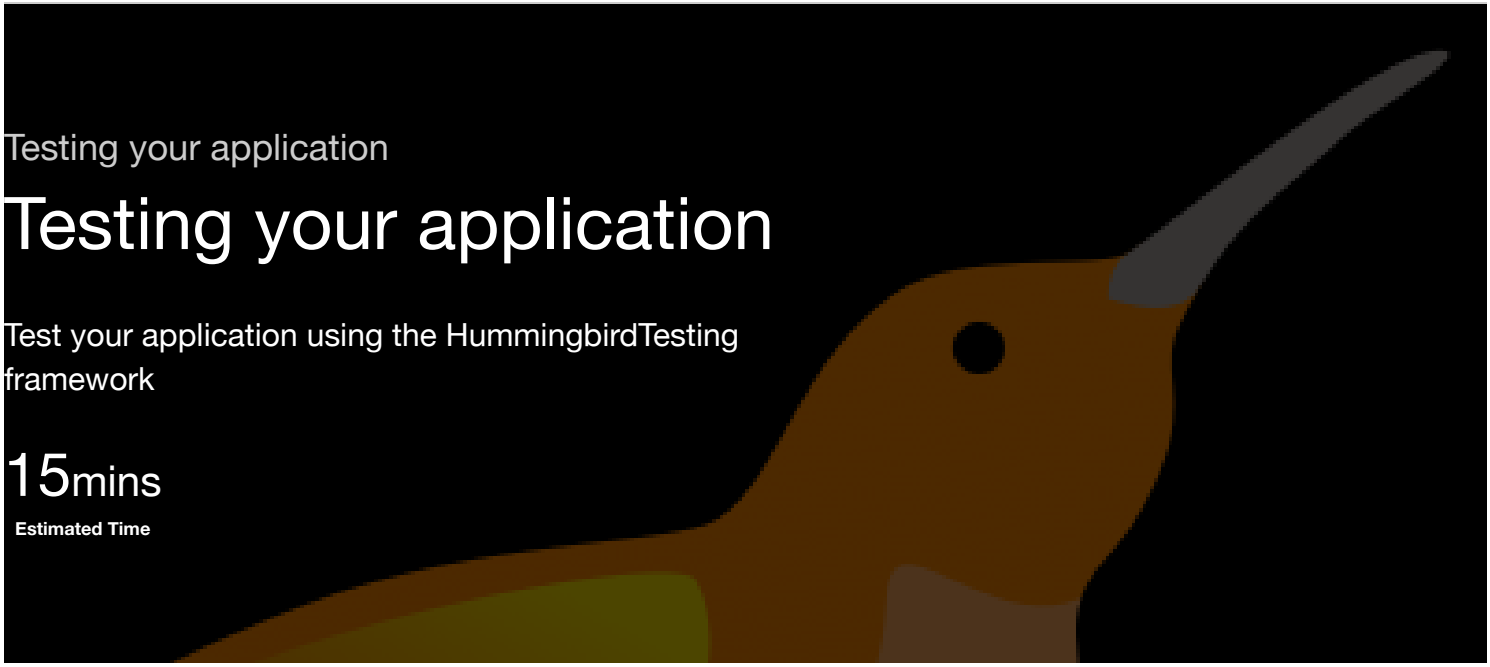

Testing your application

Testing your application

Test your application using the HummingbirdTesting framework

15mins

Estimated Time

Section 1

Project testing setup

Review application testing setup from template.

Step 1

Open Package.swift

Package.swift

No Preview

```
// swift-tools-version:5.9
// The swift-tools-version declares the minimum version of Swift require
import PackageDescription

let package = Package(
    name: "Todos",
    platforms: [.macOS(.v14), .iOS(.v17), .tvOS(.v17)],
    products: [
```

You see at the bottom there is a test target called `AppTests`. It is dependent on the target `App` and the library `HummingbirdTesting`.

Step 2

Open Tests/AppTests/AppTests.swift

It contains one test, `testApp()`. This creates a copy of the Application using `buildApplication(_:)` and uses the Hummingbird test framework to verify the GET `/` endpoint returns a “Hello!” string.

Step 3

We cannot create an instance of `App`, so need another way of passing the arguments to the `buildApplication` function in our tests. So `buildApplication(_:)` doesn’t take `App` as a parameter. Instead its parameter is a type that conforms to the protocol `AppArguments` which includes the parameters the function needs. We then conform `App` to `AppArguments` and in our tests create a new type `TestArguments` which conforms to the protocol `AppArguments`.

```
        .executable(name: "App", targets: ["App"]),
    ],
    dependencies: [
        .package(url: "https://github.com/hummingbird-project/hummingbir
        .package(url: "https://github.com/apple/swift-argument-parser.gi
    ],
    targets: [
        .executableTarget(
            name: "App",
            dependencies: [
                .product(name: "ArgumentParser", package: "swift-argumen
                .product(name: "Hummingbird", package: "hummingbird"),
            ]
        ),
        .testTarget(
            name: "AppTests",
            dependencies: [
                .byName(name: "App"),
                .product(name: "HummingbirdTesting", package: "hummingbi
            ]
        ),
    ]
)
```

Section 2

Test your application

Writing Tests to ensure you application API works.

Tests/AppTests/AppTests.swift

No Preview

```swift
import Foundation
import Hummingbird
import HummingbirdTesting
import Logging
import XCTest

@testable import App

final class AppTests: XCTestCase {
    struct TestArguments: AppArguments {
        let hostname = "127.0.0.1"
        let port = 0
        let logLevel: Logger.Level? = .trace
    }

    func testCreate() async throws {
        let app = try await buildApplication(TestArguments())
        try await app.test(.router) { client in
            try await client.execute(uri: "/todos", method: .post, body:
                XCTAssertEqual(response.status, .created)
                let todo = try JSONDecoder().decode(Todo.self, from: res
                XCTAssertEqual(todo.title, "My first todo")
            }
        }
    }
}
```

Step 1

Lets replace the `testApp` function with a test for the create todo function. Application testing is done with the function `test( : :)`. The first parameter indicates what test framework you want to use. Here we are using `.router` which sends our request directly to the router without a live server process.

In the closure passed to `test` you are provided with a client to interact with the current test framework. With this you can send requests and verify the contents of their responses.

Step 2

Writing the whole execute line out each time and converting the responses to something readable can become tiresome. So lets break out the create API call to a separate function. You'll notice in this function we return the decoded `Todo` from the execute closure.

Now the create test has been simplified to two lines of code. Call create function, test return value.

Step 3

In actual fact lets create helper functions for all the API calls. With these it shoud be a lot easier to write tests

Step 4

We can now create more complex test functions. This one edits a todo twice and verifies the edits have been stored.

Step 5

The following is the equivalent of the list of curl commands we wrote in the previous chapter to test everything was working ok. Its not the most sensible test but it demonstrates how much easier it is test your application using HummingbirdTesting.

Step 6

Here are some tests that haven't been written yet. Maybe you could complete them for me.

Hint: A couple of these require you to use execute directly instead of calling the helper functions we wrote at the top.

Next

Use PostgresNIO to store your Todos in a Postgres database

Now we have a working API and a way to test it, lets look into storing our todos in a Postgres database with PostgresNIO.

Get started