Chapter 19: API Authentication, Part 2

By Tim Condon

既然您已经实现了API身份认证，那么您的测试和iOS应用程序都不再有效。在本章中，您将学习考虑新身份认证要求所需的技术。

> 注意：您必须在项目中设置和配置PostgreSQL。如果还没有执行此操作，请按照第6章“Configuring a Database”中的步骤操作。

更新测试程序

现在您已经保护了API中的所有路由，您需要更新测试。在Xcode中，将方案设置为**TILApp-Package**，将部署目标设置为**My Mac**。打开**UserTests.swift**，找到testUserCanBeSavedWithAPI()并替换：

```
let user = User(name: usersName, username: usersUsername)
```

用以下内容：

```
let user = User(
  name: usersName,
  username: usersUsername,
  password: "password")
```

这包括密码，以便正确设置请求中的JSON body。接下来，打开**Models+Testable.swift**并在import FluentPostgreSQL下面添加以下内容：

```
import Crypto
```

这会导入Crypto模块以允许您使用BCrypt。接下来，使用以下内容替换User扩展中的create(name:username:on:)：

```
// 1
static func create(
  name: String = "Luke",
  username: String? = nil,
  on connection: PostgreSQLConnection
) throws -> User {
  let createUsername: String
  // 2
  if let suppliedUsername = username {
    createUsername = suppliedUsername
  // 3
  } else {
    createUsername = UUID().uuidString
  }

  // 4
  let password = try BCrypt.hash("password")
  let user = User(
    name: name,
    username: createUsername,
    password: password)
  return try user.save(on: connection).wait()
}
```

这是你改变的：

1. 将username参数设置为默认值为nil的可选字符串。

2. 如果提供了用户名，就使用它。

3. 如果未提供用户名，请使用UUID创建一个新的随机用户名。这可确保用户名在migration时是唯一的。

4. 创建用户。

在终端中，运行以下命令：

```
# 1
docker stop postgres-test
# 2
docker rm postgres-test
# 3
docker run --name postgres-test -e POSTGRES_DB=vapor-test \
  -e POSTGRES_USER=vapor -e POSTGRES_PASSWORD=password \
  -p 5433:5432 -d postgres
```

这是它的作用：

1. 停止测试PostgreSQL容器。

2. 删除测试PostgreSQL容器；这也将删除现有数据库。

3. 按第11章“Testing”中所述再次运行测试容器。

如果您现在运行测试，它们会因为对任何已认证路由的调用失败而崩溃。您需要为这些请求提供身份认证。

打开**Application+Testable.swift**并替换

```
import App
```

用以下内容：

```
@testable import App
import Authentication
```

这使您可以使用Token并导入身份认证模块。接下来，使用以下代码替换sendRequest<T>(to:method:headers:body:)的原型：

```
func sendRequest<T>(
  to path: String,
  method: HTTPMethod,
  headers: HTTPHeaders = .init(),
  body: T? = nil,
  loggedInRequest: Bool = false,
  loggedInUser: User? = nil
) throws -> Response where T: Content {
```

这会将loggedInRequest和loggedInUser添加为参数。您可以根据需要使用这些来告诉您的测试发送Authorization标头或使用指定的用户。接下来，在let responder = try self.make(Responder.self)之前添加以下内容：

```
var headers = headers
// 1
if (loggedInRequest || loggedInUser != nil) {
  let username: String
  // 2
  if let user = loggedInUser {
    username = user.username
  } else {
    username = "admin"
  }
  // 3
  let credentials = BasicAuthorization(
    username: username,
    password: "password")

  // 4
  var tokenHeaders = HTTPHeaders()
  tokenHeaders.basicAuthorization = credentials

  // 5
  let tokenResponse = try self.sendRequest(
    to: "/api/users/login",
    method: .POST,
```

```
      headers: tokenHeaders)
    // 6
    let token = try tokenResponse.content.syncDecode(Token.self)
    // 7
    headers.add(name: .authorization,
                value: "Bearer \(token.token)")
  }
```

这是新代码的作用：

1. 确定此请求是否需要身份认证。

2. 如果提供了用户，使用用户的详细信息创建BasicAuthorization类型。注意：这需要您知道用户的密码。由于测试中的所有用户都使用密码“password”，因此这不是问题。如果未指定用户，请使用“admin”。

3. 创建BasicAuthorization凭据。

4. 添加登录请求的基本授权标头。

5. 发送用户登录的请求并获取响应。

6. 从登录请求解码Token。

7. 将token添加到您尝试发送的请求的授权标头中。

在**Application+Testable.swift**中更改剩余的四个请求帮助程序以接受loggedInRequest和loggedInUser参数并将它们传递给sendRequest<T>(to:method:headers:body:loggedInRequest:loggedInUser:)：

```
  func sendRequest(
    to path: String,
    method: HTTPMethod,
    headers: HTTPHeaders = .init(),
    loggedInRequest: Bool = false,
    loggedInUser: User? = nil
  ) throws -> Response {
    let emptyContent: EmptyContent? = nil
    return try sendRequest(
      to: path, method: method,
      headers: headers, body: emptyContent,
      loggedInRequest: loggedInRequest,
      loggedInUser: loggedInUser)
  }

  func sendRequest<T>(
    to path: String,
    method: HTTPMethod,
    headers: HTTPHeaders,
    data: T,
    loggedInRequest: Bool = false,
```

```
    loggedInUser: User? = nil
  ) throws where T: Content {
    _ = try self.sendRequest(
      to: path, method: method,
      headers: headers, body: data,
      loggedInRequest: loggedInRequest,
      loggedInUser: loggedInUser)
  }

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

  func getResponse<T>(
    to path: String,
    method: HTTPMethod = .GET,
    headers: HTTPHeaders = .init(),
    decodeTo type: T.Type,
    loggedInRequest: Bool = false,
    loggedInUser: User? = nil
  ) throws -> T where T: Content {
    let emptyContent: EmptyContent? = nil
    return try self.getResponse(
      to: path, method: method,
      headers: headers, data: emptyContent,
      decodeTo: type,
      loggedInRequest: loggedInRequest,
      loggedInUser: loggedInUser)
  }
```

打开**AcronymTests.swift**，在testAcronymCanBeSavedWithAPI()中，将调用更改为app.getResponse(to:method:headers:data:decodeTo:)以设置loggedInRequest：

```
  let receivedAcronym = try app.getResponse(
    to: acronymsURI,
    method: .POST,
    headers: ["Content-Type": "application/json"],
    data: acronym,
    decodeTo: Acronym.self,
    loggedInRequest: true)
```

在testUpdatingAnAcronym()中，将用户传递给发送请求帮助程序：

```
try app.sendRequest(
  to: "\(acronymsURI)\(acronym.id!)",
  method: .PUT,
  headers: ["Content-Type": "application/json"],
  data: updatedAcronym,
  loggedInUser: newUser)
```

在testDeletingAnAcronym()中，在发送请求时设置loggedInRequest：

```
_ = try app.sendRequest(
  to: "\(acronymsURI)\(acronym.id!)",
  method: .DELETE,
  loggedInRequest: true)
```

接着，在testGettingAnAcronymsUser()中将解码的用户类型更改为User.Public：

```
let acronymsUser = try app.getResponse(
  to: "\(acronymsURI)\(acronym.id!)/user",
  decodeTo: User.Public.self)
```

由于应用程序不再在请求中返回用户的密码，因此必须将解码类型更改为User.Public。

接下来，在testAcronymsCategories()中用以下内容替换这两个请求：

```
let request1URL =
  "\(acronymsURI)\(acronym.id!)/categories/\(category.id!)"
_ = try app.sendRequest(
  to: request1URL,
  method: .POST,
  loggedInRequest: true)

let request2URL =
  "\(acronymsURI)\(acronym.id!)/categories/\(category2.id!)"
_ = try app.sendRequest(
  to: request2URL,
  method: .POST,
  loggedInRequest: true)
```

最后，使用以下内容替换XCTAssertEqual语句下的请求：

```
let request3URL =
  "\(acronymsURI)\(acronym.id!)/categories/\(category.id!)"
_ = try app.sendRequest(
  to: request3URL,
  method: .DELETE,
  loggedInRequest: true)
```

这些请求现在使用已认证的用户。

打开**CategoryTests.swift**并更改testCategoryCanBeSavedWithAPI()以使用已认证的请求：

```
let receivedCategory = try app.getResponse(
  to: categoriesURI,
  method: .POST,
  headers: ["Content-Type": "application/json"],
  data: category,
  decodeTo: Category.self,
  loggedInRequest: true)
```

接下来，在testGettingACategoriesAcronymsFromTheAPI()中，使用以下内容替换两个POST请求以使用已认证的用户：

```
let acronym1URL =
  "/api/acronyms/\(acronym.id!)/categories/\(category.id!)"

_ = try app.sendRequest(
  to: acronym1URL,
  method: .POST,
  loggedInRequest: true)

let acronym2URL =
  "/api/acronyms/\(acronym2.id!)/categories/\(category.id!)"

_ = try app.sendRequest(
  to: acronym2URL,
  method: .POST,
  loggedInRequest: true)
```

现在，打开**UserTests.swift**。首先，在testUsersCanBeRetrievedFromAPI()中更改请求，替换：

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

用以下内容：

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

这会将解码类型更改为User.Public。更新断言以说明admin用户：

```
XCTAssertEqual(users.count, 3)
XCTAssertEqual(users[1].name, usersName)
XCTAssertEqual(users[1].username, usersUsername)
XCTAssertEqual(users[1].id, user.id)
```

接下来，在testUserCanBeSavedWithAPI()中更新请求：

```
let receivedUser = try app.getResponse(
  to: usersURI,
  method: .POST,
  headers: ["Content-Type": "application/json"],
  data: user,
  decodeTo: User.Public.self,
  loggedInRequest: true)
```

这会将解码类型更改为User.Public并设置loggedInRequest标志。接着，更改第二个请求解码类型：

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

然后，更新testUserCanBeSavedWithAPI()中的断言以说明admin用户：

```
XCTAssertEqual(users.count, 2)
XCTAssertEqual(users[1].name, usersName)
XCTAssertEqual(users[1].username, usersUsername)
XCTAssertEqual(users[1].id, receivedUser.id)
```

最后，在testGettingASingleUserFromTheAPI()中更新请求：

```
let receivedUser = try app.getResponse(
  to: "\(usersURI)\(user.id!)",
  decodeTo: User.Public.self)
```

这会将解码类型更改为User.Public，因为响应不再包含用户的密码。构建并运行测试；他们应该都会通过。

更新iOS应用程序

由于API现在需要身份认证，iOS应用程序无法再创建缩略词。与测试一样，iOS应用程序必须更新以适应已认证的路由。TILiOS starter项目已更新，以在启动时显示新的LoginTableViewController。该项目还包含了Token模型，该模型与TIL Vapor应用程序的基本模型相同。最后，“create user”视图现在需接受密码。

在发送请求之前，请确保您的TIL Vapor应用程序正在运行。

Logging in

打开**AppDelegate.swift**。在application(_:didFinishLaunchingWithOptions:)中，应用程序检查新的Auth对象是否有token。如果没有token，它会启动登录界面；否则，它会正常显示缩略词列表。

打开**Auth.swift**。从AppDelegate调用的token检查使用**TIL-API-KEY**键在UserDefaults中查找token。在Auth中设置token时，它会将该token保存在UserDefaults中。

在Auth的底部创建一个新方法来登录用户：

```
// 1
func login(
  username: String,
  password: String,
  completion: @escaping (AuthResult) -> Void
) {
  // 2
  let path = "http://localhost:8080/api/users/login"
  guard let url = URL(string: path) else {
    fatalError()
  }
  // 3
  guard
    let loginString = "\(username):\(password)"
      .data(using: .utf8)?
      .base64EncodedString()
      else {
        fatalError()
  }

  // 4
  var loginRequest = URLRequest(url: url)
  // 5
  loginRequest.addValue(
    "Basic \(loginString)",
    forHTTPHeaderField: "Authorization")
  loginRequest.httpMethod = "POST"

  // 6
  let dataTask = URLSession.shared
    .dataTask(with: loginRequest) { data, response, _ in

    // 7
    guard
      let httpResponse = response as? HTTPURLResponse,
      httpResponse.statusCode == 200,
      let jsonData = data
      else {
        completion(.failure)
        return
    }

    do {
```

```
        // 8
        let token = try JSONDecoder()
          .decode(Token.self, from: jsonData)
        // 9
        self.token = token.token
        completion(.success)
      } catch {
        // 10
        completion(.failure)
      }
    }
    // 11
    dataTask.resume()
  }
```

这是新方法的作用：

1. 声明一种登录用户的方法。这将用户的用户名，密码和完成处理程序作为参数。
2. 构造登录请求的URL。
3. 为标头创建用户凭据的base64编码表示形式。
4. 为用户登录的请求创建URLRequest。
5. 为HTTP Basic验证添加必要的标头，并将HTTP方法设置为**POST**。
6. 创建一个新的URLSessionDataTask以发送请求。
7. 确保响应有效，状态码为200并包含响应body。
8. 将响应body解码为Token。
9. 将收到的token保存为Auth token。
10. 捕获任何错误并使用failure情况调用完成处理程序。
11. 启动数据任务以发送请求。

打开**LoginTableViewController.swift**。当用户点击**Login**时，应用程序调用loginTapped(_:)。在loginTapped(_:)的末尾，添加以下内容：

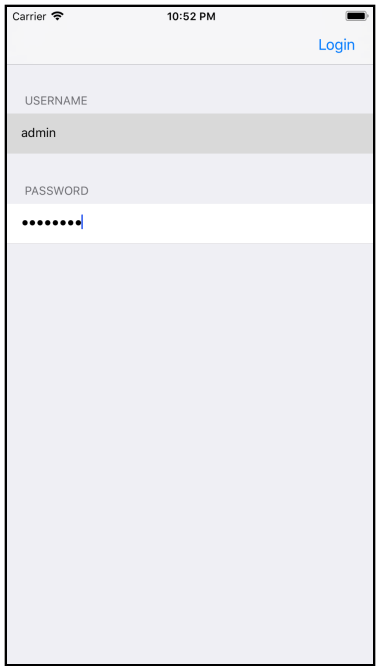
```
// 1
Auth().login(username: username, password: password) { result in
  switch result {
  case .success:
    DispatchQueue.main.async {
      let appDelegate =
        UIApplication.shared.delegate as? AppDelegate
      // 2
```

```
        appDelegate?.window?.rootViewController =
          UIStoryboard(name: "Main", bundle: Bundle.main)
          .instantiateInitialViewController()
      }
    case .failure:
      let message =
        "Could not login. Check your credentials and try again"
      // 3
      ErrorPresenter.showError(message: message, on: self)
    }
  }
```

这是它的作用：

1. 创建一个Auth实例并调用login(username:password:completion:)。
2. 如果登录成功，加载**Main.storyboard**以显示缩略词列表。
3. 如果登录失败，使用ErrorPresenter显示警报提示。

构建并运行。应用程序启动时，将显示登录界面。输入admin凭据并点按“**Login**”：



该应用程序会将您登录并带您进入主缩略词列表。

打开**Auth.swift**并将以下实现添加到logout()：

```
// 1
self.token = nil
DispatchQueue.main.async {
  // 2
```

```
  guard let applicationDelegate =
    UIApplication.shared.delegate as? AppDelegate else {
      return
  }
  let rootController =
    UIStoryboard(name: "Login", bundle: Bundle.main)
      .instantiateViewController(
        withIdentifier: "LoginNavigation")
  applicationDelegate.window?.rootViewController =
    rootController
}
```

这是它的作用：

1. 删除任何现有token。
2. 加载**Login.storyboard**并切换到登录界面。

构建并运行。由于您已经登录，因此应用程序会将您带到主缩略词视图。切换到“**Users**”选项卡，然后点击“**Logout**”。该应用程序返回到登录界面。

Creating models

starter项目简化了CreateAcronymTableViewController，因为您在创建缩略词时不再需要提供用户。打开**ResourceRequest.swift**。在save(_:completion:)中的var urlRequest = URLRequest(url: resourceURL)之前添加以下内容：

```
// 1
guard let token = Auth().token else {
  // 2
  Auth().logout()
  return
}
```

这是它的作用：

1. 从Auth服务获取token。
2. 如果token不存在，调用logout()，因为用户需要再次登录才能获得新token。

接下来，在urlRequest.addValue("application/json", forHTTPHeaderField:"Content-Type")下面添加：

```
urlRequest.addValue(
  "Bearer \(token)",
  forHTTPHeaderField: "Authorization")
```

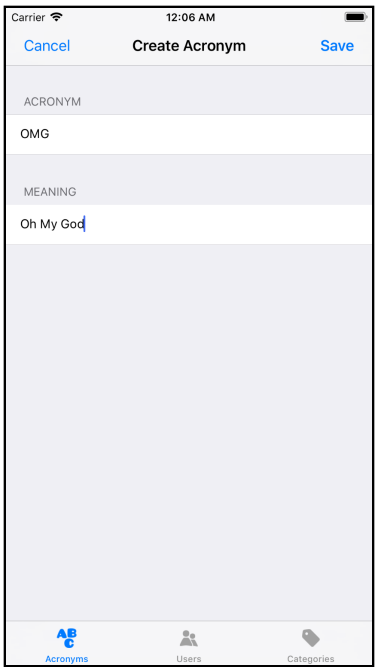
这会使用**Authorization**标头将token添加到请求中。

最后，在guard httpResponse.statusCode == 200, let jsonData = data else { 里的completion(.failure)之前添加以下内容：

```
if httpResponse.statusCode == 401 {
  Auth().logout()
}
```

这将检查失败的状态码。如果响应返回**401 Unauthorized**，则表示token无效。将用户退出以触发新的登录流程。

构建并运行，并再次登录。单击“+”，您将看到新的创建缩略词页面，没有用户选项：



填写表单并单击“**Save**”以创建缩略词。您还可以创建用户和类别。请注意，“create user”流程现在包含一个新模型CreateUser。应用程序将此模型发送到API，因为它包含password属性。

Acronym requests

`URLRequest(url: resource)` add the following:

您仍然需要为缩略词请求添加身份认证。打开**AcronymRequest.swift**并在update(with:completion:)中的var urlRequest = URLRequest(url: resource)之前添加以下内容：

```
guard let token = Auth().token else {
  Auth().logout()
  return
}
```

与ResourceRequest一样，它从Auth获取token，如果出现错误则调用logout()。在urlRequest.addValue("application/json", forHTTPHeaderField: "Content-Type")之后添加：

```
urlRequest.addValue(
  "Bearer \(token)",
  forHTTPHeaderField: "Authorization")
```

这会将token添加到**Authorization**标头。接下来，在guard httpResponse.statusCode == 200中的completion(.failure)之前添加：

```
if httpResponse.statusCode == 401 {
  Auth().logout()
}
```

如果token无效，则调用logout()。接下来，更改delete()以向请求添加身份认证。在函数的开头添加：

```
guard let token = Auth().token else {
  Auth().logout()
  return
}
```

接下来，在urlRequest.httpMethod = "DELETE"之后添加以下内容：

```
urlRequest.addValue(
  "Bearer \(token)",
  forHTTPHeaderField: "Authorization")
```

最后在add(category:completion:)中的let url = ... 之前获取令牌：

```
guard let token = Auth().token else {
  Auth().logout()
  return
}
```

接下来，在urlRequest.httpMethod = "POST"之后，将token添加到请求中：

```
urlRequest.addValue(
  "Bearer \(token)",
  forHTTPHeaderField: "Authorization")
```

最后，如果响应返回**401 Unauthorized**，则在guard httpResponse.statusCode == 201 else 里将用户注销：

```
if httpResponse.statusCode == 401 {
  Auth().logout()
}
```

构建并运行。您现在可以删除和编辑缩略词并为其添加类别。

然后去哪儿?

在本章中，您学习了如何使用HTTP基本身份认证更新测试以获取token，并在相应的测试中使用该token。您还更新了随附的iOS应用，以使用已认证的API。

目前，只有已认证的用户才能在API中创建缩略词。但是，该网站仍然是开放的，任何人都可以做任何事情！在下一章中，您将学习如何将身份认证应用于Web前端。您将了解验证API和网站之间的区别，以及如何使用cookies和sessions。