

Create a file named `Galaxy.swift`, and add the following Fluent Model.

This Fluent model has the ‘id’ and a ‘name’ properties.

Sources/App/Galaxy.swift

No Preview

```swift
import FluentKit
import Foundation
import Hummingbird

final class Galaxy: Model, @unchecked Sendable, ResponseCodable {
    // Name of the table or collection.
    static let schema = "galaxies"

    // Unique identifier for this Galaxy.
    @ID(key: .id)
    var id: UUID?

    // The Galaxy's name.
    @Field(key: "name")
    var name: String

    // Creates a new, empty Galaxy.
    init() { }

    // Creates a new Galaxy with all properties set.
    init(id: UUID? = nil, name: String) {
        self.id = id
        self.name = name
    }
}
```

Step 2

Before being able to use a Model, a migration must be added.

A migration creates or reverts a diff to the schema in the database.

Step 3

Open `Sources/App/Application+build.swift` again. Fluent is now a completely blank slate, let's set it up.

The newly added migrations are added to Fluent. Make sure that any new migrations are added to Fluent in the right order.

Step 4

The final step to set up Fluent is to run the migrations.

It's common to explicitly run migrations, but for small scale set-ups can also run migrations on every app launch.

Step 5

If we look further down the file we can find the `buildRouter()` function.

Here we create the `Router`. We add a logging middleware to it (this logs all requests to the router). The function uses a result builder to create a stack of middleware, but you can also use `Router.add(middleware:)` to add individual middleware. Finally we add a single endpoint GET `/` which returns "Hello!"

Step 6

We’ll add a single route `GET /galaxies`, which lists all registered galaxies.

Because the database is empty now, we’ll add a route `PUT /galaxies` to add your own galaxies.

Section 4

Test your Backend

Now that your Fluent backend is complete, it’s time to validate the results!

Test Application

No Preview

```
> curl -i -X PUT -H "Content-Type: application/json" -d '{"name":"Androm
HTTP/1.1 201 Created
Content-Length: 0
Date: Sat, 23 Nov 2024 09:22:26 GMT
Server: TodosFluent
```

Step 1

We can run the application and use curl to test it works.

First, create your own galaxy!

Step 2

Then, query the list of galaxies.

Step 3

You can see the galaxy added in the first call, is returned when we ask to list all the galaxies.