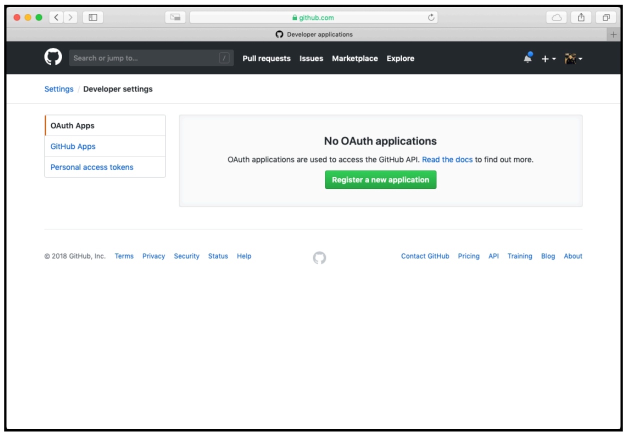
# Chapter 23: GitHub Authentication

By Tim Condon

In the previous chapter, you learned how to authenticate users using Google. In this chapter, you'll see how to build upon this and allow users to log in with their GitHub accounts.

## Setting up your application with GitHub

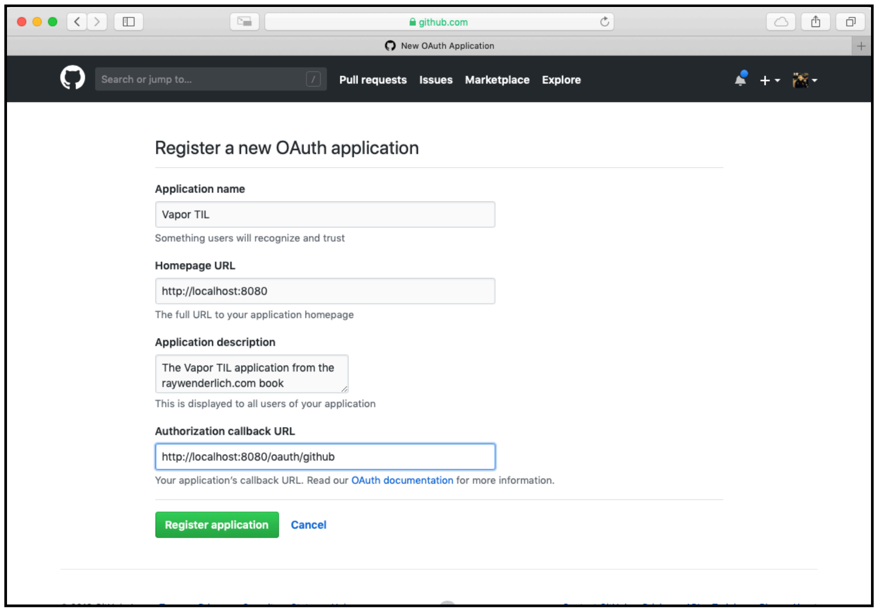
To be able to use GitHub OAuth in your application, you must first register the application with GitHub. In your browser, go to https://github.com/settings/developers. Click **Register a new application**:



OAuth applications are used to access the GitHub API. Read the docs to find out more.

**Note**: You must have a GitHub account to complete this chapter. If you don't have one, visit https://github.com/join to create one. This chapter also assumes you added Imperial as a dependency to your project in the previous chapter.

Fill in the form with an appropriate name, e.g. **Vapor TIL**. Set the **Homepage URL** to **http://localhost:8080** for this application and provide a sensible description. Set the **Authorization callback URL** to **http://localhost:8080/oauth/github**. This is the URL that GitHub redirects back to once users have allowed your application access to their data:



Register a new OAuth application

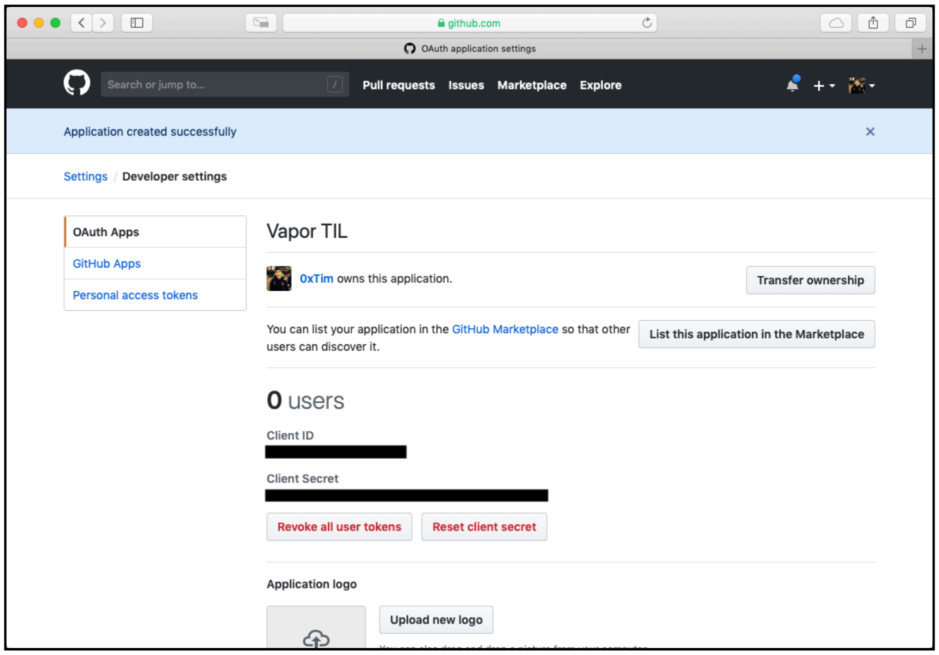
Application name
Vapor TIL
Something users will recognize and trust

Homepage URL
http://localhost:8080
The full URL to your application homepage

Application description
The Vapor TIL application from the raywenderlich.com book
This is displayed to all users of your application

Authorization callback URL
http://localhost:8080/oauth/github
Your application's callback URL. Read our OAuth documentation for more information.

Click **Register application**. After it creates the application, the site takes you back to the application's information page. That page provides the client ID and client secret that you need:



Application created successfully

Settings / Developer settings

Vapor TIL

0xTim owns this application.

You can list your application in the GitHub Marketplace so that other users can discover it.

0 users

Client ID
[redacted]

Client Secret
[redacted]

> **Note**: You *must* keep these safe and secure. Your secret allows you access to GitHub's APIs and you should not share or check the secret into source control. You should treat it like a password.

## Integrating with Imperial

Now that you've registered your application with GitHub, you can start integrating Imperial. Open **ImperialController.swift** and add the following under `processGoogleLogin(request:token:)`:

```
func processGitHubLogin(request: Request, token: String)
  throws -> Future<ResponseEncodable> {
    return request.future(request.redirect(to: "/"))
}
```

This defines a method to handle the GitHub login, similar to the initial handler for Google logins. The handler simply redirects the user to the home page. Imperial uses this method as the final callback once it has handled the GitHub redirect.

Next, set up the Imperial routes by adding the following at the bottom of `boot(router:)`:

```
guard let githubCallbackURL =
  Environment.get("GITHUB_CALLBACK_URL") else {
    fatalError("GitHub callback URL not set")
}
try router.oAuth(
  from: GitHub.self,
  authenticate: "login-github",
  callback: githubCallbackURL,
  completion: processGitHubLogin)
```

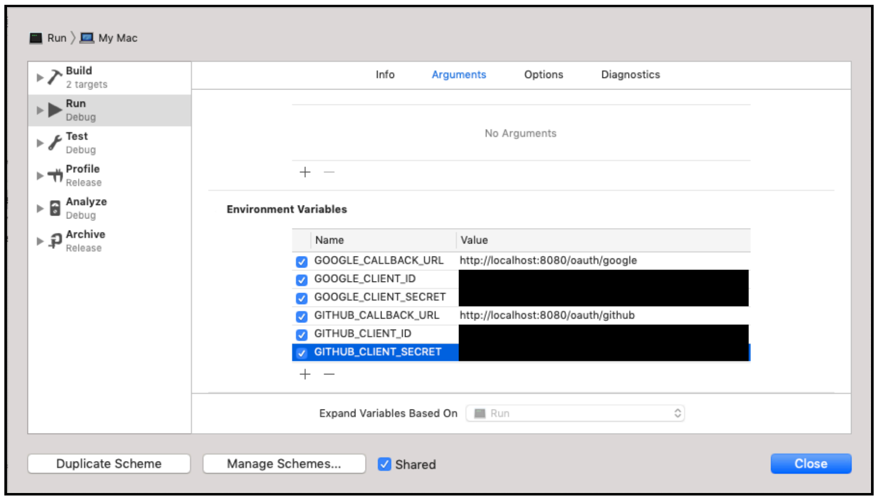
Here's what this does:

- Get the callback URL from an environment variable — this is the URL you set up when registering the application with GitHub.
- Register Imperial's GitHub OAuth router with your app's router.
- Tell Imperial to use the GitHub handler.
- Set up the **/login-github** request as the route that triggers the OAuth flow. This is the route the application uses to allow users to log in via GitHub.
- Provide the callback URL to Imperial.

- Set the completion handler to `processGitHubLogin(request:token:)` - the method you created above.

As before, you need to provide Imperial the client ID and client secret that GitHub gave you using environment variables. You must also provide the redirect URL. To do this in Xcode, click the **Run** scheme, then click **Edit scheme**. Under the **Arguments** tab, add three new **Environment Variables** as shown below:

- **GITHUB_CALLBACK_URL**: **http://localhost:8080/oauth/github** — this is the URL you provided to GitHub.
- **GITHUB_CLIENT_ID**: The client ID provided by GitHub.
- **GITHUB_CLIENT_SECRET**: The client secret provided by GitHub.



Environment Variables

| Name | Value |
|---|---|
| ✓ GOOGLE_CALLBACK_URL | http://localhost:8080/oauth/google |
| ✓ GOOGLE_CLIENT_ID | [redacted] |
| ✓ GOOGLE_CLIENT_SECRET | [redacted] |
| ✓ GITHUB_CALLBACK_URL | http://localhost:8080/oauth/github |
| ✓ GITHUB_CLIENT_ID | [redacted] |
| ✓ GITHUB_CLIENT_SECRET | [redacted] |

Expand Variables Based On: Run

✓ Shared

**Note**: Be sure you still have environment variables set for **GOOGLE_CALLBACK_URL, GOOGLE_CLIENT_ID** and **GOOGLE_CLIENT_SECRET** or your app won't start.

## Integrating with web authentication

As in the previous chapter, it's important to match the experience for a regular login. Again, you'll create a new user when a user logs in with GitHub for the first time. You can use GitHub's API with the user's OAuth token.

At the bottom of **ImperialController.swift**, add a new type to decode the data from GitHub's API:

```
struct GitHubUserInfo: Content {
  let name: String
  let login: String
}
```

The request to GitHub's API returns many fields. However, you only care about the login, which becomes the username, and the name.

Next, under `GitHubUserInfo`, add the following:

```
extension GitHub {
  // 1
  static func getUser(on request: Request)
    throws -> Future<GitHubUserInfo> {
      // 2
      var headers = HTTPHeaders()
      headers.bearerAuthorization =
        try BearerAuthorization(token: request.accessToken())

      // 3
      let githubUserAPIURL = "https://api.github.com/user"
      // 4
      return try request
        .client()
        .get(githubUserAPIURL, headers: headers)
        .map(to: GitHubUserInfo.self) { response in
          // 5
          guard response.http.status == .ok else {
            // 6
            if response.http.status == .unauthorized {
              throw Abort.redirect(to: "/login-github")
            } else {
              throw Abort(.internalServerError)
            }
          }
          // 7
          return try response.content
            .syncDecode(GitHubUserInfo.self)
      }
  }
}
```

Here's what this does:

1. Add a new function to Imperial's `GitHub` service which gets a user's details from the GitHub API.
2. Set the headers for the request by adding the OAuth token to the authorization header.
3. Set the URL for the request — this is GitHub's API to get the user's information.

4. Use `request.client()` to create a client to send a request. `get()` sends an HTTP GET request to the URL provided. Unwrap the returned future response.

5. Ensure the response status is **200 OK**.

6. Otherwise return to the login page if the response was **401 Unauthorized** or return an error.

7. Decode the data from the response to `GitHub` and return the result.

Next, replace the body of `processGitHubLogin(request:token:)` with the following:

```
// 1
return try GitHub
  .getUser(on: request)
  .flatMap(to: ResponseEncodable.self) { userInfo in
    // 2
    return User
      .query(on: request)
      .filter(\.username == userInfo.login)
      .first()
      .flatMap(to: ResponseEncodable.self) { foundUser in
        guard let existingUser = foundUser else {
          // 3
          let user = User(name: userInfo.name,
                          username: userInfo.login,
                          password: UUID().uuidString)
          // 4
          return user
            .save(on: request)
            .map(to: ResponseEncodable.self) { user in
              // 5
              try request.authenticateSession(user)
              return request.redirect(to: "/")
          }
        }
        // 6
        try request.authenticateSession(existingUser)
        return request.future(request.redirect(to: "/"))
    }
}
```

Here's what the new code does:

1. Get the user information from GitHub.

2. See if the user exists in the database by looking up the login property as the username.

3. If the user doesn't exist, create a new `User` using the name and username from the user information from GitHub. Set the password to a UUID, since you don't need it.

4. Save the user and unwrap the returned future.

5. Call `request.authenticateSession(_:)` to save the created user in the session so the website allows access. Redirect back to the home page.

6. If the user already exists, authenticate the user in the session and redirect to the home page.

The final thing to do is to add a button on the website to allow users to make use of the new functionality! Open **login.leaf** and, under `</form>`, add the following:

```
<a href="/login-github">
  <img class="mt-3" src="/images/sign-in-with-github.png"
   alt="Sign In With GitHub">
</a>
```

The sample project for this chapter contains a new image, **sign-in-with-github.png**, to display a **Sign in with GitHub** button. This adds the image as a link to **/login-github** — the route provided to Imperial to start the login. Build and run the application and then visit **http://localhost:8080** in your browser. Click **Create An Acronym** and the application takes you to the login page. You'll see the new **Sign in with GitHub** button next to the **Sign in with Google** button:

Click the new button and the application takes you to a GitHub page to allow the TIL application access to your information:

Click **Authorize** and the application redirects you back to the home page. Go to the **All Users** screen and you'll see your new user account. If you create an acronym, the application also uses that new user.

## Where to go from here?

In this chapter, you learned how to integrate GitHub login into your website using Imperial and OAuth. This complements the Google and first-party sign in experiences. This allows your users to choose a range of options for authentication.

In the next chapter, you'll learn how to integrate with a third party email provider. You'll use another community package and learn how to send emails. To demonstrate this, you'll implement a password reset flow into your application in case users forget their password.