Chapter 11: Testing

By Tim Condon

测试是软件开发过程的重要部分。编写单元测试并尽可能自动化它们可以让您快速开发和发展您的应用程序。

在本章中，您将学习如何为Vapor应用程序编写测试。您将了解为什么测试很重要以及它如何与Swift Package Manager一起使用。然后，您将学习如何为前面章节中的TIL应用程序编写测试。最后，您将了解为什么在Linux上进行测试以及如何使用Docker在Linux上测试代码。

为什么要写测试?

软件测试与软件开发本身一样古老。现代服务器应用程序每天都要部署多次，因此确保一切都按预期工作非常重要。为您的应用程序编写测试可以让您确信代码是健全的。

在重构代码时，测试还可以让您放心。在过去的几章中，您已经发展并更改了TIL应用程序。手动测试应用程序的每个部分是缓慢而费力的，而且这个应用程序很小！要快速开发新功能，您需要确保现有功能不会中断。通过一组广泛的测试，您可以在更改代码时验证所有内容是否仍然有效。

测试还可以帮助您设计代码。测试驱动开发是一种流行的开发过程，您可以在编写代码之前编写测试。这有助于确保您拥有代码的完整测试覆盖率。测试驱动开发还可以帮助您设计代码和API。

使用SPM编写测试

在iOS上，Xcode将测试链接到特定的测试目标。 Xcode配置一个方案来使用该目标，并在Xcode中运行测试。 Objective-C运行时扫描您的XCTestCases并选择名称以test开头的方法。在Linux上，使用SPM，没有Objective-C运行时。也没有Xcode项目可以记住方案和哪些测试属于哪里。

在项目中打开**Package.swift**。targets数组中定义了一个测试目标：

```
.testTarget(name: "AppTests", dependencies: ["App"]),
```

这定义了一个依赖于App的testTarget类型。测试必须存在于**Tests/**目录中。在这种情况下，这是**Tests/AppTests**。

使用**vapor xcode -y**生成Xcode项目并打开它。如果选择**TILApp-Package**方案，则会将AppTests设置为测试目标。您可以使用**Command-U**或**Product▸Test**正常运行这些测试：

Testing users

编写第一个测试

在Xcode中关闭您的项目，然后在终端中，为用户相关的测试创建一个文件：

```
touch Tests/AppTests/UserTests.swift
vapor xcode -y
```

这会将文件添加到目录层次结构中的正确位置，并重新生成Xcode项目以确保正确构建新文件。在Xcode中，打开**UserTests.swift**并添加以下内容：

```
@testable import App
import Vapor
import XCTest
import FluentPostgreSQL

final class UserTests: XCTestCase {

}
```

这将创建您用于测试用户的XCTestCase，并导入必要的模块以使一切正常。

接下来，在UserTests中添加以下内容以测试从API获取用户：

```
  func testUsersCanBeRetrievedFromAPI() throws {
    // 1
    let expectedName = "Alice"
    let expectedUsername = "alice"

    // 2
    var config = Config.default()
    var services = Services.default()
    var env = Environment.testing
    try App.configure(&config, &env, &services)
    let app = try Application(
      config: config,
      environment: env,
      services: services)
    try App.boot(app)

    // 3
    let conn = try app.newConnection(to: .psql).wait()

    // 4
    let user = User(
      name: expectedName,
      username: expectedUsername)
    let savedUser = try user.save(on: conn).wait()
    _ = try User(
      name: "Luke",
      username: "lukes").save(on: conn).wait()

    // 5
    let responder = try app.make(Responder.self)

    // 6
    let request = HTTPRequest(
      method: .GET,
      url: URL(string: "/api/users")!)
    let wrappedRequest = Request(http: request, using: app)

    // 7
    let response = try responder
      .respond(to: wrappedRequest)
      .wait()

    // 8
    let data = response.http.body.data
    let users = try JSONDecoder().decode([User].self, from: data!)

    // 9
    XCTAssertEqual(users.count, 2)
    XCTAssertEqual(users[0].name, expectedName)
    XCTAssertEqual(users[0].username, expectedUsername)
    XCTAssertEqual(users[0].id, savedUser.id)
```

```
    // 10
    conn.close()
  }
```

这个测试有很多事情要做；这是分析：

1. 为测试定义一些预期值：用户的name和username。

2. 创建一个类似于**App.swift**的应用程序。这将创建一个完整的Application对象，但不会开始运行该应用程序。这有助于确保在测试调用相同的App.configure(_:_:_:)时正确配置实际应用程序。注意，您在这里使用.testing环境。

3. 创建数据库连接以执行数据库操作。注意在这里和整个测试中使用.wait()。由于您没有在EventLoop上运行测试，因此可以使用wait()等待future返回。这有助于简化代码。

4. 创建几个用户并将其保存在数据库中。

5. 创建一个Responder类型；这是您请求的响应内容。

6. 向**/api/users**发送GET HTTPRequest，这是获取所有用户的端点。 Request对象包装了HTTPRequest，因此有一个Worker来执行它。由于这是一个测试，您可以强制解包变量以简化代码。

7. 发送请求并获得响应。

8. 将响应数据解码为User数组。

9. 确保响应中有正确数量的用户，并且用户与测试开始时创建的用户匹配。

10. 测试完成后关闭与数据库的连接，并停止应用程序以正确释放已用资源。

接下来，您必须更新应用程序的配置以支持测试。打开**configure.swift**及在var databases = DatabasesConfig()下面添加以下内容：

```
  let databaseName: String
  let databasePort: Int
  // 1
  if (env == .testing) {
    databaseName = "vapor-test"
    databasePort = 5433
  } else {
    databaseName = "vapor"
    databasePort = 5432
  }
```

这将根据环境设置数据库名称和端口的属性。您将使用不同的名称和端口来测试和运行应用程序。接下来，使用以下命令替换对PostgreSQLDatabaseConfig的调用：

```
let databaseConfig = PostgreSQLDatabaseConfig(
  hostname: "localhost",
  port: databasePort,
  username: "vapor",
  database: databaseName,
  password: "password")
```

这将用上面设置的属性来设置数据库端口和名称。这些更改允许您在除了生产数据库以外的数据库上运行测试。这可确保您以已知状态启动每个测试，并且不会破坏实时数据。由于您使用Docker来托管数据库，因此在同一台计算机上设置另一个数据库非常简单。在终端中，键入以下内容：

```
docker run --name postgres-test -e POSTGRES_DB=vapor-test \
  -e POSTGRES_USER=vapor -e POSTGRES_PASSWORD=password \
  -p 5433:5432 -d postgres
```

这类似于您在第6章“Configuring a Database”中使用的命令，但它更改了容器名称和数据库名称。 Docker容器也映射到主机端口5433，以避免与现有数据库冲突。

运行测试，他们应该通过。但是，如果再次运行测试，它们将失败。第一次测试运行将两个用户添加到数据库，第二次测试运行现在有四个用户了，因为数据库未重置。

打开**configure.swift**并将以下内容添加到configure(_:_:_:)的底部：

```
var commandConfig = CommandConfig.default()
commandConfig.useFluentCommands()
services.register(commandConfig)
```

这会将Fluent命令添加到您的应用程序，从而允许您手动运行migrations。它还允许您还原migrations。打开**UserTests.swift**，并在testUsersCanBeRetrievedFromAPI()的开头添加以下内容：

```
// 1
let revertEnvironmentArgs = ["vapor", "revert", "--all", "-y"]
// 2
var revertConfig = Config.default()
var revertServices = Services.default()
var revertEnv = Environment.testing
// 3
revertEnv.arguments = revertEnvironmentArgs
// 4
try App.configure(&revertConfig, &revertEnv, &revertServices)
```

```
let revertApp = try Application(
  config: revertConfig,
  environment: revertEnv,
  services: revertServices)
try App.boot(revertApp)
// 5
try revertApp.asyncRun().wait()

// 6
let migrateEnvironmentArgs = ["vapor", "migrate", "-y"]
var migrateConfig = Config.default()
var migrateServices = Services.default()
var migrateEnv = Environment.testing
migrateEnv.arguments = migrateEnvironmentArgs
try App.configure(&migrateConfig, &migrateEnv, &migrateServices)
let migrateApp = try Application(
  config: migrateConfig,
  environment: migrateEnv,
  services: migrateServices)
try App.boot(migrateApp)
try migrateApp.asyncRun().wait()
```

这是它的作用：

1. 设置Application应该执行的参数。

2. 设置服务，配置和测试环境。

3. 设置测试环境的参数。

4. 像稍早前的测试里一样设置应用程序。这将创建一个执行revert命令的不同Application对象。

5. 调用asyncRun()启动应用程序并执行revert命令。

6. 再次重复此过程以运行migrations。这会在单独的连接上建立数据库，类似于Vapor的工作方式。

再次构建并运行测试，这次它们将通过！

Test extensions

第一个测试包含许多所有测试都需要的代码。提取公共部分以使测试更易于阅读并简化将来的测试。在Xcode中关闭项目，然后在终端中为这些扩展创建两个新文件：

```
touch Tests/AppTests/Application+Testable.swift
touch Tests/AppTests/Models+Testable.swift
vapor xcode -y
```

项目重新生成后，打开**Application+Testable.swift**并添加以下内容：

```
import Vapor
import App
import FluentPostgreSQL

extension Application {
  static func testable(envArgs: [String]? = nil) throws
    -> Application {
    var config = Config.default()
    var services = Services.default()
    var env = Environment.testing

    if let environmentArgs = envArgs {
      env.arguments = environmentArgs
    }

    try App.configure(&config, &env, &services)
    let app = try Application(
      config: config,
      environment: env,
      services: services)

    try App.boot(app)
    return app
  }
}
```

此函数允许您创建可测试的Application对象。如果需要，您可以指定环境参数。这将删除测试中的几行重复代码。在testable(envArgs:)下面添加以下函数来重置数据库：

```
static func reset() throws {
  let revertEnvironment = ["vapor", "revert", "--all", "-y"]
  try Application.testable(envArgs: revertEnvironment)
    .asyncRun()
    .wait()
  let migrateEnvironment = ["vapor", "migrate", "-y"]
  try Application.testable(envArgs: migrateEnvironment)
    .asyncRun()
    .wait()
}
```

这使用上述函数创建一个应用程序，它先运行revert命令，然后运行migrate命令。这简化了在每个测试中重置数据库的过程。接下来，在文件底部添加以下内容：

```
struct EmptyContent: Content {}
```

这定义了在没有body的请求发送时要使用的空的Content类型。由于您无法为泛型类型定义nil，因此EmptyContent允许您提供满足编译器的类型。接下来，在reset()下面，添加以下内容：

```
// 1
func sendRequest<T>(
  to path: String,
  method: HTTPMethod,
  headers: HTTPHeaders = .init(),
  body: T? = nil
) throws -> Response where T: Content {
  let responder = try self.make(Responder.self)
  // 2
  let request = HTTPRequest(
    method: method,
    url: URL(string: path)!,
    headers: headers)
  let wrappedRequest = Request(http: request, using: self)
  // 3
  if let body = body {
    try wrappedRequest.content.encode(body)
  }
  // 4
  return try responder.respond(to: wrappedRequest).wait()
}

// 5
func sendRequest(
  to path: String,
  method: HTTPMethod,
  headers: HTTPHeaders = .init()
) throws -> Response {
  // 6
  let emptyContent: EmptyContent? = nil
  // 7
  return try sendRequest(
    to: path,
    method: method,
    headers: headers,
    body: emptyContent)
}

// 8
func sendRequest<T>(
  to path: String,
  method: HTTPMethod,
  headers: HTTPHeaders,
  data: T
) throws where T: Content {
  // 9
  _ = try self.sendRequest(
    to: path,
    method: method,
    headers: headers,
    body: data)
}
```

这是代码的作用：

1. 定义一个向路径发送请求并返回Response的方法。允许设置HTTP方法和标头；这是为了以后的测试。还允许为请求body提供可选的泛型Content。

2. 像以前一样创建响应者、请求和包装的请求。

3. 如果测试包含body，请将body编码为请求的内容。使用Vapor的encode(_:)可以利用您设置的任何自定义编码器。

4. 发送请求并返回响应。

5. 定义一个方便的方法，将没有body的请求发送到路径。

6. 创建一个EmptyContent的body参数以满足编译器。

7. 使用之前创建的方法发送请求。

8. 定义一个向路径发送请求并接受泛型Content类型的方法。这种便捷方法允许您在不关心响应时发送请求。

9. 使用上面创建的第一个方法发送请求并忽略响应。

接下来，在这些帮助程序下，添加以下方法以获取请求的响应：

```
// 1
func getResponse<C, T>(
  to path: String,
  method: HTTPMethod = .GET,
  headers: HTTPHeaders = .init(),
  data: C? = nil,
  decodeTo type: T.Type
) throws -> T where C: Content, T: Decodable {
  // 2
  let response = try self.sendRequest(
    to: path,
    method: method,
    headers: headers,
    body: data)
    // 3
    return try response.content.decode(type).wait()
  }

// 4
func getResponse<T>(
  to path: String,
  method: HTTPMethod = .GET,
  headers: HTTPHeaders = .init(),
  decodeTo type: T.Type
```

```
  ) throws -> T where T: Decodable {
    // 5
    let emptyContent: EmptyContent? = nil
    // 6
    return try self.getResponse(
      to: path,
      method: method,
      headers: headers,
      data: emptyContent,
      decodeTo: type)
  }
```

这是发生了什么：

1. 定义一个接受Content类型和Decodable类型的泛型方法来获取对请求的响应。
2. 使用上面创建的方法发送请求。
3. 将响应body解码为泛型类型并返回结果。
4. 定义一个便捷的泛型方法，它接受一个Decodable类型来获取对没有提供body的请求的响应。
5. 创建一个空的Content以满足编译器。
6. 使用之前的方法获取对请求的响应。

接下来，打开**Models+Testable.swift**并创建一个扩展来创建User：

```
  @testable import App
  import FluentPostgreSQL

  extension User {
    static func create(
      name: String = "Luke",
      username: String = "lukes",
      on connection: PostgreSQLConnection
    ) throws -> User {
      let user = User(name: name, username: username)
      return try user.save(on: connection).wait()
    }
  }
```

此方法将把由详细信息创建的用户保存在数据库中。它具有默认值，因此如果您不关心它们，则无需提供任何值。

创建所有这些后，您现在可以重写用户测试。打开**UserTests.swift**并删除testUsersCanBeRetrievedFromAPI()。

接下来，在UserTests中为所有测试创建公共属性：

```
let usersName = "Alice"
let usersUsername = "alicea"
let usersURI = "/api/users/"
var app: Application!
var conn: PostgreSQLConnection!
```

接着实现setUp()来运行必须在每次测试之前执行的代码：

```
override func setUp() {
  try! Application.reset()
  app = try! Application.testable()
  conn = try! app.newConnection(to: .psql).wait()
}
```

这将还原数据库，生成测试应用程序，并创建与数据库的连接。

接着，实现teardown()以关闭与数据库的连接，并关闭应用程序：

```
override func tearDown() {
  conn.close()
  try? app.syncShutdownGracefully()
}
```

最后，重写testUsersCanBeRetrievedFromAPI()以使用所有新的辅助方法：

```
func testUsersCanBeRetrievedFromAPI() throws {
  let user = try User.create(
    name: usersName,
    username: usersUsername,
    on: conn)
  _ = try User.create(on: conn)

  let users = try app.getResponse(
    to: usersURI,
    decodeTo: [User].self)

  XCTAssertEqual(users.count, 2)
  XCTAssertEqual(users[0].name, usersName)
  XCTAssertEqual(users[0].username, usersUsername)
  XCTAssertEqual(users[0].id, user.id)
}
```

此测试与以前完全相同，但更具可读性。它还使下一个测试更容易编写。再次运行测试以确保它们仍然有效。

Testing the User API

打开**UserTests.swift**并使用测试辅助方法添加以下内容，以测试通过API保存用户：

```
func testUserCanBeSavedWithAPI() throws {
  // 1
  let user = User(name: usersName, username: usersUsername)
  // 2
  let receivedUser = try app.getResponse(
    to: usersURI,
    method: .POST,
    headers: ["Content-Type": "application/json"],
    data: user,
    decodeTo: User.self)

  // 3
  XCTAssertEqual(receivedUser.name, usersName)
  XCTAssertEqual(receivedUser.username, usersUsername)
  XCTAssertNotNil(receivedUser.id)

  // 4
  let users = try app.getResponse(
    to: usersURI,
    decodeTo: [User].self)

  // 5
  XCTAssertEqual(users.count, 1)
  XCTAssertEqual(users[0].name, usersName)
  XCTAssertEqual(users[0].username, usersUsername)
  XCTAssertEqual(users[0].id, receivedUser.id)
}
```

这是它的作用：

1. 创建具有已知值的User对象。

2. 使用getResponse(to:method:headers:data:decodeTo:)向API发送POST请求并获取响应。使用用户对象作为请求body并正确设置标头以模拟JSON请求。将响应转换为User对象。

3. 判断API的响应与预期值是否匹配。

4. 从API获取所有用户。

5. 确保响应仅包含您在第一个请求中创建的用户。

运行测试以确保新测试工作正常！

接下来，添加以下测试以从API检索单个用户：

```
func testGettingASingleUserFromTheAPI() throws {
  // 1
  let user = try User.create(
    name: usersName,
    username: usersUsername,
    on: conn)
  // 2
  let receivedUser = try app.getResponse(
    to: "\(usersURI)\(user.id!)",
    decodeTo: User.self)

  // 3
  XCTAssertEqual(receivedUser.name, usersName)
  XCTAssertEqual(receivedUser.username, usersUsername)
  XCTAssertEqual(receivedUser.id, user.id)
}
```

这是测试的作用：

1. 使用已知值将用户保存在数据库中。
2. 获取**/api/users/<USER ID>**的用户。
3. 判断值与创建用户时提供的值是否相同。

用户测试API的最后一部分检索用户的缩略词。打开**Models+Testable.swift**，并在文件末尾创建一个新的扩展来创建缩略词：

```
extension Acronym {
  static func create(
    short: String = "TIL",
    long: String = "Today I Learned",
    user: User? = nil,
    on connection: PostgreSQLConnection
  ) throws -> Acronym {
    var acronymsUser = user

    if acronymsUser == nil {
      acronymsUser = try User.create(on: connection)
    }

    let acronym = Acronym(
      short: short,
      long: long,
      userID: acronymsUser!.id!)
    return try acronym.save(on: connection).wait()
  }
}
```

这将创建一个缩略词，并使用提供的值将其保存在数据库中。如果您不提供任何值，则使用默认值。如果您没有为缩略词提供用户，则会先创建一个用户以使用。

接下来，打开**UserTests.swift**并创建一个方法来测试获取用户的缩略词：

```
func testGettingAUsersAcronymsFromTheAPI() throws {
  // 1
  let user = try User.create(on: conn)
  // 2
  let acronymShort = "OMG"
  let acronymLong = "Oh My God"
  // 3
  let acronym1 = try Acronym.create(
    short: acronymShort,
    long: acronymLong,
    user: user,
    on: conn)
  _ = try Acronym.create(
    short: "LOL",
    long: "Laugh Out Loud",
    user: user,
    on: conn)

  // 4
  let acronyms = try app.getResponse(
    to: "\(usersURI)\(user.id!)/acronyms",
    decodeTo: [Acronym].self)

  // 5
  XCTAssertEqual(acronyms.count, 2)
  XCTAssertEqual(acronyms[0].id, acronym1.id)
  XCTAssertEqual(acronyms[0].short, acronymShort)
  XCTAssertEqual(acronyms[0].long, acronymLong)
}
```

这是测试的作用：

1. 为缩略词创建用户。
2. 为缩略词定义一些预期值。
3. 使用创建的用户在数据库中创建两个缩略词。为第一个缩略词使用预期值。
4. 通过向**/api/users/<USER ID>/acronyms**发送请求，从API获取用户的缩略词。
5. 断言判断响应是否返回正确的缩略词数，第一个缩略词是否匹配预期值。

运行测试以确保更改有效！

Testing acronyms and categories

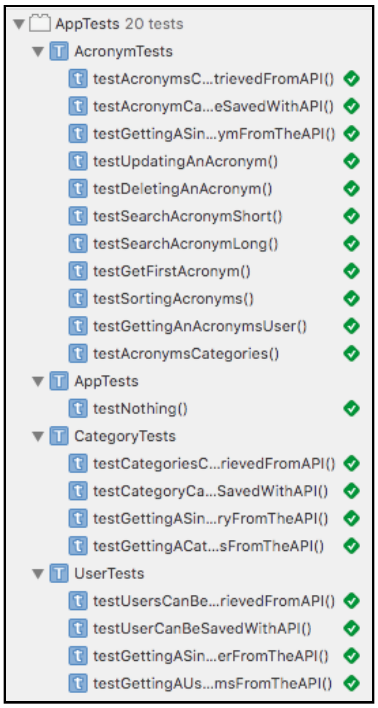
打开**Models+Testable.swift**，并在文件的底部添加一个新扩展以简化创建类别：

```
extension App.Category {
  static func create(
    name: String = "Random",
    on connection: PostgreSQLConnection
  ) throws -> App.Category {
    let category = Category(name: name)
    return try category.save(on: connection).wait()
  }
}
```

与其他模型辅助函数一样，create(name:on:)将名称作为参数并在数据库中创建一个类别。缩略词API和类别API的测试是本章starter项目的一部分。打开**CategoryTests.swift**并取消注释所有代码。这些测试遵循与用户测试相同的模式。

打开**AcronymTests.swift**并取消注释所有代码。这些测试也遵循与之前类似的模式，但是对于缩略词API中的额外路由有一些额外的测试。这些包括更新缩略词，删除缩略词和不同的Fluent查询路由。

运行所有测试以确保它们都能正常工作。每一条测试路由都应该进行大量的绿色测试！



Testing on Linux

在本章的前面部分，您了解了为什么测试应用程序非常重要。对于服务器端Swift，在Linux上进行测试尤为重要。例如，当您将应用程序部署到Vapor Cloud时，您将部署到与用于开发的操作系统不同的操作系统。在部署它的同一环境中测试应用程序至关重要。

为什么会这样？ Linux上的Foundation与macOS上的Foundation不同。在撰写本文时，基于macOS的Foundation仍然使用Objective-C框架，该框架已经过多年的全面测试。Linux使用的是纯Swift Foundation框架，它不够强大。 implementation status list 显示许多功能在Linux上仍未实现。如果您使用这些功能，您的应用程序可能会崩溃。虽然情况不断改善，但您仍必须确保一切按预期在Linux上运行。

Declaring tests on Linux

在Linux上运行测试与您在macOS上运行它们所需要做的事情是不同的。如前所述，Objective-C运行时确定了XCTestCases提供的测试方法。

在Linux上没有运行时来执行此操作，因此您必须将Swift指向正确的方向。

在Linux上，您在**Tests**目录中的**LinuxMain.swift**中声明测试用例。此文件不是Xcode项目的一部分。您可以通过在Finder中双击它在Xcode中打开它，也可以使用其他文本编辑器对其进行编辑。用以下内容替换**LinuxMain.swift**的内容：

```
import XCTest
// 1
@testable import AppTests

// 2
XCTMain([
  testCase(AcronymTests.allTests),
  testCase(CategoryTests.allTests),
  testCase(UserTests.allTests)
])
```

此文件是**main.swift**的测试等效项。这是它的作用：

1. 导入包含测试的AppTests模块。

2. 为每个XCTestCase提供一系列测试到XCTMain(_:)。这些是在Linux上测试应用程序时执行的。

您必须为每个XCTestCase提供一个数组。按照惯例，您将此数组称为allTests。它包含一个元组列表，包括测试名称和测试本身。目前，您必须自己创建和维护它。

AcronymTests和CategoryTests已包含allTests数组。在Xcode中打开**UserTests.swift**，并在UserTests的底部添加数组：

```
static let allTests = [
  ("testUsersCanBeRetrievedFromAPI",
   testUsersCanBeRetrievedFromAPI),
  ("testUserCanBeSavedWithAPI", testUserCanBeSavedWithAPI),
  ("testGettingASingleUserFromTheAPI",
   testGettingASingleUserFromTheAPI),
  ("testGettingAUsersAcronymsFromTheAPI",
   testGettingAUsersAcronymsFromTheAPI)
]
```

当您在Linux上调用swift test或vapor test时，测试可执行文件使用此数组来确定要运行的测试。

Running tests in Linux

早期反馈在软件开发中始终很有价值，Linux上的运行测试也不例外。使用Continuous Integration系统在Linux上自动测试至关重要，但是如果要在Mac上的Linux上进行测试会发生什么？

好吧，你已经使用Docker在Linux上运行PostgreSQL数据库了！因此，您也可以使用Docker在Linux环境中运行测试。在项目目录中，创建一个名为**Dockerfile**的新文件（没有扩展名）。在文本编辑器中打开文件并添加以下内容：

```
# 1
FROM swift:4.2

# 2
WORKDIR /package
# 3
COPY . ./
# 4
RUN swift package resolve
RUN swift package clean
# 5
CMD ["swift", "test"]
```

这是**Dockerfile**的作用：

1. 使用Swift 4.2镜像.
2. 将工作目录设置为/**package**。

3. 将当前目录的内容复制到容器中的**/package**中。
4. 获取依赖项并清理项目的构建。
5. 将默认命令设置为swift test。这是Docker在运行Dockerfile时执行的命令。

测试需要一个PostgreSQL数据库才能运行。默认情况下，Docker容器无法看到对方。但是，Docker有一个工具Docker Compose，旨在将不同的容器链接在一起，以便测试和运行应用程序。在项目目录中创建一个名为**docker-compose.yml**的新文件。在编辑器中打开文件并添加以下内容：

```
# 1
version: '3'
# 2
services:
  # 3
  til-app:
    # 4
    depends_on:
      - postgres
    # 5
    build: .
    # 6
    environment:
      - DATABASE_HOSTNAME=postgres
      - DATABASE_PORT=5432
  # 7
  postgres:
    # 8
    image: "postgres"
    # 9
    environment:
      - POSTGRES_DB=vapor-test
      - POSTGRES_USER=vapor
      - POSTGRES_PASSWORD=password
```

这是它的作用：

1. 指定Docker Compose版本。
2. 定义此应用程序的服务。
3. 为TIL应用程序定义服务。
4. 设置Postgres容器的依赖关系，因此Docker Compose首先启动Postgres容器。
5. 在当前目录中构建Dockerfile - 您之前创建的Dockerfile。

6. 注入DATABASE_HOSTNAME环境变量。 Docker Compose有一个内部DNS解析器。这允许til-app容器使用主机名postgres连接到postgres容器。还要设置数据库的端口。

7. 为Postgres容器定义服务。

8. 使用标准的Postgres镜像。

9. 设置环境变量，与章节开头时测试数据库使用的相同。

最后在Xcode中打开**configure.swift**，并允许将数据库端口设置为用于测试的环境变量。在if:(env == .testing)里用以下内容替换行databasePort = 5433：

```
if let testPort = Environment.get("DATABASE_PORT") {
  databasePort = Int(testPort) ?? 5433
} else {
  databasePort = 5433
}
```

如果设置了环境变量，则使用DATABASE_PORT环境变量，否则将端口默认为5433.接下来，在let databaseName: String上面添加以下内容：

```
let hostname = Environment.get("DATABASE_HOSTNAME") ?? "localhost"
```

这将使用DATABASE_HOSTNAME环境变量（如果存在）创建属性。如果没有，则使用localhost。接下来，在PostgreSQLDatabaseConfig初始化程序调用中使用新属性替换hostname: "localhost"：

```
hostname: hostname,
```

这些允许您使用**docker-compose.yml**中设置的主机名和端口。要在Linux中测试您的应用程序，请打开终端并键入以下内容：

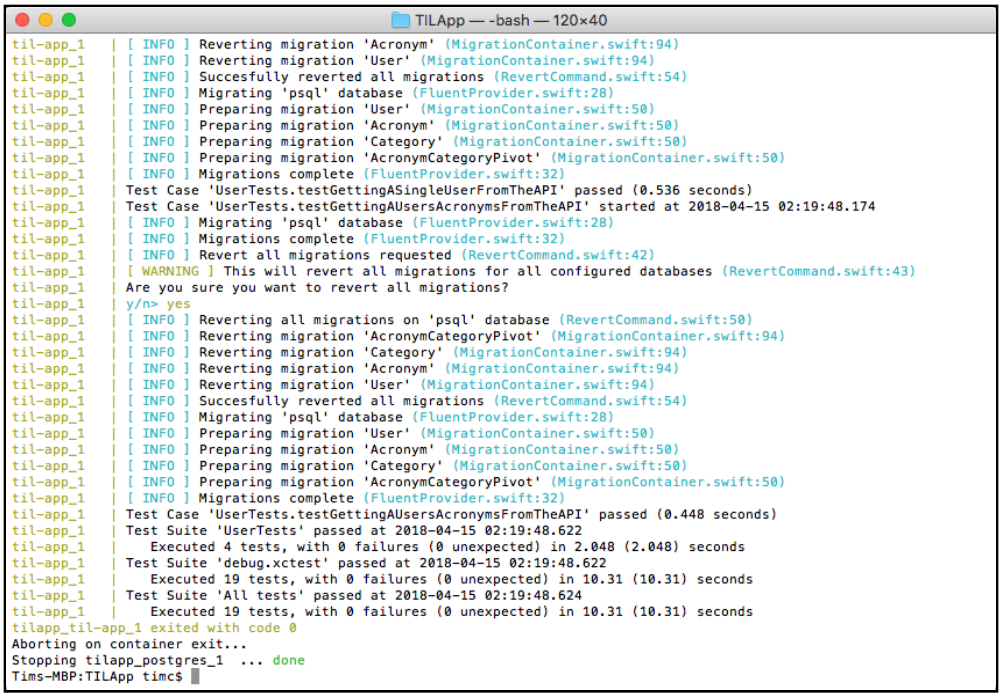
```
# 1
docker-compose build
# 2
docker-compose up --abort-on-container-exit
```

这是它的作用：

1. 构建不同的docker容器。

2. 启动不同的容器并运行测试。 --abort-on-container-exit告诉Docker Compose在til-app容器停止时停止postgres容器。用于此测试的postgres容器与您在开发期间使用的容器不同，并且不会发生冲突。

当测试完成运行时，您将在终端中看到所有测试通过的输出：



然后去哪儿?

在本章中，您学习了如何测试Vapor应用程序以确保它们正常工作。为您的应用程序编写测试意味着您可以在Linux上运行这些测试。这使您可以放心应用程序在部署时可以正常工作。拥有一个好的测试套件可以让您快速发展和调整您的应用程序。

Vapor的架构严重依赖协议。这与Vapor的依赖注入服务框架相结合，使测试变得简单且可扩展。对于大型应用程序，您甚至可能希望引入数据抽象层，这样您就不会使用真实数据库进行测试。

这意味着您不必连接到数据库来测试主逻辑并加快测试速度。

定期运行测试非常重要。使用Jenkins或Bitbucket Pipelines等持续集成（CI）系统可以测试每个提交。您还必须使您的测试保持最新。在将来更改行为的章节中，例如引入身份认证时，您将更改测试以使用这些新功能。