Auth0 Identity Labs

Welcome to the home for Auth0's digital identity labs! These exercises serve as a learning tool to be combined with our Learn Identity video series. Each lab is meant to be completed once a video (or series of videos) is complete.

A few general things to keep in mind as you work through these labs:

Plan to take around 1 hour or so (longer depending on your coding experience) for each lab. Optionally, video demonstrations of the lab exercises are available.

These labs are designed to illustrate the basic concepts of digital identity, OAuth, and OpenID Connect. The goal is to see the parameters and data that come together to create a complete authentication flow. As such, take your time and read through each section carefully. Completing the lab successfully is less important than understanding the concepts within.

The code samples here should not be used as-is in a production app. The code here was written for instructional purpose and simplicity. For guidance on integrating Auth0 with a new or existing app, please see the Quickstarts listed on our documentation home page (choose an application type, then the technology you're using).

Each lab will have a list of pre-requisites to complete or install. Please take note of specific version numbers as these can have an effect on how the labs work.

ℹ️ The error messages displayed in the browser and in the console can often clue you into something that is going wrong. Auth0 error pages typically include a link under the "Technical Details" header that will give you more information about what went wrong. A "SyntaxError" line in your terminal window when starting the server indicates a typo or missed line.

ℹ️ The code samples will indicate what lines to add or modify. In most cases, the order of operations (as in, when a particular line of code runs) matters greatly, so pay attention to those lines and the description above each snippet.

```
// lab-01/begin/server.js
// 👆 That is the file you should be in for these changes.

require('dotenv').config();
// 👆 This is code that should not be changed.

// ... other required packages
// 👆 This is information to help you place the new code.
```

```
// Add the code below 👇
const session = require('cookie-session');
const { auth } = require('express-openid-connect');
// 👆 This is what should be added

// ...
// 👆 This indicates that there is other code after that should not be changed.
```

ℹ️ Terminal commands are proceeded by a `❯` character. If you're copying those commands, exclude that character and the space that follows. Exclude all lines that do not start with `❯`; those are there to show the expected output.

```
# This line is informational; read but don't use.
❯ this line is the command to copy

This line shows sample output; read but don't use.
```

And with that, let's get started!

Lab 1: Web Sign-In

Lab 1, Exercise 1: Adding Web Sign-In

In this exercise, you will learn how to add sign-in to an app using:

- Node.js + Express
- An Express middleware to handle checking authentication and redirecting to login
- Auth0 as an Authorization Server

Lab

A simple Node.js Express application has been created to get you started. This is a web application with two pages. The first page, served under the root path /, shows “Hello World” and a link (“Expenses”) to the second page. The second page, served at /expenses, shows a table with expenses. At this point, these expenses are hard-coded; you will learn how to consume them from an API secured with Auth0 in the next lab.

1. Open your Terminal app, clone the identity exercise repo, then go to the /lab-01/begin folder:

```
❯ git clone https://github.com/auth0/identity-102-exercises.git
Cloning into 'identity-102-exercises'...

❯ cd identity-102-exercises/lab-01/begin
```

2. Open a code editor like VS Code or Atom in the same directory (File > Open) and review the server.js code. This is a generic Node.js web application that uses ejs for views and morgan to log HTTP requests.

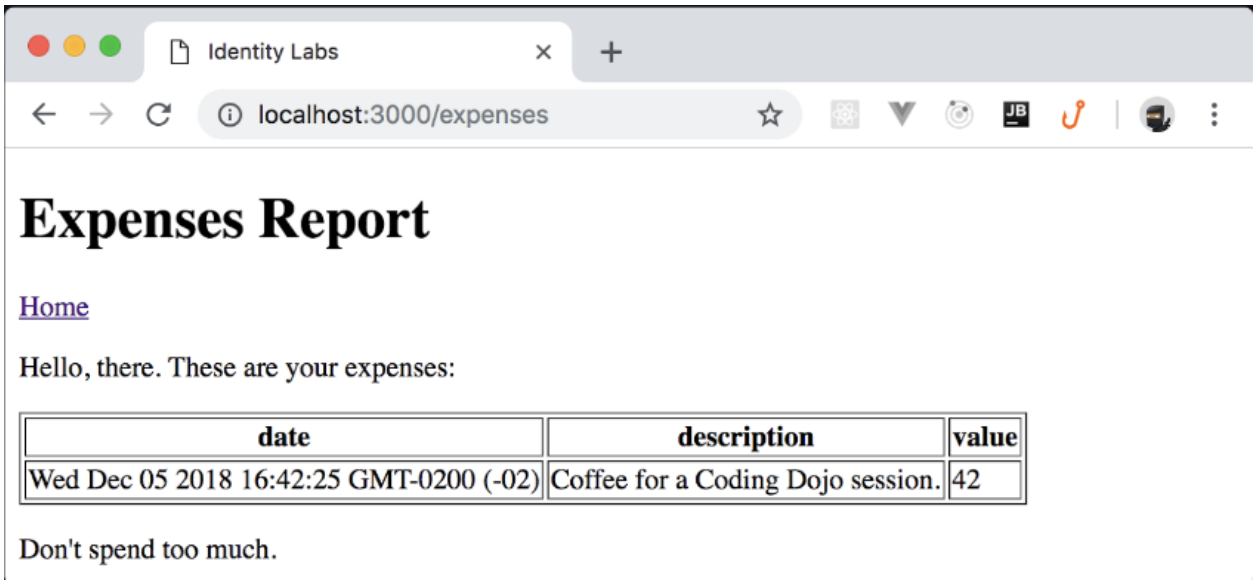
3. The .env-sample file will be used for the environment variables you need for this lab. It's populated with a PORT (the port number where the app will run) and an APP_SESSION_SECRET (value used to encrypt the cookie data). You will set the other values later on in the lab. For now, create a copy of this file in the same folder and name it .env. Run the following commands in your terminal (or copy, paste, and rename the sample file in your editor):

4. In your terminal, use npm to install all the dependencies and start the application:

```
❯ npm install
# Ignore any warnings

added XX packages in X.XXs
❯ npm start

listening on http://localhost:3000
```

NOTE: If you see a message like "Error: listen EADDRINUSE :::3000" in your terminal after starting the application, this means that port 3000 is in use somewhere. Change the PORT value in your .env file to "4000" and try again.

5. Open a Web browser and go to localhost:3000 (or http://localhost:PORT where PORT is the value of the environment variable, in case you changed its value). You should see a page with a "Hello World" message. Click the Expenses link to view the expenses page.



6. Now, we're ready to start adding authentication! Switch to your terminal window and press [CTRL] + [c] to stop the server, then use npm to install the package you'll use to secure the app. The express-openid-connect package is a simple Express middleware that provides OpenID Connect and JWT implementation.

```
# Continuing from previous terminal session ...
listening on http://localhost:3000
^C # Command to stop the server
❯ npm install express-openid-connect@1.0.2 --save
# Ignore any warnings

+ express-openid-connect@1.0.2
added XX packages in X.XXs
```

7. Next, update your application code to require express-openid-client in the server.js file:

```
// lab-01/begin/server.js

require('dotenv').config();
// ... other required packages

// Add the line below 👇
const { auth } = require('express-openid-connect');

// ...
```

8. Now add the authentication middleware that will be used for all application routes:

```
// lab-01/begin/server.js
// ...

const app = express();
app.set('view engine', 'ejs');
app.use(morgan('combined'));

// Add the code below 👇
app.use(auth({
  auth0Logout: true,
  baseURL: appUrl
}));

// ... other app routes
```

The middleware you installed automatically defines three routes in your application:

- **/login** - builds the OpenID Connect request and redirects to the authorization server (in this case, Auth0). For this to work properly, the middleware needs to include specific parameters with the request. You will configure these values using environment variables in the next step.
- **/callback** - handles the response from the authorization server, performs required validations like nonce, state, and token verification using the openid-client package, and sets the user in the session from the ID token claims.
- **/logout** - terminates the session in the application and redirects to Auth0 to end the session there as well.

The middleware will also augment Express's request object with additional properties whenever the request is authenticated. For example, **req.openid.user** is a property that will contain user information.

Note: The **auth0Logout: true** configuration key passed to auth() tells the middleware that, when the user logs out of the application, they should be redirected to a specific Auth0 URL to end their session there as well.

The middleware needs to be initialized with some information to build a proper OpenID request and send it to the authorization server. This information includes:

- The URL of the authorization server. This URL will be used to download the OpenID Connect configuration from the discovery document, available at the URL *https://{your-auth0-domain}/.well-known/openid-configuration* (here is the configuration for the main Auth0 tenant). The discovery document is a standard OpenID Connect mechanism used to publish relevant discovery metadata of the

OpenID Connect provider, including a link to what keys should be used for validating the tokens it issues.

- The unique identifier for your application. This is created on the authorization server and is a unique string that identifies your application. This identifier must be provided in each request, so the authorization server knows what application the authentication request is for.

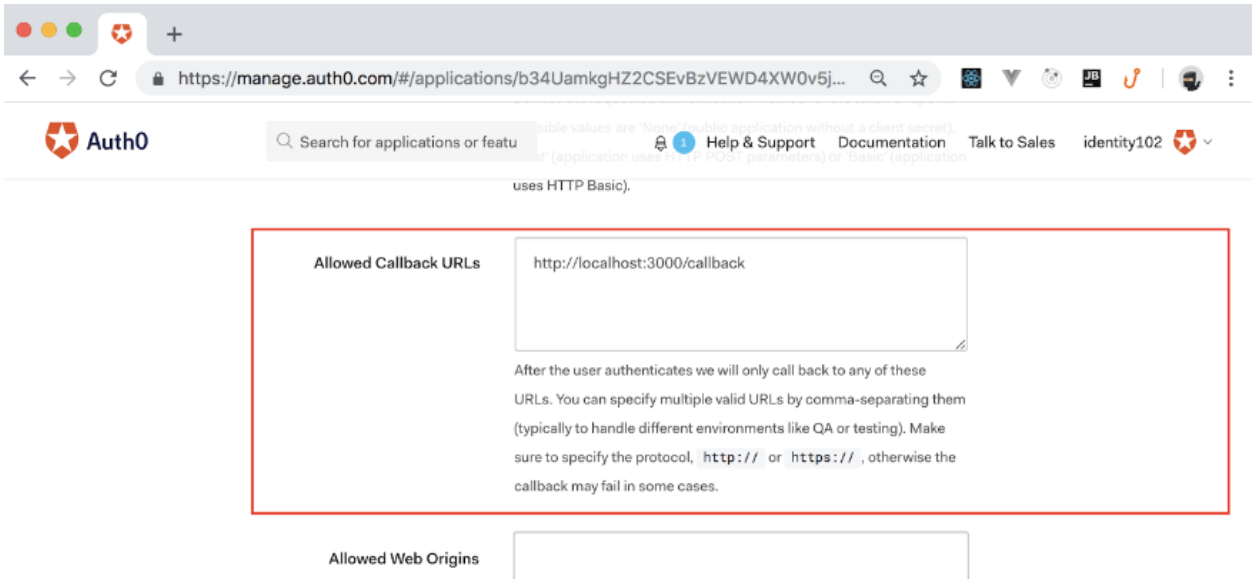
You will use the Auth0 Dashboard to register your application with Auth0. Afterward, you'll be able to retrieve the two values above and configure them as environment variables for your app. The middleware will read these environment variables and use them to build the request when a user tries to authenticate.

9. Log into the Auth0 Dashboard, go to the Applications page, and click the Create Application button.

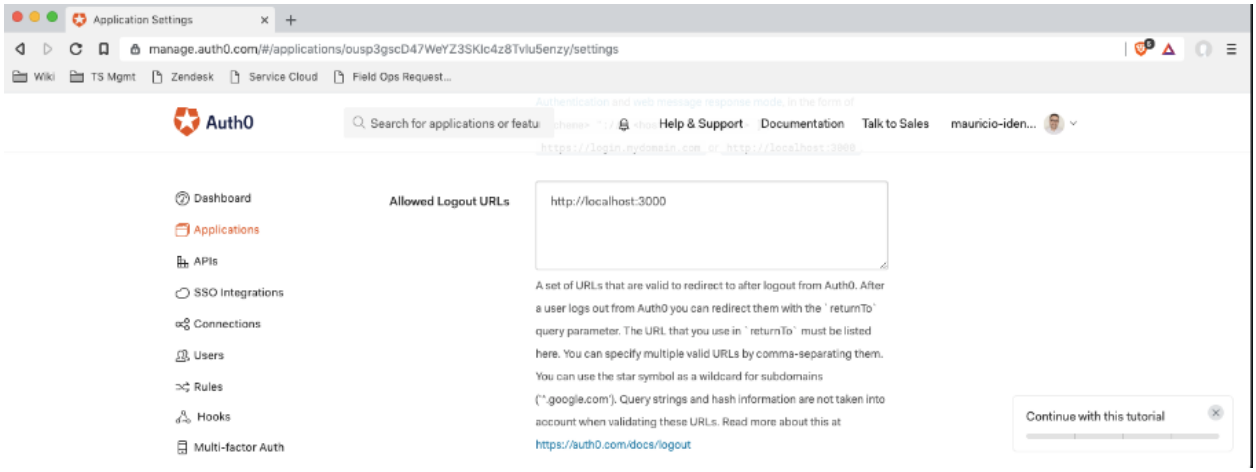
10. Set a descriptive name (e.g., "Identity Lab 1 - Web Sign In"), choose Regular Web Applications for the type, and click Create.

11. You should now see the Quickstart section that describes how to integrate Auth0 with a production application. Click the Settings tab at the top to see the Application settings.

12. Add your application's callback URL - http://localhost:3000/callback (adjust the port number if needed) - to the Allowed Callback URLs field. Auth0 will allow redirects only to the URLs in this field after authentication. If the one provided in the authorization URL does not match any in this field, an error page will be displayed.



13. Next, add http://localhost:3000 (adjust the port number if needed) to the Allowed Logout URLs field. Auth0 will allow redirects only to the URLs in this field after logging out of the authorization server.



14. Scroll down and click Show Advanced Settings, then OAuth. Make sure JsonWebToken Signature Algorithm is set to *RS256*.

15. Scroll down and click Save Changes

16. Open your *.env* file. Add https:// to the Domain from Auth0 as the value for the *ISSUER_BASE_URL* key. Add the Client ID from Auth0 as the value for the *CLIENT_ID* key. Add a long, random string and the value for the *APP_SESSION_SECRET* key. You *.env* file should look similar to the sample below:

```
ISSUER_BASE_URL=https://your-tenant-name.auth0.com
CLIENT_ID=0VMFtHgN9mUa1YFoDx3CD2Qnp2Z11mvx
APP_SESSION_SECRET=a36877de800e31ba46df86ec947dab2fc8a2f7e1d23688ce2010cd076539bd28
PORT=3000
```

Note: Mac users can enter the following in Terminal to get a random string suitable for the secret value: openssl rand -hex 32. This value is used by the session handler in the SDK to generate opaque session cookies.

17. Save the changes to *.env* and restart the server as before, but do not open it in a browser yet.

    Your app is now ready to authenticate with Auth0 using OpenID Connect! Before testing it, continue to the next exercise, where you will review the interactions that happen under the hood between your app and Auth0 while you sign up and log in.

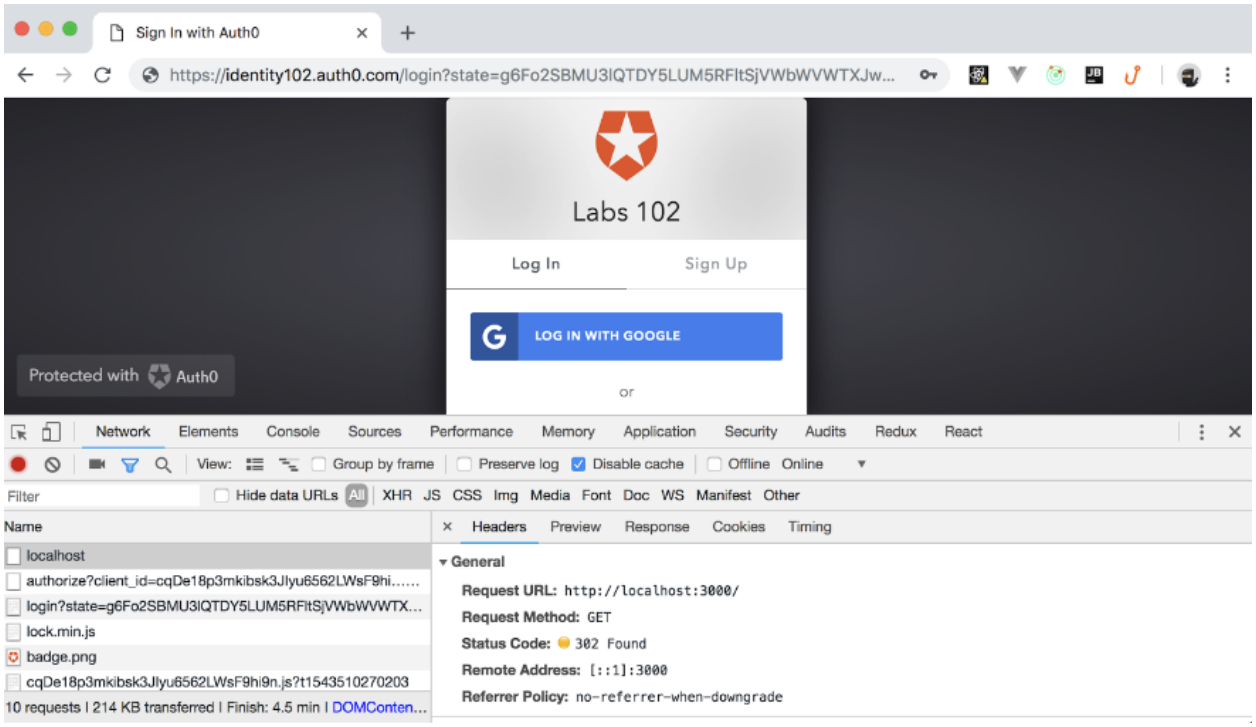
```
# Continuing from previous terminal session ...
listening on http://localhost:3000
^C # Command to stop the server
❯ npm start

listening on http://localhost:3000
```

Lab 1, Exercise 2: Using Network Traces

In this exercise, you will sign up for your application (which will also log you in) while exploring some of the relevant network traces of the authentication process.

1. Using Chrome, open Developer Tools. Switch to the Network tab then open your local application. You should immediately be redirected to Auth0 to login.

2. The first request you should see is a GET request to your application homepage:

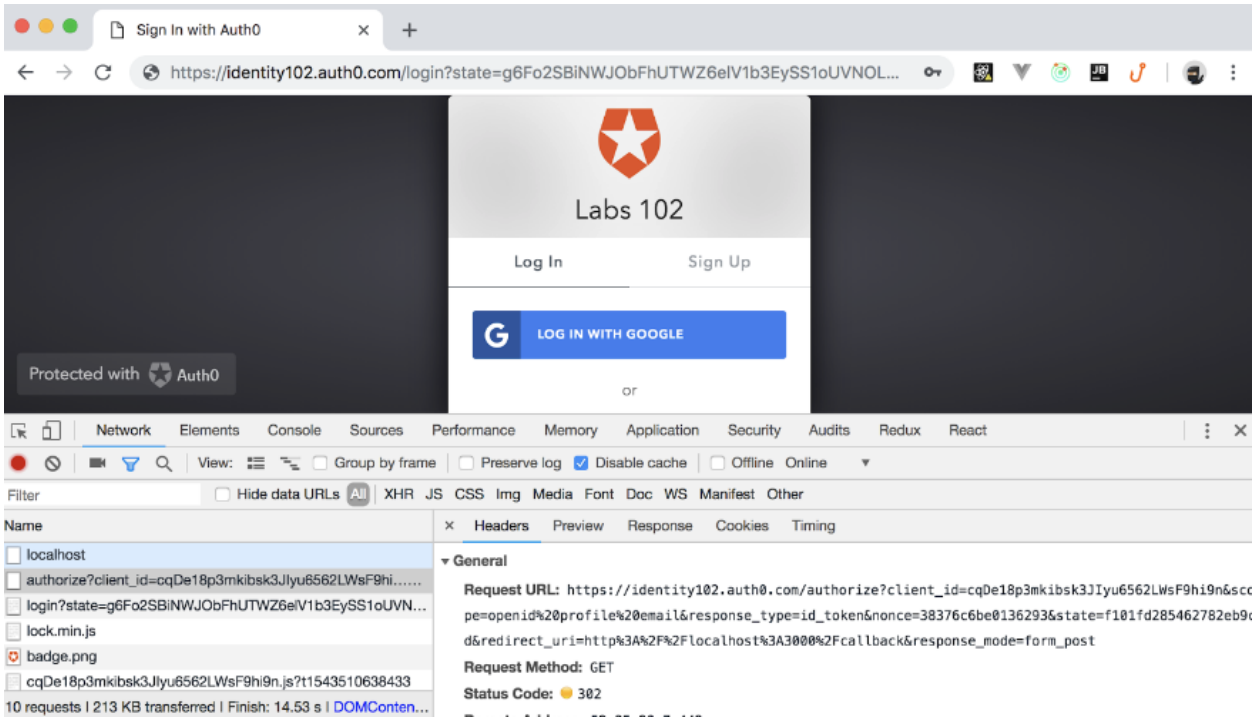


3. After that, you should see a GET request to https://your-tenant-name.auth0.com/authorize. This is the middleware added in exercise 1 taking over. The middleware checks if the user is logged in and, because they are not, it builds the OpenID Connect request to the authorization server URL and forwards the user to it. In this case, the complete GET request URL will look something like this (line breaks added for clarity):

```
https://YOUR_DOMAIN/authorize
?client_id=YOUR_CLIENT_ID
&scope=openid%20profile%20email
&response_type=id_token
&nonce=71890cc63567e17b
&state=85d5152581b310e3389b
&redirect_uri=http%3A%2F%2Flocalhost%3A3000
&response_mode=form_post
```

The middleware sends several parameters. The important ones for this lab are:

- **client_id**: the unique identifier of your app at the authorization server
- **response_type**: the requested artifacts; in this case, you are requesting an ID token
- **scope**: why the artifacts are required, i.e. what content and capabilities are needed
- **redirect_uri**: where the results are to be sent after the login operation, i.e. the callback URL.
- **response_mode**: how the response from the server is to be sent to the app; in this case, the response we want is a POST request.

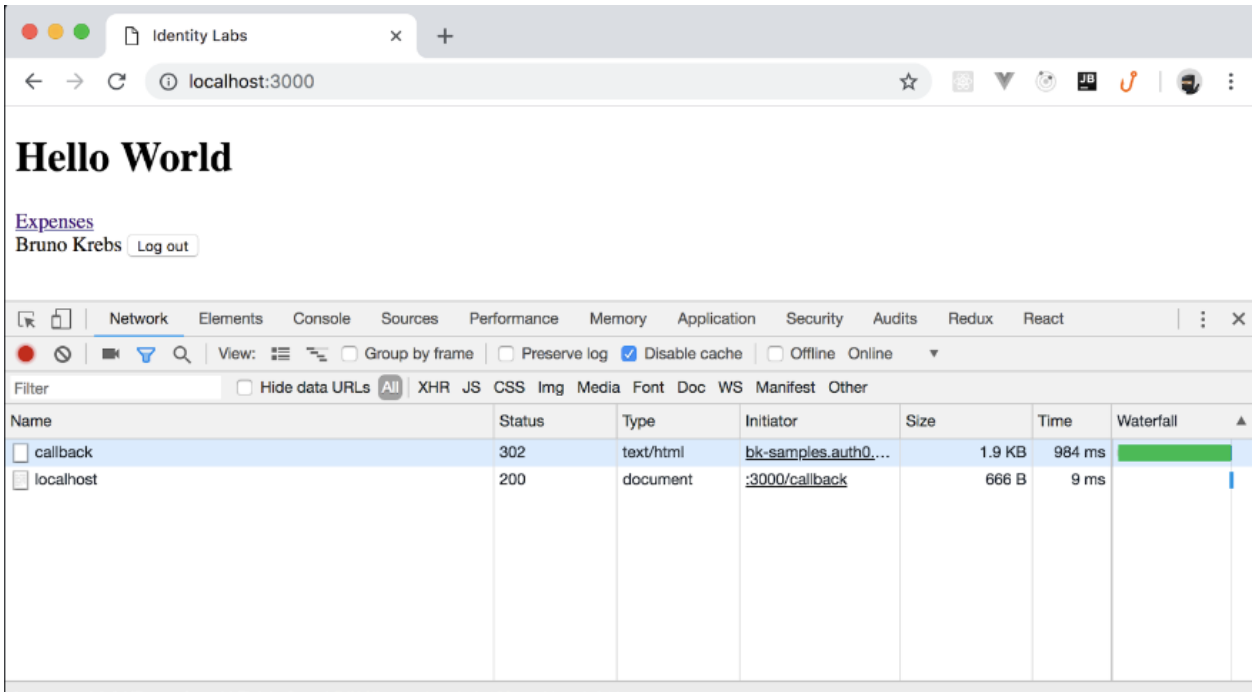


Note: If you scroll down while on the Headers tab in Chrome Developer Tools to the Query String Parameters section, you can see the different URL parameters in a more-readable table format.

4. If you already have a user created, enter your credentials and continue below. If not, click the Sign Up link at the bottom (if you're using the classic page, this will be a tab at the top) and enter an email and password.

5. A consent dialog will be shown requesting access to your profile and email. Click the green button to accept and continue.

6. The authorization server will log you in and POST the response - an error if something went wrong or the ID token if not - back to the callback URL for your application. Once you've successfully logged in, you should see your user name on the page. This means authentication has been configured properly!

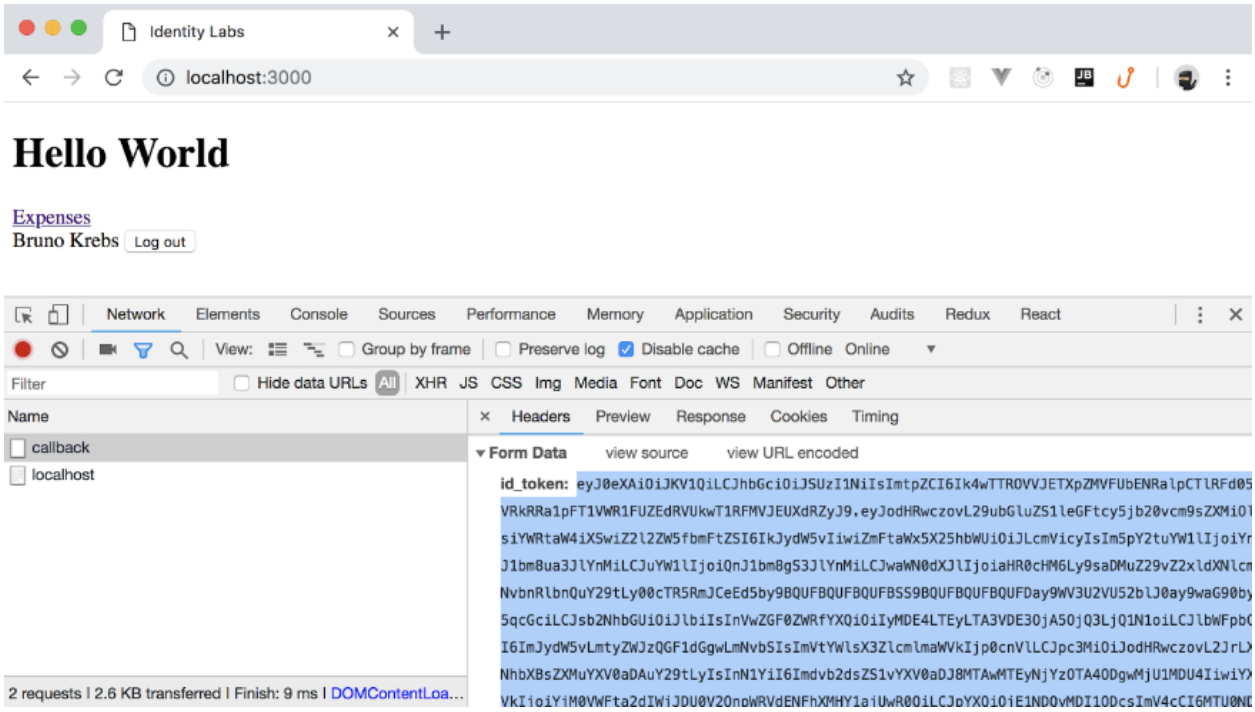


The complete trace of the callback request is:

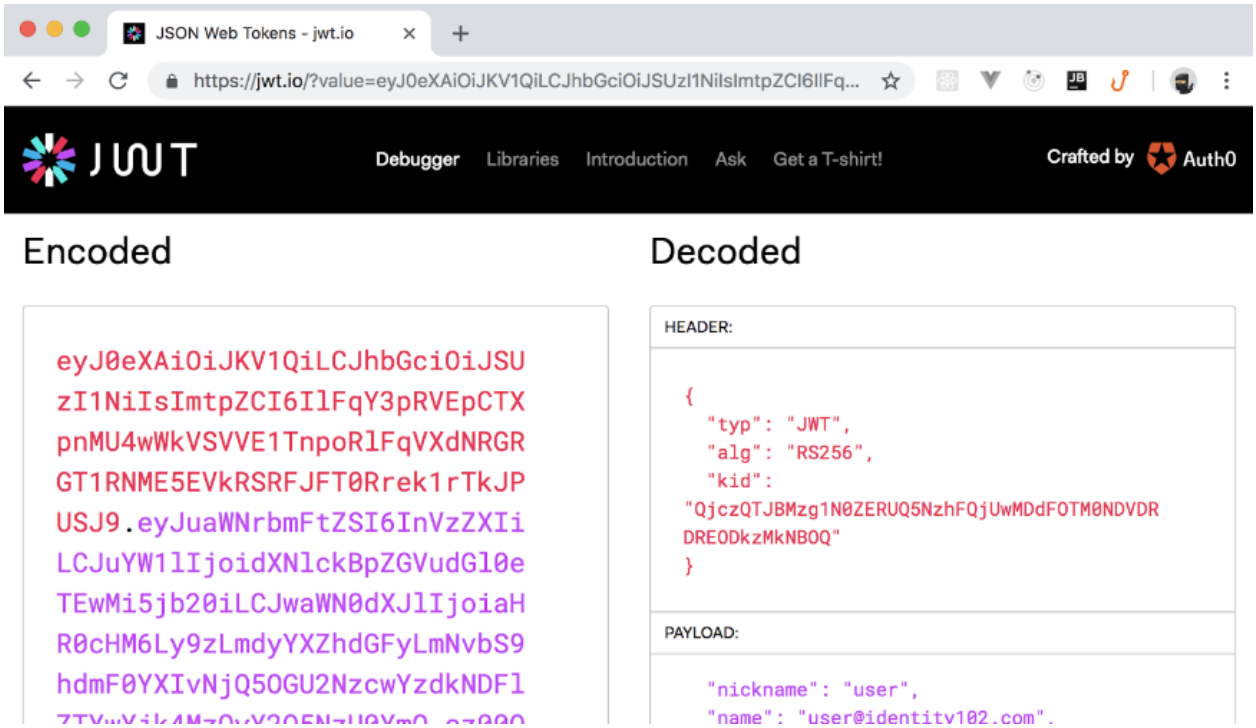
```
Request URL: `http://localhost:3000/callback`
Request Method: POST
Status Code: 302 Found
Remote Address: [::1]:3000
Referrer Policy: no-referrer-when-downgrade
Connection: keep-alive
Content-Length: 46
Content-Type: text/html; charset=utf-8
Date: Mon, 12 Nov 2018 23:00:08 GMT
Location: /
Set-Cookie: identity102-lab=eyJyZX[..]; path=/; httponly
Set-Cookie: identity102-lab.sig=wld5z7[..]; path=/; httponly
Vary: Accept
X-Powered-By: Express
id_token: eyJ0eX[..].eyJuaW[..].IEpcS5[..]
state: 85d5152581b310e3389b
```

Note: If you see an error in your console about an ID token used too early, this is likely a clock skew issue in your local environment. Try restarting your machine and walking through the login steps again from the beginning.

7. Click on the callback request, then search for the Form Data section of the Headers tab of the Developer Console. Copy the complete *id_token* value.



8. Go to jwt.io and paste the ID token copied from the last step into the text area on the left. Notice that as soon as you paste it, the contents of the text area on the right are updated. This is because the site decodes your ID token and displays its contents (claims) in that panel.



Note the following:

- The token structure: it consists of the header (information about the token), the payload (the token's claims and user profile information), and the signature.
- The claim *iss* is for the issuer of the token. It denotes who created and signed it. The value should match your Auth0 Domain value with an *https://* prefixed.
- The claim *sub* is the subject of the token. It denotes to whom the token refers. In our case, the value matches the ID of the Auth0 user.
- The claim *aud* is the audience of the token. It denotes for which app the token is intended. In our case, this matches the Client ID of the application that made the authentication request.
- The claim *iat* shows when the token was issued (seconds since Unix epoch) and can be used to determine the token's age.
- The claim *exp* shows when the token expires (seconds since Unix epoch).

🎉 You have completed Lab 1 by building a web application with sign-on using OpenID Connect! 🎉

Lab 2: Calling an API

This lab covers the process for adding sign-in to a basic Node.js application and calling an API. This lab is the same exercise we provide for new employees in a technical role here at Auth0.

Warning
The Node OIDC and bearer token npm packages that this lab uses has not been tested, licensed, or officially released and should not be used in production.

Prerequisites

- Read the introduction
- Read Using Express Middleware (optional)
- Read Beginner's Guide to Using npm (optional)

What You'll Need

- Node environment - Install Node.js directly or using Homebrew or NVM on a Mac. The labs were tested on Node.js v10.15.0 and NPM 6.4.1 (though they may work in other versions as well).
- An Auth0 account - Sign up for a free Auth0 account here. We recommend starting with a new, empty tenant that can be deleted when you have completed the exercises. If you're using an existing test or dev tenant, make sure that all Rules are turned off and MFA is turned off.
- An Auth0 database user - Use a new username/email and password user in a test database connection instead of a social, enterprise, or passwordless login. While social logins might work, using development keys can cause the labs to run differently. Choose a simple password that's easy to type as you will be logging in and out multiple times. You can use the same user across all of the labs.
- A web browser - This lab was built and tested using Google Chrome; Safari, Firefox, and Edge should all work fine as well. Disable any active ad blockers used for the domain of your local site, as well as for the Auth0 domain from your tenant.
- The Identity Labs Git repo - All the code you need to start, as well as the completed exercise for guidance, is located here. You need to clone that repo just once for all four labs. Use the correct folder relative to the lab you are working on. All file references in this lab are relative to /begin unless otherwise indicated. An /end folder is included as well to help with troubleshooting and compare your work with a working sample.
- For macOS users - If you are new to macOS, check these quick tips for developers new to Mac. Make sure you allow the display of hidden files and become familiar with running basic commands in the terminal.
- For Windows users - We recommend that you use the Windows PowerShell terminal (instead of the Windows command line) so that the terminal commands provided in the lab instructions work as they are. This is because the syntax of the commands used in the labs is the same for the Mac and PowerShell terminals.

Lab 2, Exercise 1: Consuming APIs

After learning how to secure your web application with Auth0 in lab 1, you will now learn how to make this application consume APIs on behalf of your users. You will start by running an unsecured API and a web application to see both working together, and then you will secure your API with Auth0.

1. Open a new terminal and browse to */lab-02/begin/api* in your locally-cloned copy of the identity exercise repo. This is where the code for your API resides. The API is an Express backend that contains a single endpoint. This endpoint (served under the root path) returns expenses, which are data that belong to each user (though they are static and the same for all).

2. Install the dependencies using npm:

```
# Make sure we're in the right directory
❯ pwd
/Users/username/identity-102-exercises/lab-02/begin/api

❯ npm install
# Ignore any warnings

added XX packages in X.XXs
```

3. Next, copy *.env-sample* to *.env* and start the API:

```
❯ cp .env-sample .env
❯ npm start

listening on http://localhost:3001
```

Note: If you see a message like Error: listen EADDRINUSE :::3001 in your terminal after starting the application, this means that port 3001 is in use somewhere. Change the PORT value in your .env file to "3002" and try again.

4. In a new terminal window or tab, navigate to the */lab-02/begin/webapp* directory and install the dependencies using npm:

```
# Navigating from the previous directory
❯ cd ../webapp

# Make sure we're in the right directory
❯ pwd
/Users/username/identity-102-exercises/lab-02/begin/webapp

❯ npm install
# Ignore any warnings

added XX packages in X.XXs
```

5. Once again, copy the .env-sample to .env for the web application:

```
❯ cp .env-sample .env
```

6. Update the web application .env file you just created with the same values as you used in lab 1.

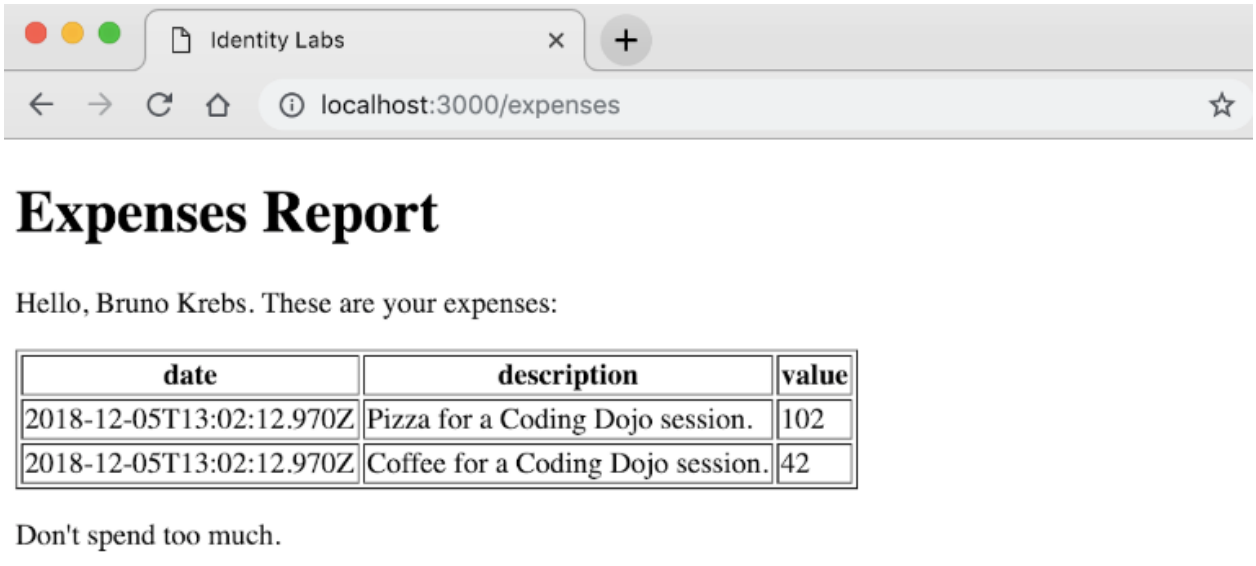
```
ISSUER_BASE_URL=https://YOUR_DOMAIN
CLIENT_ID=YOUR_CLIENT_ID
API_URL=http://localhost:3001
PORT=3000
APP_SESSION_SECRET=LONG_RANDOM_STRING
```

Note: If you changed the port for the API above, make sure to update the API_URL with this new value.

7. Start the web application using npm:

```
❯ npm start

listening on http://localhost:3000
```

8. Open localhost:3000 in your browser. There, you will see the homepage of the web application and, if you log in, you will be able to access the expenses report. The page might look similar to the Lab 1 solution, however, the difference is that an external API provides the Expenses information instead of being hard-coded in the Web app.



Right now, even though the application requires authentication, the API does not. That is, you are calling the API from the Web app, without any authentication information. If you browse to the API's URL at localhost:3001 without logging in, you will see the expenses. In the following steps, you will update your application to call the API with a token.

9. Open webapp/server.js in your code editor and make the following change:

```
// webapp/server.js

app.use(auth({
  required: false,
  auth0Logout: true,
  baseURL: appUrl,

  // Add the additional configuration keys below 👇
  appSessionSecret: false,
  authorizationParams: {
    response_type: 'code id_token',
    response_mode: 'form_post',
    audience: process.env.API_AUDIENCE,
    scope: 'openid profile email read:reports'
  },
  handleCallback: async function (req, res, next) {
    req.session.openidTokens = req.openidTokens;
    req.session.userIdentity = req.openidTokens.claims();
    next();
  },
  getUser: async function (req) {
    return req.session.userIdentity;
  }
  // 👆

}));
```

This change updates the configuration object passed to *auth()* and defines how you want the *express-openid-connect* library to behave. In this case, you configured the library with a new property called *authorizationParams* and passed in an object with three properties:

- **response_type** - setting this field to *code id_token* indicates that you no longer want the middleware to fetch just an ID token (which is the default behavior for this package). Instead, you are specifying that you want an ID token and an authorization code. When you configure the *express-openid-connect* library to fetch an authorization code, the middleware automatically exchanges this code for an access token (this process is known as the Authorization Code Grant flow). Later, you will use the access token to call the API.
- **response_mode** - This is the same mode used in lab 1, a POST request from the authorization server to the application.
- **audience** - this tells the middleware that you want access tokens valid for a specific resource server (your API, in this case). As you will see soon, you will

configure an API_AUDIENCE environment variable to point to the identifier of an API that you will register with Auth0.

- **scope** - securing your API uses a delegated authorization mechanism where an application (your web app) requests access to resources controlled by the user (the resource owner) and hosted by an API (the resource server). Scopes, in this case, are the permissions that the access token grants to the application on behalf of the user. In your case, you are defining four scopes: the first three (*openid*, *profile*, and *email*) are scopes related to the user profile (part of OpenID Connect specification). The last one, *read:reports*, is a custom scope that will be used to determine whether the caller is authorized to retrieve the expenses report from the API on behalf of a user.

The *appSessionSecret*, *handleCallback*, and *getUser* additions change how the user session is handled and stores the incoming access and refresh tokens somewhere we can access later.

10. Back in the *webapp/server.js* file, find the */expenses* endpoint definition. In this code, you are making a request to the API, without any authorization information, to get a JSON resource. Note the use of the requiresAuth() middleware. This will enforce authentication for all requests to this endpoint.

11. Update the endpoint definition to include authorization information in the request:

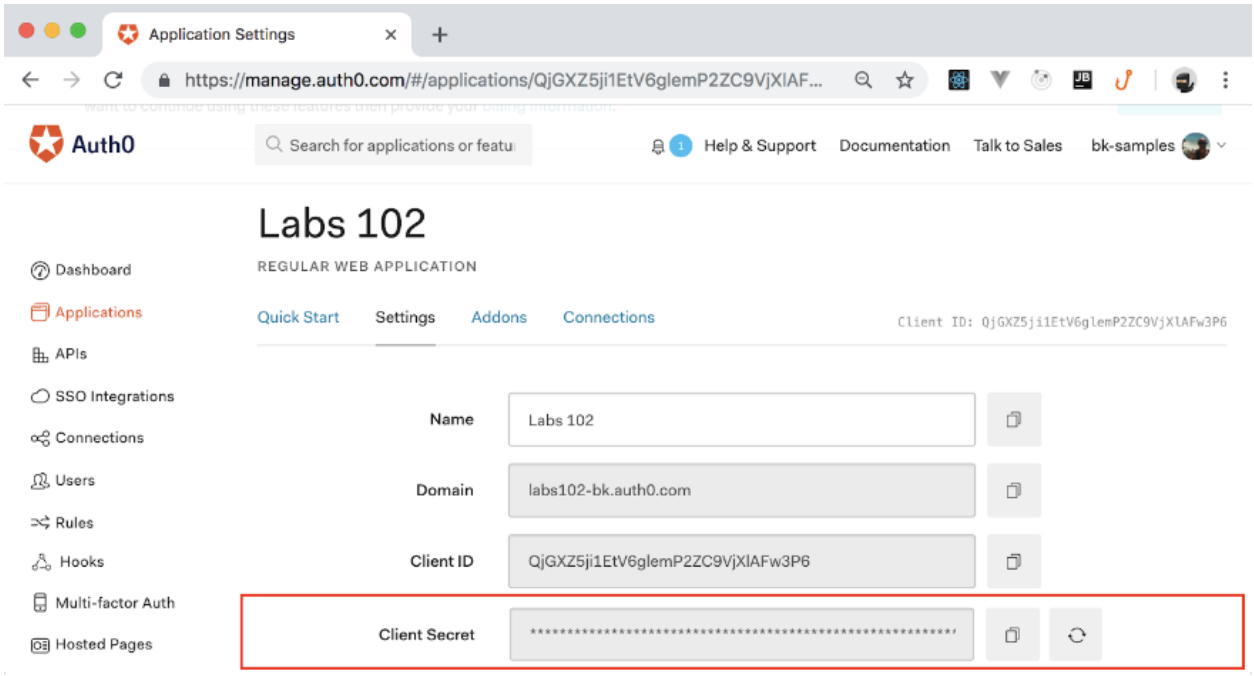
```
// webapp/server.js

app.get('/expenses', requiresAuth(), async (req, res, next) => {
  try {

    // Replace this code ❌
    /*
    const expenses = await request(process.env.API_URL, {
      json: true
    });
    */

    // ... with the code below 👇
    let tokenSet = req.openid.makeTokenSet(req.session.openidTokens);
    const expenses = await request(process.env.API_URL, {
      headers: { authorization: "Bearer " + tokenSet.access_token },
      json: true
    });

    // ... keep the rest
  }
  // ...
});
```

In the new version of this endpoint, you are sending the access token in an *Authorization* header when sending requests to the API. By doing so, the web application consumes the API on behalf of the logged in user.

12. Add the following two environment variables to the *webapp/.env* file:

```
API_AUDIENCE=https://expenses-api
CLIENT_SECRET=YOUR_APPLICATION_CLIENT_SECRET
```

The *API_AUDIENCE* value is the identifier for the API that will be created in the following exercise. To get your Client Secret, go to your Application settings page in the Auth0 Dashboard:



And that's it! You have just configured your web application to consume the API on behalf of the logged in user.

If you restart the application in your terminal, logout, and try to log back in, you will see an error because no resource server with the identifier *https://expenses-api* has been registered yet. In the next exercise, you will learn how to create and secure APIs with Auth0, and this request will begin to work.

Lab 2, Exercise 2: Securing APIs with Auth0

In this exercise, you will register the API with Auth0 so that tokens can be issued for it. You will also learn how to secure your API with Auth0. You will refactor the API that your web application is consuming by installing and configuring some libraries needed to secure it with Auth0.

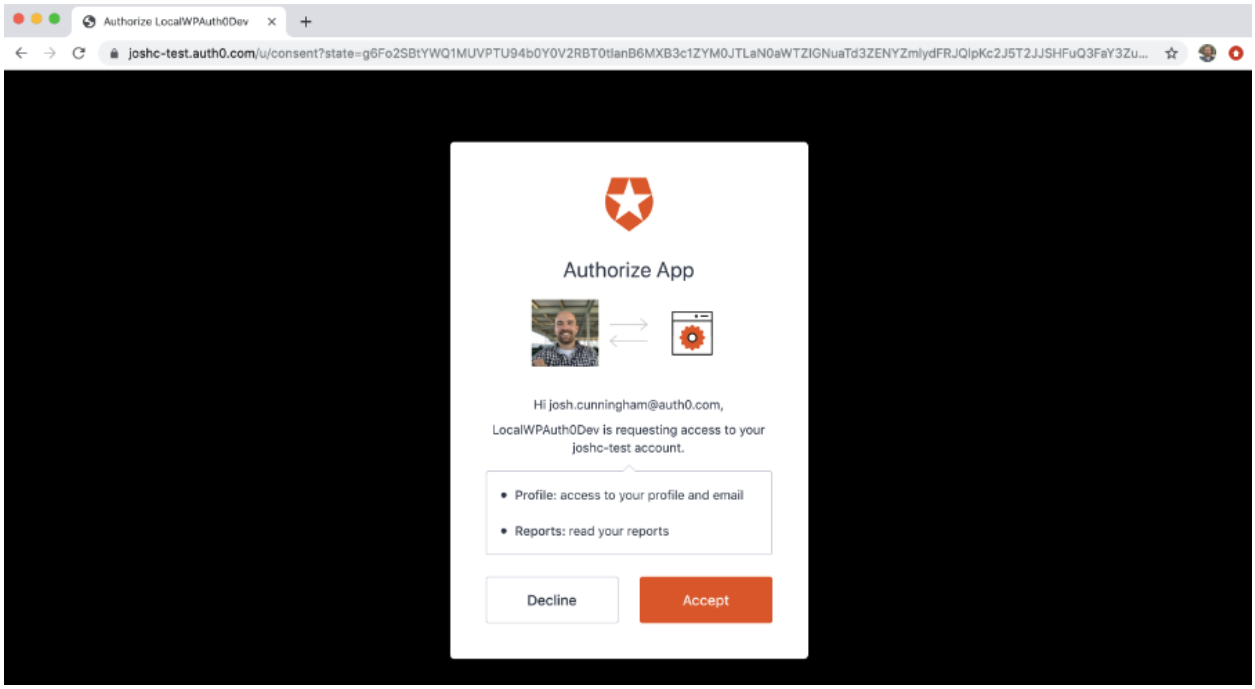
1. To register the API with Auth0, open the Auth0 Dashboard and go to the APIs screen.

2. Click the Create API button. Add a descriptive Name, paste *https://expenses-api* into the Identifier field, and click Create.

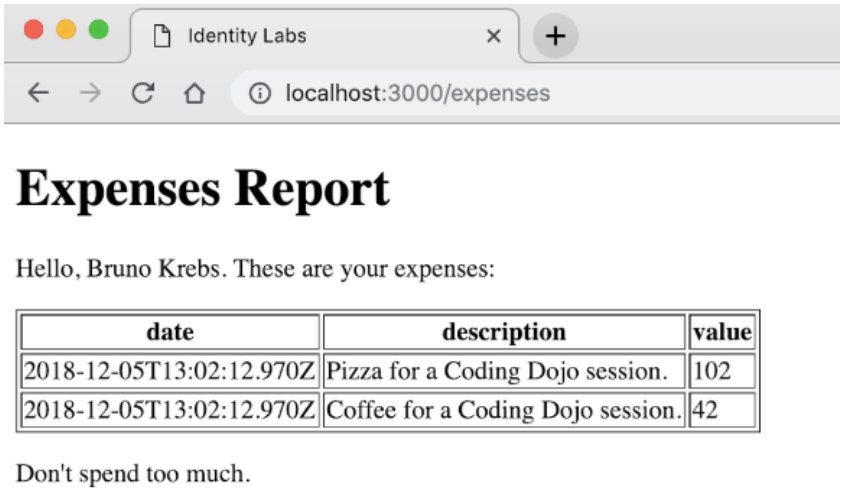
3. Click the Permissions tab and add a new permission called *read:reports* with a suitable description. This custom permission is the one you will use to determine whether the client is authorized to retrieve expenses.

4. In your terminal, restart your web application with *[CTRL] + [c]*, then *npm start*.

5. Log out of the web application by clicking logout, then log in again. When logging in, you will see a consent screen where Auth0 mentions that the web application is requesting access to the read:reports scope:



6. Agree to this delegation by clicking the Accept button, and Auth0 will redirect you back to the application. Now, you should still be able to see your expenses on the expenses page, localhost:3000/expenses:



Note: If at any point, you want to see the consent screen again when logging in, you can go to the Users screen in the Auth0 Dashboard, click on the user you'd like to modify, click the Authorized Applications tab, find the application you're using, and click Revoke. The next time you log in, the consent screen will appear again.

As mentioned earlier, the expenses API is still not secure. You can see this by navigating directly to localhost:3001. The expense data is available publicly, without an access token. The next steps will change the API to require a properly-scoped access token to view.

7. In your terminal, stop your API with [*CTRL*] + [*c*].

8. Install the *express-oauth2-bearer* npm package. This is an Express authentication middleware used to protect OAuth2 resources, which validates access tokens:

```
# Make sure we're in the right directory
❯ pwd
/Users/username/identity-102-exercises/lab-02/begin/api

❯ npm install express-oauth2-bearer@0.4.0 --save
# Ignore any warnings

+ express-oauth2-bearer@0.4.0
added XX packages in X.XXs
```

9. Open the *api/api-server.js* file and add a statement to import the library. Make sure this is added after the dotenv require statement:

```
// api/api-server.js

require('dotenv').config();
// ... other require statements

// Add the line below 👇
const { auth, requiredScopes } = require('express-oauth2-bearer');
```

10. Configure the Express app to use the authentication middleware for all requests:

```
// api/api-server.js

// ... other require statements
const app = express();

// Add the line below 👇
app.use(auth());
```

11. Find the */* endpoint code and update it to require the *read:reports* scope in access tokens. This is done by adding a *requiredScopes* middleware, as shown below:

```
// lab-02/begin/api/api-server.js

// Change only the line below 👇
app.get('/', requiredScopes('read:reports'), (req, res) => {

    // ... leave the endpoint contents unchanged.

});
```

The next time you run your API, all requests that do not include a valid access token (expired token, incorrect scopes, etc.) will return an error instead of the desired data.

12. Open the api/.env file you created before and change the ISSUER_BASE_URL value to your own Auth0 base URL (same as the one in your application). The .env file should look like this:

```
PORT=3001
ISSUER_BASE_URL=https://your-tenant-name.auth0.com
ALLOWED_AUDIENCES=https://expenses-api
```

13. Once again, start the API server with npm:

```
❯ npm start

listening on http://localhost:3001
```

To test your secured API, refresh the expenses page in your application - localhost:3000/expenses. If everything works as expected, you will still be able to access this view (which means that the web app is consuming the API on your behalf). If you browse directly to the API at localhost:3001, however, you will get an error saying the token is missing.

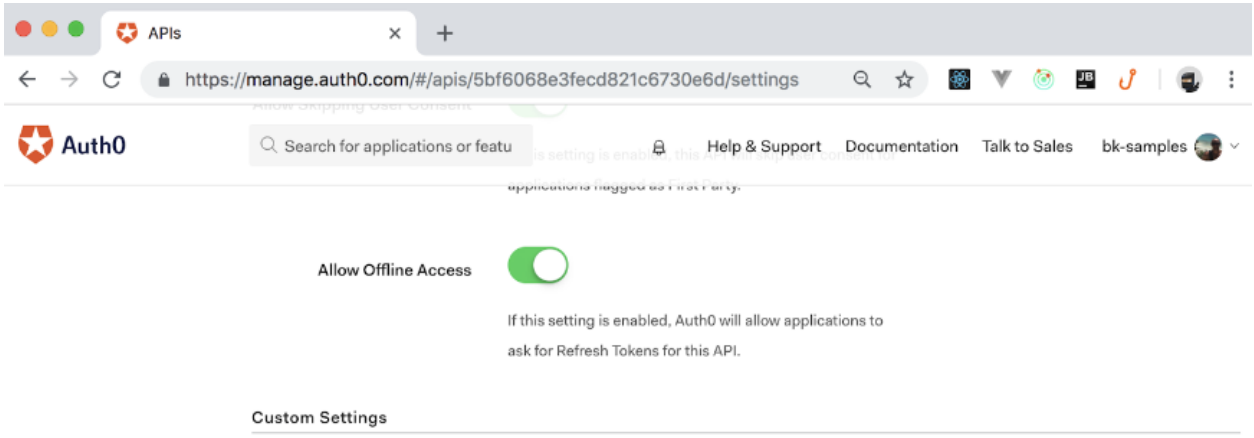
Lab 2, Exercise 3: Working with Refresh Tokens

Right now, if your users stay logged in for too long and try to refresh the /expenses page, they will face a problem. Access tokens were conceived to be exchanged by different services through the network (which makes them more prone to leakage), so they should expire quickly. When an access token is expired, your API won't accept it anymore, and your web application won't be able to fetch the data needed. A token expired error will be returned instead.

To change this behavior, you can make your web app take advantage of yet another token: the refresh token. A refresh token is used to obtain new access tokens and/or ID tokens from the

authorization server. In this exercise, we're going to modify the application to obtain a refresh token and use it to get a new access token when it expires.

1. Navigate to the APIs screen in your Auth0 Dashboard and open the API created in the last exercise. Scroll down, turn on the Allow Offline Access option, and click Save:



2. Now, Open the *webapp/server.js* file and add *offline_access* to the *authorizationParams.scope* field passed to the *auth() middleware*:

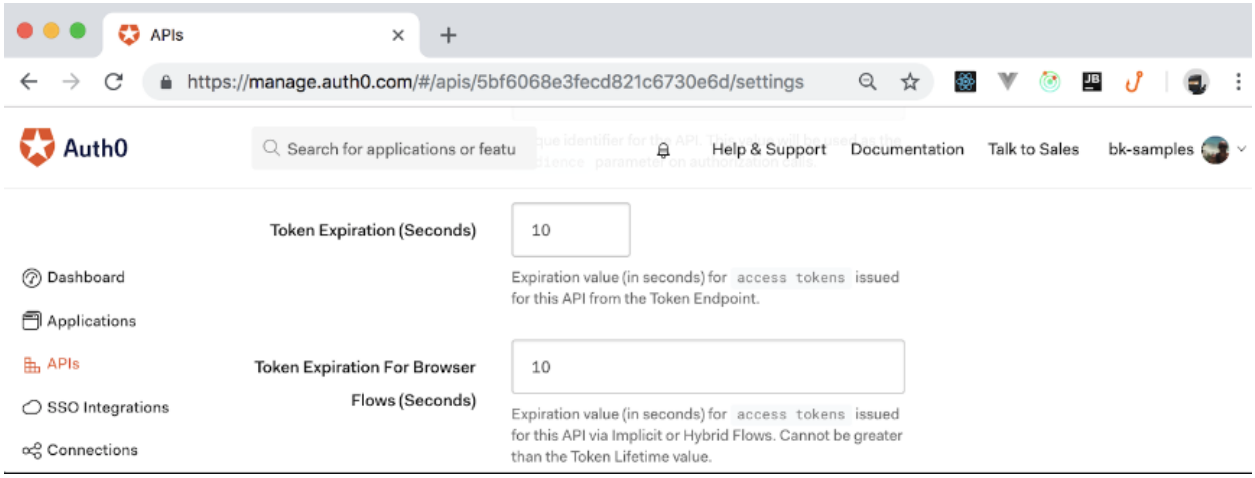
```
// webapp/server.js

app.use(auth({
  required: false,
  auth0Logout: true,
  appSessionSecret: false,
  authorizationParams: {
    response_type: 'code id_token',
    response_mode: 'form_post',
    audience: process.env.API_AUDIENCE,

    // Change only the line below 👇
    scope: 'openid profile email read:reports offline_access'

  },

  // ... keep the rest

}));
```

3. Now, find the following line in the /expenses endpoint code and replace it with the following:

```
// webapp/server.js

app.get('/expenses', requiresAuth(), async (req, res, next) => {
  try {

    let tokenSet = req.openid.makeTokenSet(req.session.openidTokens);

    // Add the code block below 👇
    if (tokenSet.expired()) {
      tokenSet = await req.openid.client.refresh(tokenSet);
      tokenSet.refresh_token = req.session.openidTokens.refresh_token;
      req.session.openidTokens = tokenSet;
    }

    // ... keep the rest
  }
  // ...
});
```

This change will update your endpoint to check if the *tokenSet* is expired. If it is, the Issuer class will create a client that is capable of refreshing the *tokenSet*. To see the refreshing process in action, you will have to make a small change to your Auth0 API configuration.

4. Navigate to the APIs screen in your Auth0 Dashboard and open the API created in the last exercise. Set both the Token Expiration (Seconds) and Token Expiration For Browser Flows (Seconds) values to 10 seconds or less and click Save:



5. Back in your editor, add a log statement to *api/api-server.js* to show when the new access token was issued:

```
// api/api-server.js

app.get('/', requiredScopes('read:reports'), (req, res) => {

  // Add the line below 👇
  console.log(new Date(req.auth.claims.iat * 1000));

  // ...
});
```

6. Restart both the application and API ([*CTRL*] + [*c*], then *npm start*).

7. Log out and log in again. This will get you a complete set of tokens (ID token, access token, and refresh token). Note, at this point, you will see a new consent screen for the offline_access scope, which you need to accept.

Open localhost:3000/expenses in your browser and refresh the page. You will see that your API logs a timestamp in the terminal. The same timestamp will be logged every time you refresh the page as long as your token remains valid. Then, if you wait a few seconds (more than ten) and refresh the view again, you will see that your API starts logging a different timestamp, which corresponds to the new token retrieved. This shows that you are getting a different access token every ten seconds and that your web application uses the refresh token automatically to get them.

Note: If you see an error in your console about an ID token used too early, this is likely a clock skew issue in your local environment. Try restarting your machine and walking through the login steps again from the beginning. You can also try going to "Date & Time" settings, unlock them if needed by clicking on the lock icon at the bottom, and disable and re-enable the "Set date and time automatically" option.

Note: If you don't see changes in the "Issued At" claim in the console, make sure you have logged out and logged in again after applying the changes above.

Note: If you are using PowerShell in Windows and you see blank lines instead of the timestamp logging in the terminal, it could be the font color of the logs is the same as the background. As an alternative, you can run the API server from the Windows command line, or change the background color in PowerShell.

🎉 You have completed Lab 2 by building a web application that calls an API with refresh capability! 🎉

Lab 2, Exercise 4: Step Up Authentication

Resources

[StepUp Authentication for Web Apps]

Flow

1. The user logs in to the application with an email and password
2. When the user clicks the expenses link, the app will
    a. Check to see if the user has already performed MFA by looking for the amr claim in the ID Token

```
{
   "iss": "https://YOUR_DOMAIN/",
   "sub": "auth0|1a2b3c4d5e6f7g8h9i",
   "aud": "YOUR_CLIENT_ID",
   "iat": 1522838054,
   "exp": 1522874054,
   "acr": "http://schemas.openid.net/pape/policies/2007/06/multi-factor",
   "amr": [
      "mfa"
   ]
}
```

    b. If the amr claim is missing or does not contain ‘mfa“, the user will be redirected to /authorize with the additional scope read:reports
3. After performing MFA, the user will be redirected back to the expenses route with a new id and access token.
    a. The Id Token will have the amr claim with the mfa value
    b. The Access Token will have the read:reports scope
4. The app will then check if the access token is expired and refresh if needed, then call the API to retrieve the expenses

Steps

1. Enable any type of MFA in your tenant
2. Make a copy of the completed project from lab 2 and name it StepUp
3. Open `webapp/server.js` in your code editor and make the following change
    a. Remove the read:reports scope from existing authorizationParams object

```
app.use(
  auth({
    baseURL: appUrl,
    required: false,
    auth0Logout: true,
    routes: false,
    appSession: false,
    authorizationParams: {
      response_type: "code id_token",
      response_mode: "form_post",
      audience: process.env.API_AUDIENCE,
      scope: "openid profile email read:reports offline_access",
    },
    handleCallback: async function (req, res, next) {
      req.session.openidTokens = req.openidTokens;
      req.session.userIdentity = req.openidTokens.claims();
      next();
    },
    getUser: async function (req) {
      return req.session.userIdentity;
    },
  })
);
```

b. Under the expenses route, add a condition to see if the id token has the amr claim and it's set to mfa. If either condition is false, stepup MFA should be triggered

c. Inside of the condition block, define an object named stepUpAuthorizationParams. The object will look similar to the authorizationParams defined earlier except it will have the read:reports scope

d. Now, trigger the redirect to /authorize by calling the login function on the auth SDK. This function takes the stepUpAuthorizationParams and the /expenses route.
   i. These stepUpAuthorizationParams are added as query parameters to the /authorize redirect
   ii. By setting returnTo = "/expenses", we are telling the API to redirect us back here after authentication is complete

```
if (req.openid.user.amr === undefined || req.openid.user.amr.includes('mfa') === false) {

  const stepUpAuthorizationParams = {
    response_type: "code id_token",
    response_mode: "form_post",
    audience: process.env.API_AUDIENCE,
    scope: "openid profile email read:reports offline_access",
  };

  res.openid.login({ authorizationParams: stepUpAuthorizationParams, returnTo: "/expenses" })
}
```

e. Next, wrap the existing logic (under the expenses route, up to } catch) in an else
    i. The expenses API should only be called if MFA has been completed

```
    res.openid.login({ authorizationParams: stepUpAuthorizationParams, returnTo: "/expenses" })
  }
  else {

    let tokenSet = req.openid.makeTokenSet(req.session.openidTokens);

    if (tokenSet.expired()) {
      tokenSet = await req.openid.client.refresh(tokenSet);
      tokenSet.refresh_token = req.session.openidTokens.refresh_token;
      req.session.openidTokens = tokenSet;
    }

    const expenses = await request(process.env.API_URL, {
      headers: { authorization: "Bearer " + tokenSet.access_token },
      json: true,
    });

    res.render("expenses", {
      user: req.openid && req.openid.user,
      expenses,
    });
  }
 } catch (err) {
  next(err);
 }
});
```

4. Add an action or rule to trigger MFA when the read:reports scope is requested. Do not add both, otherwise they will both run.

Sample Login Action

```
exports.onExecutePostLogin = async (event, api) => {

  if (event.request.query.scope.split(" ").includes('read:reports')) {
    api.multifactor.enable("any");
  };

};
```

Sample Rule

```
function (user, context, callback) {

  if(context.request.query.scope.split(" ").includes('read:reports')) {
       context.multifactor = {
      provider: 'any',
      allowRememberBrowser: false
    };
  }
  return callback(null, user, context);
}
```

Lab 3: Mobile Native App

Prerequisites

- Apple's Developing iOS Apps: Build a Basic UI tutorial to get a feeling of the Xcode's UI (optional)

What You'll Need

- A Mac computer - A Mac is required to install Xcode.
- An Apple account - Required to download Xcode and install Xcode.
- Xcode - Download and install Xcode from the App Store. After installation is complete, open it so that you go through the first-time setup, which can take up to 10 minutes. This will require around 6+GB of hard drive space and up to 30 minutes total to complete.

Lab 3, Exercise 1: Adding Authentication

In this exercise, you will add authentication to an existing iOS application. A simple iOS application has been provided to get you started. This is a single-view application with a button to launch the Auth0 authentication process.

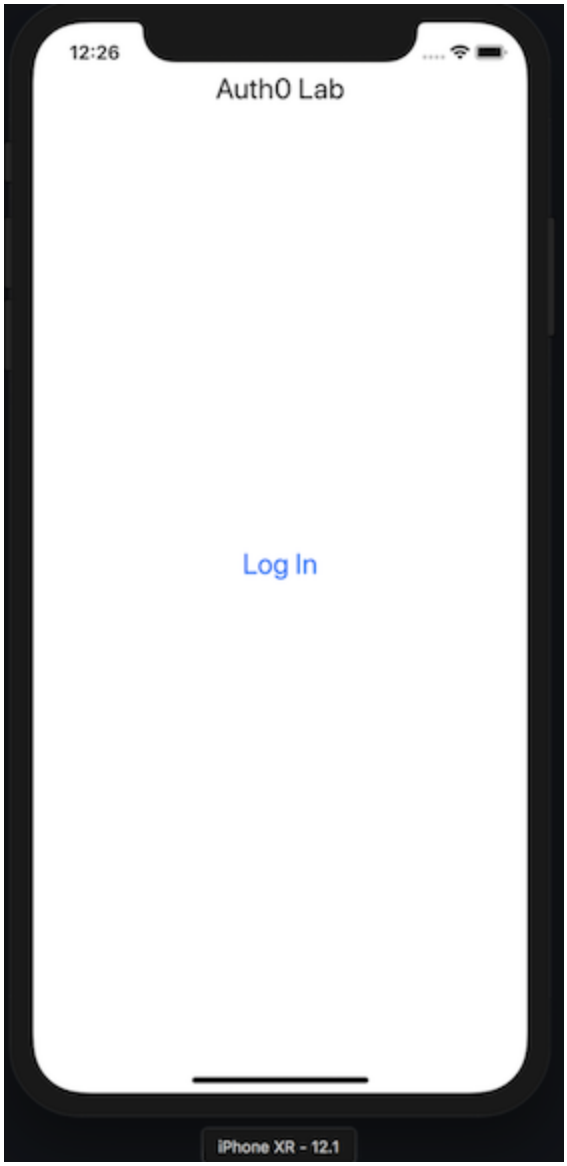
1. Launch Xcode, go to File > Open, and open */lab-03/exercise-01/begin/exercise-01.xcworkspace* in your locally-cloned copy of the identity exercise repo.

Note: If the project complains about a missing dependency, you might have opened exercise-01.xcodeproj instead of exercise-01.xcworkspace (note the extension).

This project is a bare-bones application that imports the Auth0.swift dependency to provide the OpenID Connect implementation. There is also a stub method called actionLogin for processing the touch of the login button.

2. In the bar at the top of the project window, click the device selector and pick a late-model iPhone, then click the Play button (or Product > Run from the Xcode menu) to run the app.

The simulator may take a few moments to load the first time, and then you should see the following:



3. Touch the Log In button. This will output a "Log In" message to the Debug area in Xcode. If you don’t see the Debug view, you can enable it with View > Debug Area > Show Debug Area.

4. Before any calls are made to the Auth0 authorization server, you need to set up a new Auth0 Application for handling Native Applications. Log into the Auth0 Dashboard, go to the Applications page, and click the Create Application button.

5. Enter a descriptive name, select Native as the application type, and click Create.

6. Click on the Settings tab and scroll down to the Allowed Callback URLs field. Enter the value below (modified with your tenant domain):

```
com.auth0.identity102://${account.namespace}/ios/com.auth0.identity102/callback
```

7. Scroll down and click Show Advanced Settings, then OAuth. Make sure JsonWebToken Signature Algorithm is set to *RS256*.

8. Click Save Changes

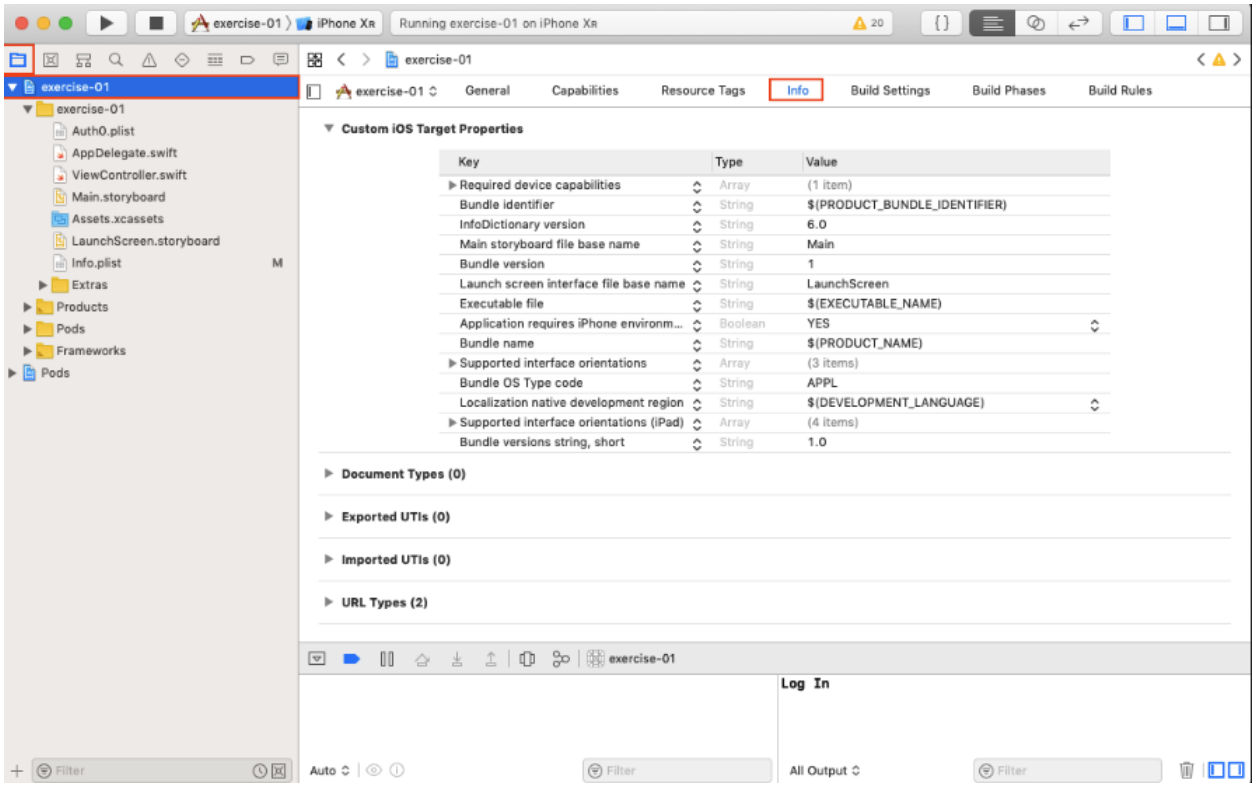
- You might be wondering why the callback URL is in this format. There are two parts to this: The first element is the scheme of the application, which for the purposes of this exercise, is defined as com.auth0.identity102. Whenever Safari needs to handle a request with this scheme, it will route it to our application (you will set up this custom URL scheme URL later in the lab).
- The rest of the URL is in a format that the Auth0.swift SDK specifies for callbacks.

9. Now the sample iOS application needs to be configured with the Client ID and Domain values from the Auth0 Application. Return to Xcode and open the exercise-01/Auth0.plist file. You should see value placeholders for ClientId and Domain. Replace these with the values from the Auth0 Application created above.

Note: The domain must not have any prefix like in the previous labs. Enter it exactly as it is provided in the Auth0 dashboard.

To be able to use the callback that was configured in the Auth0 dashboard, a URL scheme handler needs to be registered in our iOS application so that it can respond to requests made to the callback URL.

10. In the file navigator on the left, click on *exercise-01* to open the project settings, then click on the Info tab.



11. Scroll down to URL Types, expand the section, click the + button, and enter or select the following details:
    - **Identifier**: auth0
    - **URL Schemes**: $(PRODUCT_BUNDLE_IDENTIFIER)
    - **Role**: None

Just as *http* is a URL Scheme that will launch a browser, the bundle identifier of the app has a URL Scheme (which will resolve to *com.auth0.identity102*) will tell iOS that any time this scheme is used in a URL, it must be routed to our application. That will be the case of the callback used by Auth0 after you log in.

12. Now, the application needs to have the Auth0.swift SDK handle the callback in order to proceed with the authentication flow. In the Project Navigator on the left, open *exercise-01/AppDelegate.swift* and add the following import statement just below the other one:

```swift
// exercise-01/AppDelegate.swift

import UIKit

// Add the line below 👇
import Auth0
```

13. In the same file, add the following method inside the *AppDelegate* class:

```swift
// exercise-01/AppDelegate.swift
// ...
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    // Add the code below 👇
    func application(_
        app: UIApplication,
        open url: URL,
        options: [UIApplication.OpenURLOptionsKey : Any]
        ) -> Bool {
        return Auth0.resumeAuth(url, options: options)
    }

    // ... other existing methods
}
```

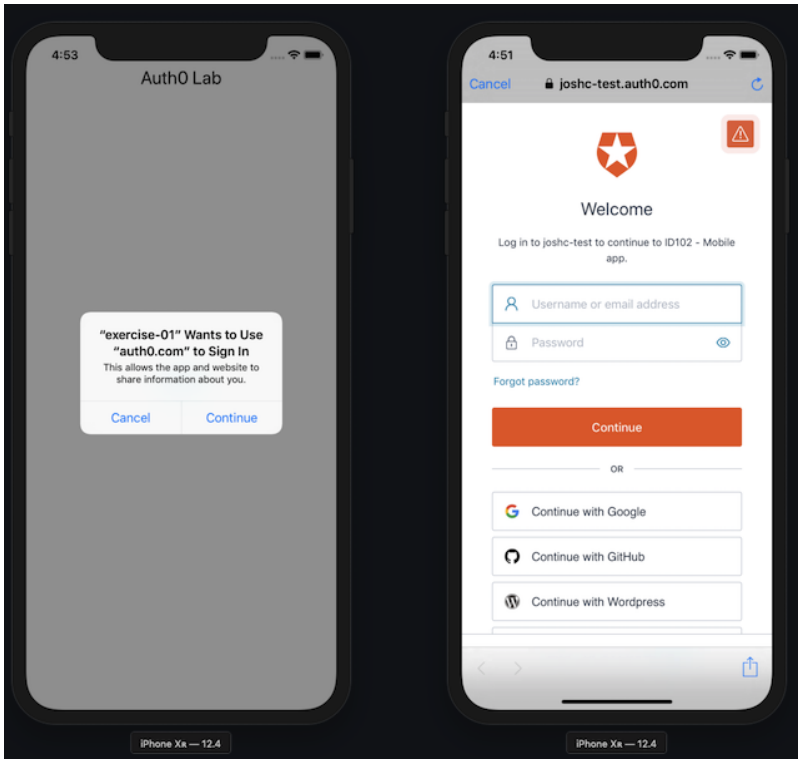
When another app requests a URL containing the custom scheme, the system will launch your app if necessary, bring it to the foreground, and call the method above. The iOS Framework provides the delegate method above for you to implement so that you can parse the contents of the URL and take appropriate action. In this case, you need this information to continue the authentication flow process. You will see later in this exercise why this step is needed.

Now that the iOS application is configured with your Auth0 application credentials and is able to receive and process callbacks, complete the following steps to see how to construct the OpenID Connect request to the authorization server.

14. Open *exercise-01/ViewController.swift* and add the following code inside the *actionLogin* method, after the line that prints the "Log In" message to the console:

```swift
// exercise-01/ViewController.swift
// ...
  @IBAction func actionLogin(_ sender: Any) {
      print("Log In")

      // Add the code below 👇
      Auth0
        .webAuth()
        .scope("openid profile email")
        .logging(enabled: true)
        .start { response in
            switch(response) {
                case .success(let result):
                    print("Authentication Success")
                    print("Access Token: \(result.accessToken ?? "No Access Token Found")")
                    print("ID Token: \(result.idToken ?? "No ID Token Found")")
                case .failure(let error):
                    print("Authentication Failed: \(error)")
                }
          }
  }
// ...
```

15. Run the app again by clicking the Play button (or Product > Run from the Xcode menu). Once the app has launched, touch the Log In button. You should see a permission prompt from iOS. Touch Continue to proceed to the Auth0 login page, which is rendered within a browser.



16. Log in using your database user, and you will be taken back to the app. Nothing will have changed visually, but if you take a look at the Debug Area in Xcode you will see something like this:

```
Authentication Success
Access Token: vxPp0Xtg3wkZJudFZWzqMQByYF98Qyer
ID Token: eyJ0eX[..].eyJodH[..].kLtZDg[..]
```

To view the contents of your ID Token, you can copy and paste it into jwt.io to view the claims.

Now that you have an ID token, it's important to validate it to ensure that it can be trusted. A helper method isTokenValid is already included in the project; you can review its code in Extras/Utils.swift to learn how the validation is performed. It should be called after obtaining the token, to illustrate how it is used.

17. Back in the actionLogin method in ViewController.swift, add the line below:

```swift
// exercise-01/ViewController.swift
// ...

  @IBAction func actionLogin(_ sender: Any) {
      print("Log In")

      Auth0
        // ...
              case .success(let result):
                  // ... other print statements

                  // Add the line below 👇
                  print("ID Token Valid: \(isTokenValid(result.idToken!))")

              // ... failure case
      }
  }
// ...
```

18. Run the app again, log in, and take a look at the logs in Xcode. You should see an entry "ID Token Valid:" with the status of the validation (true or false).

Congratulations! You have successfully added Auth0 authentication to your native iOS app using an authorization code grant!

The authorization code grant by itself has some security issues when implemented on native applications. For instance, a malicious attacker can intercept the authorization code returned by

Auth0 and exchange it for an access token. The Proof Key for Code Exchange (PKCE) is a technique used to mitigate this authorization code interception attack.

With PKCE, for every authorization request, the application creates a cryptographically random key called the code verifier, hashes that value into a code challenge, and sends the code challenge to the authorization server to get the authorization code. When the application receives the code after a successful login, it will send the code and the code verifier to the token endpoint to exchange them for the requested tokens.

Since you previously enabled logging in our *WebAuth* call with the *logging()* method, it is easy to see the process flow in the Debug Area.

19. Run the iOS Application, touch the Log In button, and then take a look at the Debug Area. The iOS application initiates the flow and redirects the user to the */authorize* endpoint, sending the *code_challenge* and *code_challenge_method* parameters. It also sends a *response_type* of *code* (line breaks added below for readability):

```
SafariAuthenticationSession:
https://${account.namespace}/authorize
?code_challenge=VsPaQ0gJjnluA2vwV0piY-D-DTCltGI9GbYkBNHvPHQ
&response_type=code
&redirect_uri=com.auth0.identity102://${account.namespace}/ios/com.auth0.identity102/callback
&state=RFnNyPj4NOZMUW8IpDBr-j3UgO4gCbhBZtLpWB_vmDo
&client_id=${account.clientId}
&scope=openid%20profile
&code_challenge_method=S256
&auth0Client=eyJzd2lmdC12ZXJzaW9uIjoiMy4wIiwibmFtZSI6IkF1dGgwLnN3aWZ0IiwidmVyc2lvbiI6IjEuMTMuMCJ9
```

20. Once again, enter your credentials and log in. Auth0 redirects the user back to the iOS application by calling the callback with the authorization code in the query string:

```
iOS Safari:
com.auth0.identity102://${account.namespace}/ios/com.auth0.identity102/callback
?code=6SiMHrJHbG2aAPrj
&state=RFnNyPj4NOZMUW8IpDBr-j3UgO4gCbhBZtLpWB_vmDo
```

21. The Auth0.swift SDK will process the query string and send the authorization *code* and *code_verifier* together with the *redirect_uri* and the *client_id* to the token endpoint of the authorization server:

```
POST /oauth/token

{"grant_type":"authorization_code",
"redirect_uri":"com.auth0.identity102:\/\/${account.namespace}\/ios\/com.auth0.identity102\/callback",
"code":"6SiMHrJHbG2aAPrj",
"code_verifier":"qiV8gYUrPco3qBlejLeZzgC9DMtXZY1GddzZpmVxyxw",
"client_id":"${account.clientId}"}
```

22. The authorization server validates this information and returns the requested access and ID tokens. If successful, you will see the following response containing your tokens:

```
Content-Type: application/json

{"access_token":"ekhGPSE7xdhOTJuTo2dV-TYyJV-OTYrO",
"id_token":"eyJ0eX[..].eyJodH[..].1kZccn[..]",
"expires_in":86400,
"token_type":"Bearer"}
```

In the next exercise, you will use a token to validate and authorize the user and authorize against a protected API.

Lab 3, Exercise 2: Calling a Secured API

In this exercise, you are going to enable the native application to authorize against the protected API backend that was built in Lab 2, Exercise 2. In that lab, you set up an Auth0 API server for your Expenses API with an audience value of https://expenses-api.

If you have already completed lab 2, you can use the same Auth0 configuration and local files to run the API needed for this lab. Just go to /lab-02/begin/api in your locally-cloned copy of the identity exercise repo and run npm start in your terminal before beginning this exercise. Make sure your token expiration times in Auth0 are back to normal (at least an hour for both).

1. Go to */lab-02/end/api* and run *npm install* in your terminal.

2. Follow steps 1-3 to create an API in Auth0.

3. Create a copy of the *.env* file in the same directory as above, change the *ISSUER_BASE_URL* value to include your tenant name, and save the file.

4. Back in the terminal, run *npm start*.

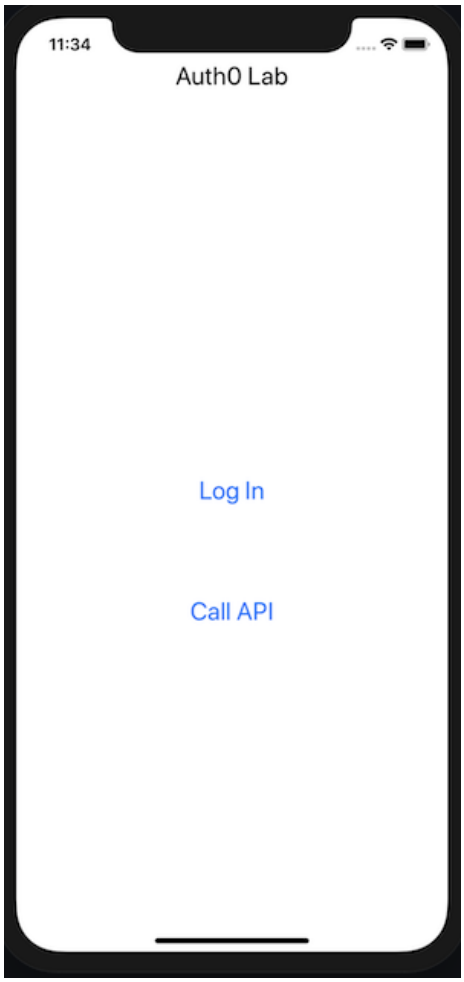
```
# Starting from the Lab 3 begin folder...
❯ cd ../../../lab-02/end/api

❯ pwd
/Users/username/identity-102-exercises/lab-02/end/api

❯ cp .env-sample .env

❯ vim .env
# Change the ISSUER_BASE_URL value ...

❯ npm install

added XX packages in X.XXs

❯ npm start

listening on http://localhost:3001
```

Regardless of which API codebase you're using, you should now be able to load localhost:3001 in your browser and see an error saying *UnauthorizedError: bearer token is missing.*

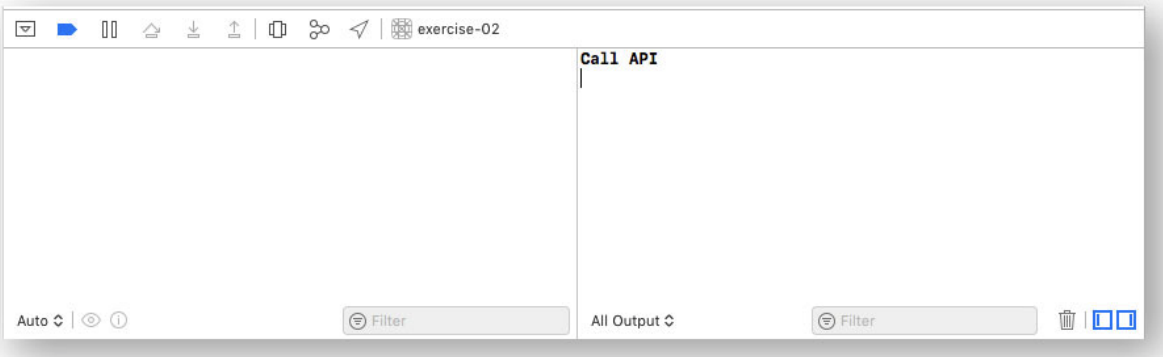
5. For this exercise, we're going to open a different project in Xcode than the one we used in exercise 1. Go to File > Open in Xcode and select *lab-03/exercise-02/begin/exercise-02.xcworkspace* (make sure you pick the right file extension), then open *exercise-02/ViewController.swift*. This code picks up where the previous exercise left off and adds a new button to call the API.

6. Open the *exercise-02/Auth0.plist* file and replace the placeholder values for ClientId and Domain with the ones from the Auth0 Application created before.

7. Click the Play button (or Product > Run from the Xcode menu) to run the app.



8. Touch the Call API button, and you should see a "Call API" message in the Debug area in Xcode.



You will now add code to make the API call from the mobile app. However, before doing so, you need to modify the authentication code to include the API's audience for

authorization and the necessary scopes so that the required permissions are requested.

9. In the *actionLogin* method, which contains our authentication call, include the audience for the API we want to access. With this in place, there will be an additional audience inside the access token after successful authentication.

```
// exercise-02/ViewController.swift
// ...

    @IBAction func actionLogin(_ sender: Any) {
        Auth0
            .webAuth()
            .scope("openid profile")

            // Add the line below 👇
            .audience("https://expenses-api")

            // ...
        }
    }
// ...
```

10. Run the app from Xcode again, click Log In, and check the debug logs. You should see a block of output like below:

```
Authentication Success
Access Token: eyJ0eXA[..].eyJpc3[..].XeiZaS[..]
ID Token: eyJ0eXA[..].eyJodH[..].Lv1TY8[..]
Token Valid: true
```

11. Copy and paste the value of the Access Token into jwt.io. Notice the *scope* value of *openid profile*. In Lab 2, the additional scope read:reports was added, which is not present in the token yet:

```
{
  "iss": "https://${account.namespace}/",
  "sub": "auth0|1234567890",
  "aud": [

    // New audience 👇
    "https://expenses-api",
    "https://${account.namespace}/userinfo"
  ],
  "iat": 1566840738,
  "exp": 1566840746,
  "azp": "${account.clientId}",

  // Existing scopes 👇
  "scope": "openid profile"
}
```

12. Now, add the *read:reports* scope to the parameter in the *scope()* method within *actionLogin*:

```
// exercise-02/ViewController.swift
// ...

  @IBAction func actionLogin(_ sender: Any) {
      Auth0
          .webAuth()

          // Replace this line ❌
          // .scope("openid profile")

          // ... with the line below 👇
          .scope("openid profile read:reports")

          // ...
      }
  }
```

13. Run the app again, log in, and check the access token in jwt.io once more. You should now see the read:reports scope in the payload. It's time to make a call to the API!

14. To use the access token we obtained during login in the actionAPI method, you need a way to access this variable. Create a private variable in the ViewController class:

```swift
// exercise-02/ViewController.swift
// ...
import Auth0

class ViewController: UIViewController {

    // Add the line below 👇
    private var accessToken: String?

    // ...
}
```

15. In the .success code block of the actionLogin method, set the new accessToken value to be what was returned from the token endpoint:

```swift
// exercise-02/ViewController.swift
// ...
    @IBAction func actionLogin(_ sender: Any) {
        // ...
                    case .success(let result):
                        // ...

                        // Add the line below 👇
                        self.accessToken = result.accessToken

                    case .failure(let error):
                        // ...
    }
// ...
```

16. In the actionAPI method in the same class, check that the user has authenticated and that you have an access token before making a call to the API:

```swift
// exercise-02/ViewController.swift
// ...
    @IBAction func actionAPI(_ sender: Any) {
        print("Call API")

        // Add the code below 👇
        guard let accessToken = self.accessToken else {
            print("No Access Token found")
            return
        }
    }
// ...
```

Here, you are assigning the class-scoped property accessToken to a local accessToken variable. If the class-scoped property is empty, an error will be returned.

17. Here, you are assigning the class-scoped property accessToken to a local accessToken variable. If the class-scoped property is empty, an error will be returned.

```swift
// exercise-02/ViewController.swift
// ...
    @IBAction func actionAPI(_ sender: Any) {
        // ... code from above

        // Add the code below 👇
        let url = URL(string: "http://localhost:3001")!
        var request = URLRequest(url: url)
    }
// ...
```

Note: If your API is running on a different port or URL, make sure to change that above.

18. You also need a way to send the access token to the API. This is done by adding an HTTP Authorization request header:

```swift
// exercise-02/ViewController.swift
// ...
    @IBAction func actionAPI(_ sender: Any) {
        // ... code from above

        // Add the code below 👇
        request.addValue("Bearer \(accessToken)", forHTTPHeaderField: "Authorization")
        request.log()
    }
// ...
```

19. Finally, the request needs to be executed. You will use the functionality built into the iOS framework - URLSession - to perform the network operation:

```swift
// exercise-02/ViewController.swift
// ...
    @IBAction func actionAPI(_ sender: Any) {
        // ... code from above

        // Add the code below 👇
        let task = URLSession.shared.dataTask(with: request) {
            data, response, error in
                print(response ?? "No Response")
        }
        task.resume() // Execute the request
    }
// ...
```

20. Let's try calling the API from our mobile app. Save your changes from above, run the app, and tap Log In. After successfully authenticating, tap the Call API button and check the logs in the Debug area for the API response:

```
Call API
GET http://localhost:3001
Headers:
 Optional(["Authorization": "Bearer eyJ0eX[..].eyJpcM[..].dpN8sK[..]"])
<NSHTTPURLResponse: 0x6000010dfdc0> { URL: http://localhost:3001/ } { Status Code: 200, Headers {
    Connection =     (
        "keep-alive"
    );
    "Content-Length" =     (
        195
    );
    "Content-Type" =     (
        "application/json; charset=utf-8"
    );
    Date =     (
        "Tue, 27 Aug 2019 14:53:40 GMT"
    );
    Etag =     (
        "W/\"c3-oBamo6wQLwSzwYwQczXJ+w5tl5o\""
    );
    "X-Powered-By" =     (
        Express
    );
} }
```

21. The Status Code: 200 (OK) lets us know the request was executed successfully. If you want to see it fail, simply comment out the line that adds the Authorization Bearer header, re-rerun the app, and try logging in again. You will see a Status Code: 401 (Unauthorized).

```swift
// exercise-02/ViewController.swift
// ...
    @IBAction func actionAPI(_ sender: Any) {
        // ... code from above

        let task = URLSession.shared.dataTask(with: request) {
            data, response, error in
                print(response ?? "No Response")

                // Add the code below 👇
                if let data = data {
                    print(String(data: data, encoding: .utf8) ?? "No Body")
                }
        }
        // ...
    }
// ...
```

22. Re-run the app, login, and call the API once more. You should now see the expenses in the debug area in Xcode:

```
[{"date":"2019-08-27T15:02:04.838Z","description":"Pizza for a Coding Dojo session.","value":102},
{"date":"2019-08-27T15:02:04.838Z","description":"Coffee for a Coding Dojo session.","value":42}]
```

    You have now integrated your native application frontend with a protected API backend! In the next exercise, you will look at how the access token can be refreshed without having the user go through the web-based authentication flow each time.

Lab 3, Exercise 3: Working with Refresh Tokens

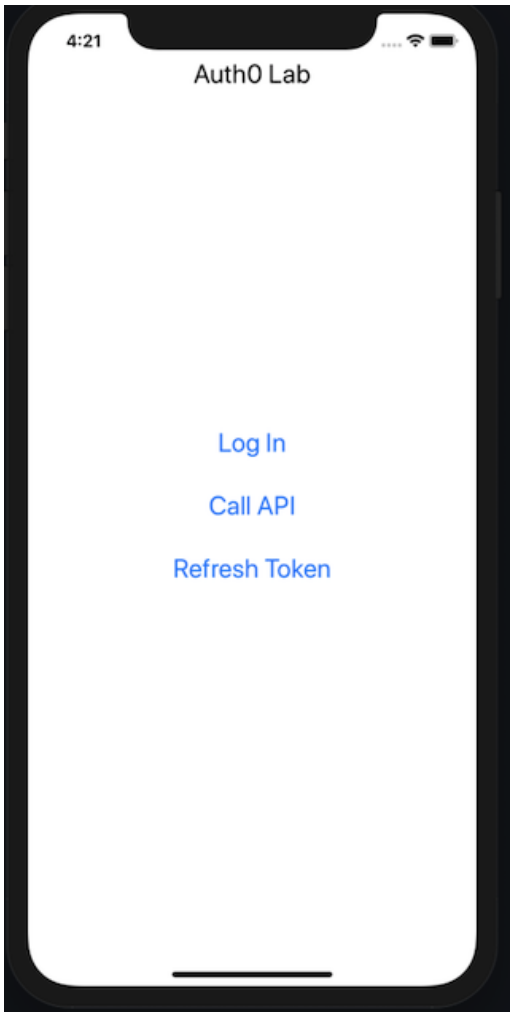
In this exercise, you will explore the use of refresh tokens. A refresh token is a special kind of token that can be used to obtain a renewed access token. You are able to request new access tokens until the refresh token is blacklisted. It's important that refresh tokens are stored securely by the application because they essentially allow a user to remain authenticated forever.

For native applications such as our iOS application, refresh tokens improve the authentication experience significantly. The user has to authenticate only once, through the web authentication process. Subsequent re-authentication can take place without user interaction, using the refresh token.

1. Go to File > Open in Xcode and select *lab-03/exercise-03/begin/exercise-03.xcworkspace* (make sure you pick the right file extension), then open *exercise-03/ViewController.swift*. This code picks up where the previous exercise left off and adds a new button to refresh the access token.

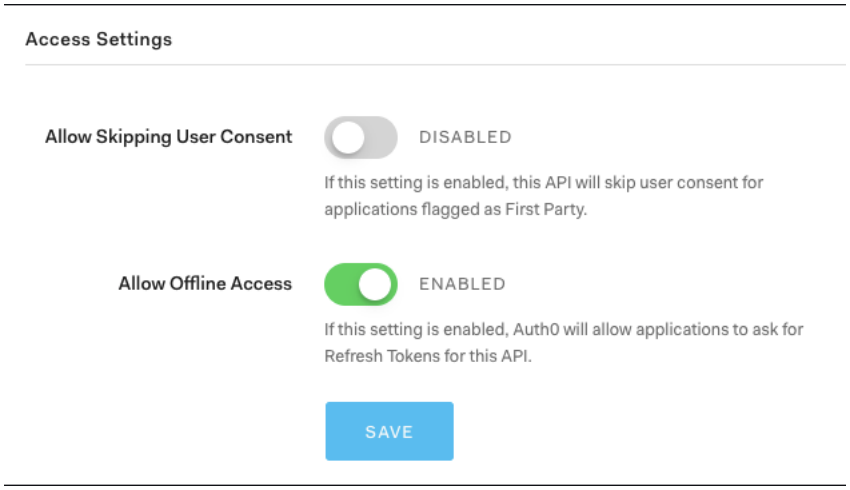
2. Open the *exercise-03/Auth0.plist* file and replace the placeholder values for ClientId and Domain with the ones from the Auth0 Application created before.

3. Click the Play button (or Product > Run from the Xcode menu) to run the app. Touch the Refresh Token button and look for a “Refresh Token” message to the Debug area in Xcode.



You are now going to add the *offline_access* scope, which gives the iOS application access to resources on behalf of the user for an extended period of time. Before you can use this scope, you need to make sure that Auth0 will allow applications to ask for refresh tokens for your API.

4. Navigate to the APIs screen in your Auth0 Dashboard. Open the API that you created to represent your expenses API and ensure the Allow Offline Access option is on.



5. Next, we're going to add the *offline_access* scope to the authentication request. Open *exercise-03/ViewController.swift* and, in the *actionLogin* method, add *offline_access* to the *.scope()* method.

```
// exercise-03/ViewController.swift
// ...
  @IBAction func actionLogin(_ sender: Any) {
      Auth0
          .webAuth()

          // Replace this line ❌
          // .scope("openid profile read:reports")

          // ... with the line below 👇
          .scope("openid profile read:reports offline_access")

          // ...
      }
  }
// ...
```

6. Click the Play button (or Product > Run from the Xcode menu) to run the app. Log in again and check the Debug area in Xcode for the response.

```
{
  "access_token":"3tjDJ3hsFOSyCr02spWHUhHNajxLRonv",

  // Here is the refresh token we asked for 👇
  "refresh_token":"sAvc4BJyOGs2I6Yc4e6r9NmReLp0kc-I6peiauDEt-usE",

  "id_token": "eyJ0eX[..].eyJodH[..].thhf0M[..]",
  "expires_in": 86400,
  "token_type": "Bearer"
}
```

7. We're going to send the refresh token to the authorization server using a *refresh_token* grant to get a new access token. In *ViewController.swift* and create a private variable in the ViewController class to create a way for actionRefresh method to access the refresh token.

```
// exercise-03/ViewController.swift
// ...
class ViewController: UIViewController {

   private var accessToken: String?

   // Add the line below 👇
   private var refreshToken: String?

   // ...
}
// ...
```

8. Assign the refresh token obtained during authentication to this private variable in the .success code block.

```
// exercise-03/ViewController.swift
// ...
    @IBAction func actionLogin(_ sender: Any) {
        // ...
                    case .success(let result):
                        // ...
                        self.accessToken = result.accessToken

                        // Add the line below 👇
                        self.refreshToken = result.refreshToken

                    case .failure(let error):
                        // ...
    }
// ...
```

9. In the *actionRefresh* method, check that the user has authenticated and that a refresh token is available before making any calls to the authentication API. In the code below, the class-scoped property *refreshToken* is assigned to a local *refreshToken* variable. If the class-scoped property is empty, an error will be returned.

```
// exercise-03/ViewController.swift
// ...
    @IBAction func actionRefresh(_ sender: Any) {
        print("Refresh Token")

        // Add the code below 👇
        guard let refreshToken = self.refreshToken else {
           print("No Refresh Token found")
           return
        }
    }
// ...
```

10. The Auth0.swift SDK makes available a .renew() method, which takes a refresh token as a parameter and performs a call to the authorization server's token endpoint using the refresh_token grant. Add the following code to the actionRefresh method after the code from the previous step.

```swift
// exercise-03/ViewController.swift
// ...
    @IBAction func actionRefresh(_ sender: Any) {
        // ... code from the previous steps

        // Add the code below 👇
        Auth0
            .authentication()
            .logging(enabled: true)
            .renew(withRefreshToken: refreshToken)
            .start { response in
                switch(response) {
                    case .success(let result):
                          print("Refresh Success")
                          print("New Access Token: \(result.accessToken ?? "No Access Token Found")")
                          self.accessToken = result.accessToken
                    case .failure(let error):
                          print("Refresh Failed: \(error)")
                }
            }
    }
// ...
```

11. Click the Play button (or Product > Run from the Xcode menu) to re-run the app. Tap Log In and, after successful authentication, touch the Refresh Token button. Look in the Xcode the debug area for the request. You should see a POST to the token endpoint, showing the refresh token grant in action.

```
POST https://${account.namespace}/oauth/token HTTP/1.1
Auth0-Client: eyJuYW1lIjoiQXV0aDAuc3dpZnQiLCJ2ZXJzaW9uIjoiMS4xMy4wIiwic3dpZnQtdmVyc2lvbiI6IjMuMCJ9
Content-Type: application/json

{"grant_type":"refresh_token","client_id":"${account.clientId}","refresh_token":"2CNxaPe0UIkX_PZkLEkKuoAuRsP6Ycg81XR1j
```

12. Look for the response after the request above. You should see a response including a new access_token, new id_token, and a new expires_in time (some of the trace was omitted for brevity).

```
HTTP/1.1 200
Content-Type: application/json
Date: Tue, 27 Aug 2019 16:25:26 GMT
x-ratelimit-remaining: 29
x-ratelimit-reset: 1566923126
x-ratelimit-limit: 30
Content-Length: 1923

{"access_token":"eyJ0eX[..].eyJpc3[..].Smqrd7[..]",
"id_token":"eyJ0eX[..].eyJua[..].Ff5Q5[..]",
"scope":"openid profile read:reports offline_access",
"expires_in":3600,"token_type":"Bearer"}
```

Notice that you don't receive a new refresh_token in the response from the authorization server. The refresh_token from the initial authentication must be retained. Also, note that in the code added to the actionRefresh method the access_token received is stored in the self.accessToken class property. This is so the new access token can be used in other methods. If you try calling the API again, the request will be made with your new access_token.

Now that you are able to obtain a fresh access token by using the refresh token, it's time to see what happens when a token expires.

13. Navigate to the APIs screen in your Auth0 Dashboard and open the expenses API. Set both the Token Expiration and the Token Expiration For Browser Flows fields to 10 seconds and save the changes.

14. In your app simulator, tap Log In to walk through the authentication process again and get a new access token with the shorter expiration. Immediately tap the Call API button to see the API call succeed.

15. Wait 10 seconds for the token to expire and click the Call API button again. You should see the API call fail with a *Status Code: 401* in the debug area.

```
<NSHTTPURLResponse: 0x600001f34460> { URL: http://localhost:3001/ } { Status Code: 401, Headers {
    Date =     (
        "Tue, 27 Aug 2019 16:36:10 GMT"
    );
    "www-authentication" =     (
        "Bearer realm=\"api\", error=\"invalid_token\", error_description=\"invalid token\""
    );
} }
```

16. Tap Refresh Token and check the debug area to see the refresh token grant happen. Then, tap Call API, and you should get a *Status Code: 200* along with the expenses data again.

🎉 You have completed Lab 3 by building a native mobile application calling a secure API with refresh capability! 🎉

Lab 4: Single Page App

Lab 4, Exercise 1: Adding Sign On

In this lab, you will learn how to add sign-on capabilities to a Single-Page Application (SPA) and how to make this app consume an API that is secured with Auth0. You will integrate the SPA with Auth0 so that your users are able to use the Auth0 Universal Login Page to authenticate.

The SPA in question is a vanilla JavaScript application that consumes an API similar to the one you have used in previous labs (this API also exposes a secured endpoint that returns a list of expenses). The difference is that the API in this lab does two additional things:

- The API supports CORS to enable the SPA to consume it from a different domain (or a different port in a local environment).
- The API exposes a public endpoint that returns a summary of its database. The SPA consumes this endpoint on its homepage to share the summary publicly.

In this exercise, you will focus on integrating the SPA with Auth0 and getting the profile of the logged-in user. Exercise 2 will show how to consume the private endpoint exposed by the API.

1. First, you will run a version of the app that is not integrated with Auth0. Open a new terminal and browse to */lab-04/exercise-01/begin/api* in your locally-cloned copy of the identity exercise repo. This is where the code for your API resides. Install the dependencies using npm.

```
# Make sure we're in the right directory
❯ pwd
/Users/username/identity-102-exercises/lab-04/exercise-01/begin/api

❯ npm install
# Ignore any warnings

added XX packages in X.XXs
```

2. Make a copy of the .env-sample file and name it .env.

```
❯ cp .env-sample .env
```

3. Edit the new .env file to add your tenant domain and save the file.

```
PORT=3001
ISSUER_BASE_URL=https://${account.namespace}
ALLOWED_AUDIENCES=https://expenses-api
```

4. Run the API:

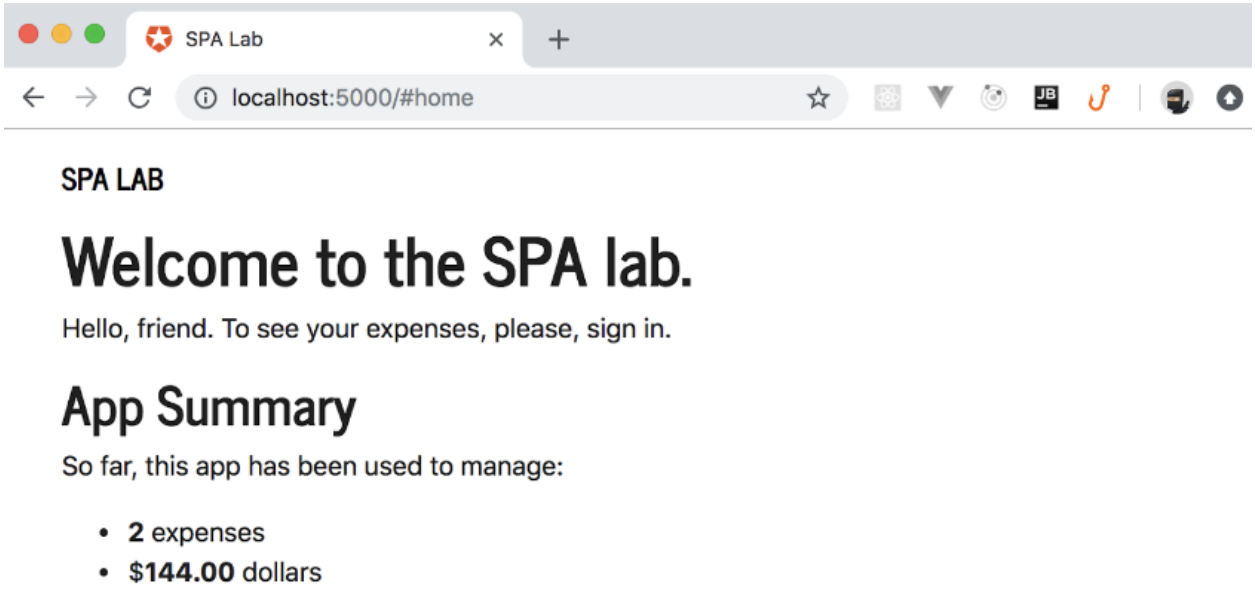
```
❯ npm start
```

5. Open a new terminal in the /lab-04/exercise-01/begin/spa folder and run http-server to host the SPA.

```
# Navigating from the previous directory
❯ cd ../spa

# Make sure we're in the right directory
❯ pwd
/Users/username/identity-102-exercises/lab-04/exercise-01/begin/spa

❯ npx http-server . -p 5000 -c-1
npx: installed 26 in 3.459s
Starting up http-server, serving .
Available on:
  http://127.0.0.1:5000
  http://10.10.10.10:5000
  http://192.168.1.4:5000
Hit CTRL-C to stop the server
```

Note: On the command above, -p 5000 makes the server listen on port 5000, and -c-1 makes browsers ignore their own cache. This last parameter is important to facilitate the development process.

6. Open localhost:5000 in a web browser, and you should see the page below. If you do not see the App Summary section, make sure your API is properly running at port 3001.



This is the homepage of your SPA. Right now, this SPA has no integration with Auth0. Also, the app is only consuming the public endpoint provided by the API. This endpoint

returns two pieces of information: the total number of expenses recorded in the database (two, in this case), and the sum of their amount ($144.00). The SPA is using this information to feed the "App Summary" section of the page you are seeing.

7. To register this SPA with Auth0, log into the Auth0 Dashboard, go to the Applications page, and click the Create Application button.

8. Set a descriptive name (e.g., "Identity Lab 4 - Single Page App"), choose Single Page Web Application for the type, and click Create.

9. You should now see the Quickstart section that describes how to integrate Auth0 with a production application. Click the Settings tab at the top to see the Application settings.

10. Add *http://localhost:5000/#callback* to the Allowed Callback URLs field. Auth0 will allow redirects only to the URLs in this field after authentication. If the one provided in the authorization URL does not match any in this field, an error page will be displayed.
11. Add http://localhost:5000 to the Allowed Web Origins field. This field defines what URLs will be able to issue HTTP requests to Auth0 during a silent authentication process. Your SPA will leverage this mechanism to check whether the current browser has an active session on the Auth0 authorization server.

12. Add **http://localhost:5000** to the Allowed Logout URLs field. Auth0 will allow redirects only to the URLs in this field after logging out of the authorization server.

13. Scroll down and click Show Advanced Settings, then OAuth. Make sure JsonWebToken Signature Algorithm is set to RS256.

14. Scroll down and click Save Changes

Now that you have registered your SPA with Auth0, you can update your code to integrate both. Below is a summary of the steps you will execute:

- Import auth0-spa-js and configure it with your own Auth0 settings.
- Add code to handle the authentication callback.
- Add code to restrict content to authenticated users only.
- Implement login and logout.
- Obtain and display user profile information.

The auth0-spa-js SDK is a simple, lightweight, and opinionated client developed by Auth0 that executes the OAuth 2.0 Authorization Code Grant Flow with PKCE. This client allows developers to quickly and securely implement authentication in their browser-based applications.

15. Open the *spa/index.html* file and search for the <*script*> tag that imports app.js. Right before that tag, add the auth0-spa-js library to your SPA using the CDN-hosted link.

    https://cdn.auth0.com/js/auth0-spa-js/1.1.1/auth0-spa-js.production.js

```
<!-- spa/index.html -->
<!-- ... -->

    <!-- Add the tag below 👇 -->
    <script src="https://cdn.auth0.com/js/auth0-spa-js/1.1.1/auth0-spa-js.production.js"></script>

    <script src="app.js"></script>
  </body>
</html>
```

16. Open *spa/app.js*. This file contains the code that starts your SPA in the browser. At the top of it, you will see several constants that reference DOM elements. Right after those definitions, add the variable declaration to allow auth0Client to be used globally.

```
// spa/app.js
// ... other constants
const loadingIndicator = document.getElementById('loading-indicator');

// Add the line below 👇
let auth0Client;
```

17. In the *window.onload* function, add the code that configures the auth0-spa-js library with your own Auth0 details. Replace both placeholders with the Domain and Client ID properties for your SPA Application in Auth0.

```
// spa/app.js
// ...

window.onload = async function() {
  let requestedView = window.location.hash;

  // Add the code below 👇
  auth0Client = await createAuth0Client({
    domain: '${account.namespace}',
    client_id: '${account.clientId}'
  });

  // ...
};
```

18. Implement the authentication callback after the snippet above in the window.onload function.

```
// spa/app.js
// ...

window.onload = async function() {
  // ... code from the previous step

  // Add the code below 👇
  if (requestedView === '#callback') {
    await auth0Client.handleRedirectCallback();
    window.history.replaceState({}, document.title, '/');
  }

  // ...
};
```

The *requestedView* variable is used to identify if the request in question refers to a user coming back from the authentication process (i.e., if this is a user being redirected back to the application by Auth0 after authenticating).

19. Restrict site content to authenticated users only by finding the allowAccess function definition (at the bottom of the file) and replacing it with an authentication check.

```
// spa/app.js
// ...

/* Replace the code below ❌
async function allowAccess() {
  await loadView('#home', content);
  return false;
}
*/

// ... with this 👇
async function allowAccess() {
  if (await auth0Client.isAuthenticated()) {
    return true;
  }
  await loadView('#home', content);
  return false;
}
```

The goal of this function is to allow or deny access to whatever route calls it. This function is called from the #expenses route to ensure the user is logged in. Although you won't use that route on this exercise (only on the next one), you can check the code in the spa/scripts/expenses.js file. The previous version of this function was hardcoded always to deny access and redirect to the #home route because no authentication mechanism was in place yet.

You are now making use of the isAuthenticated() method provided by the SPA SDK to block unauthenticated users from accessing the route calling this function. If an anonymous user tries to access it, the app will detect that they are not authenticated and will redirect them back to the homepage.

20. To give users a way to log in and view restricted content, open the spa/scripts/navbar.js file. There, you will see the definition of a few constants within the async function() IIFE. Below that, we'll add code to handle a click event on the login button.

```
// spa/scripts/navbar.js
(async function() {
  // ... other constants
  const logOutButton = document.getElementById('log-out');

  // Add the code below 👇
  logInButton.onclick = async () => {
    await auth0Client.loginWithRedirect({
      redirect_uri: 'http://localhost:5000/#callback'
    });
  };

})();
```

The logInButton button will now, when clicked, invoke the *loginWithRedirect()* method provided by the SPA SDK to start the authentication process. When users click the login button, they will be redirected to the authorization server.

The *redirect_uri* property passed to the *loginWithRedirect()* method defines the URL that Auth0 must call after the authentication phase is concluded. This is the same URL listed in the Allowed Callback URLs field in the Auth0 Dashboard. If you use another URL without whitelisting it first, Auth0 will show an error page.

21. To define what happens when users click the logout button, add the code below right after the code from the previous step in the same function.

```
// spa/scripts/navbar.js
(async function() {
  // ... code from the previous example

  // Add the code below 👇
  logOutButton.onclick = () => {
    auth0Client.logout({
      returnTo: 'http://localhost:5000'
    });
  };

})();
```

In this case, you are making the *logOutButton* button invoke the *logout()* method provided by the SPA SDK to end the user's Auth0 session. The returnTo property passed to this method works similar to the *redirect_uri* passed to the *loginWithRedirect()* method. This logout return URL was whitelisted in your Auth0 Application using the Allowed Logout URLs field.

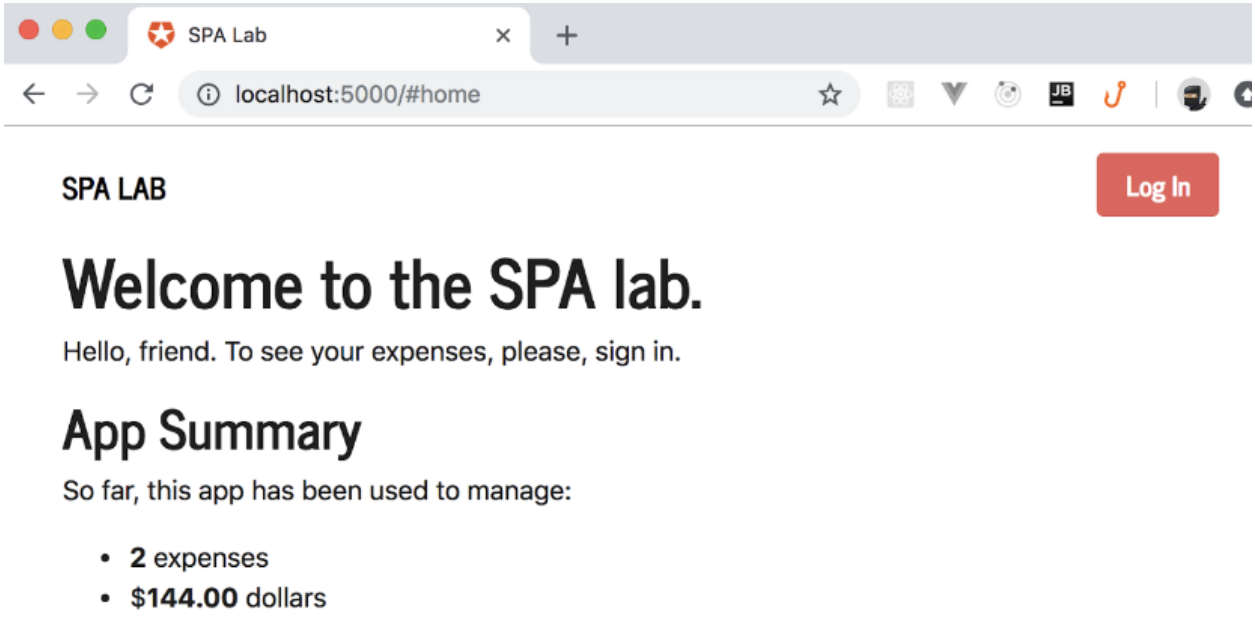
22. Now, you will implement the behavior in your application that depends on whether the user is authenticated or not. After the code from the previous step, add the following code:

```
// spa/scripts/navbar.js
(async function() {
  // ... code from the previous example

  // Add the code below 👇
  const isAuthenticated = await auth0Client.isAuthenticated();
  if (isAuthenticated) {
    const user = await auth0Client.getUser();
    profilePicture.src = user.picture;
    userFullname.innerText = user.name;

    logOutButton.style.display = 'inline-block';
  } else {
    logInButton.style.display = 'inline-block';
  }

})();
```

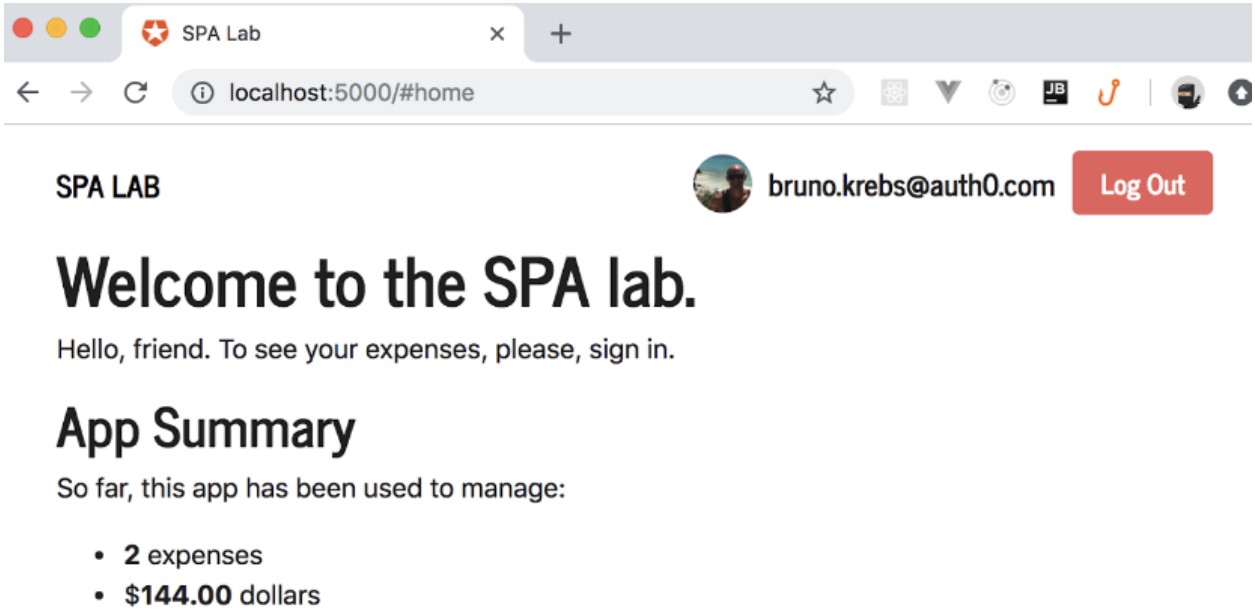
You defined a flag called *isAuthenticated* that defines if the user is authenticated or not. If they are authenticated, you use the *getUser()* method provided by the SPA SDK to extract their profile details. These details populate the screen with the name of the user and their picture.

The *isAuthenticated* flag also defines what button the app will show based on their authentication status: the logInButton or the *logOutButton*. If the user has a session with this application, the app will show the *logOutButton* button. Otherwise, it will show the *logInButton*.

23. Save all your changes and refresh the browser. You will see a screen that is slightly different from the previous one.



24. In this new screen, click the Log In button to start the authentication process. Log in with your database user and accept the consent request. After successful authentication, Auth0 will redirect you back to your app, and your profile details (username and picture) will be shown near the upper-right corner:



Note: If you log in using a social identity provider (Google, Facebook, etc.), you will need to log in every time you refresh the SPA. This happens because you are using Auth0's test development keys for the identity provider. To prevent this from happening, you would need to register your application with the relevant Identity Provider and replace the test development keys on the Auth0 dashboard with your own. However, for the purposes of this lab, you should log in with a username and password to avoid the aforementioned behavior.

For more information, see Test Social Connections with Auth0 Developer Keys. :::

If you want to test the *allowAccess()* function, which restricts access for particular routes depending on whether the user is authenticated, try navigating to localhost:5000/#expenses. If you are logged in, the page will load successfully showing a "Loading..." text (the content for this page will be implemented in the next exercise). If you are not logged in, the app will redirect you to the homepage.

25. Let's explore the relevant network traces of the authentication process used in this lab. First, click the Log Out button, then, once you return to the app with your session ended, open Chrome's Developer Tools and go to the Network tab.

26. Click the Log In button. The first thing you will see in Developer Tools is a request similar to this:

```
https://${account.namespace}/authorize?client_id=...
```

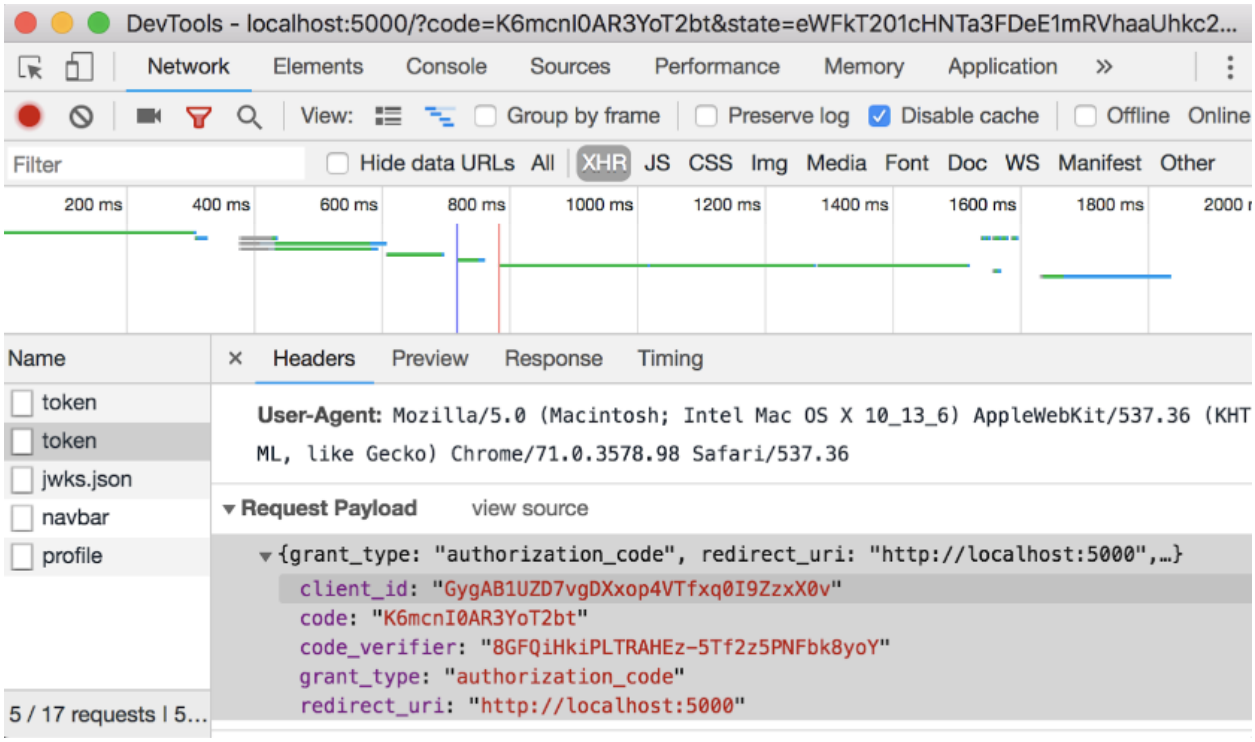
This request is created and triggered by the SPA SDK when a user clicks on the login button. If you check the query parameters passed alongside this URL, you will find, among others, the following:

- **client_id** - the unique identifier of your application for the authorization server.
- **response_type** - the artifacts needed to authenticate the user in your application. In this case, the SPA SDK is requesting an authorization code. This indicates to Auth0 that you will be using the Authorization Code Grant Flow (as defined by the OAuth 2.0 specification).
- **scope** - a space-delimited list of permissions that the application requires. In this case, your app is requesting *openid profile email*. These are scopes defined by the OpenID Connect specification and give your app access to specific data in the user profile.
- **code_challenge** - this is a code that the authorization server will store and associate with the authorization request. In a future step, before issuing tokens to your application, the authorization server will use this code to verify (against another code called **code_verifier** that is handled internally by the SPA SDK) if it is secure to issue tokens. By using the code challenge and verifier, the SPA SDK is making the authorization process use a variant of the Authorization Code Grant Flow called PKCE.

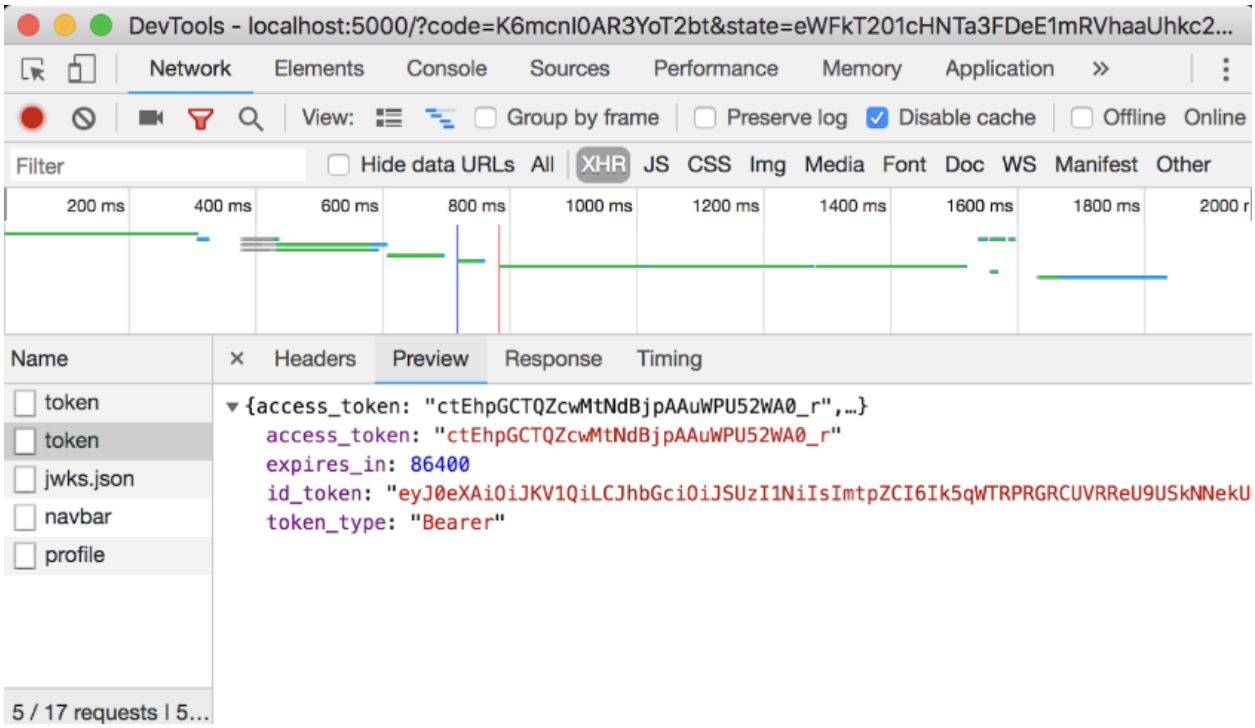
27. Log in again by clicking Log In. After Auth0 redirects back to your application, check the Network tab in Developer Tools, and you will see a list of requests, starting with the callback. Filter the requests and show only the XHR ones (those generated by an XMLHttpRequest JavaScript object).

28. Click on the "token" request (if there are two, click the second one). You will see that it is a POST request to the token endpoint of the authorization server. This request exchanges the code retrieved on the authentication process for the tokens needed in your application.

29. To see the data sent to the token endpoint, scroll to the bottom of the Headers tab. There, you will see that the request payload includes the following fields: *client_id*, *code*, *code_verifier*, *grant_type*, and *redirect_uri*.



30. Switch to the Preview tab to see the tokens returned by the authorization server.

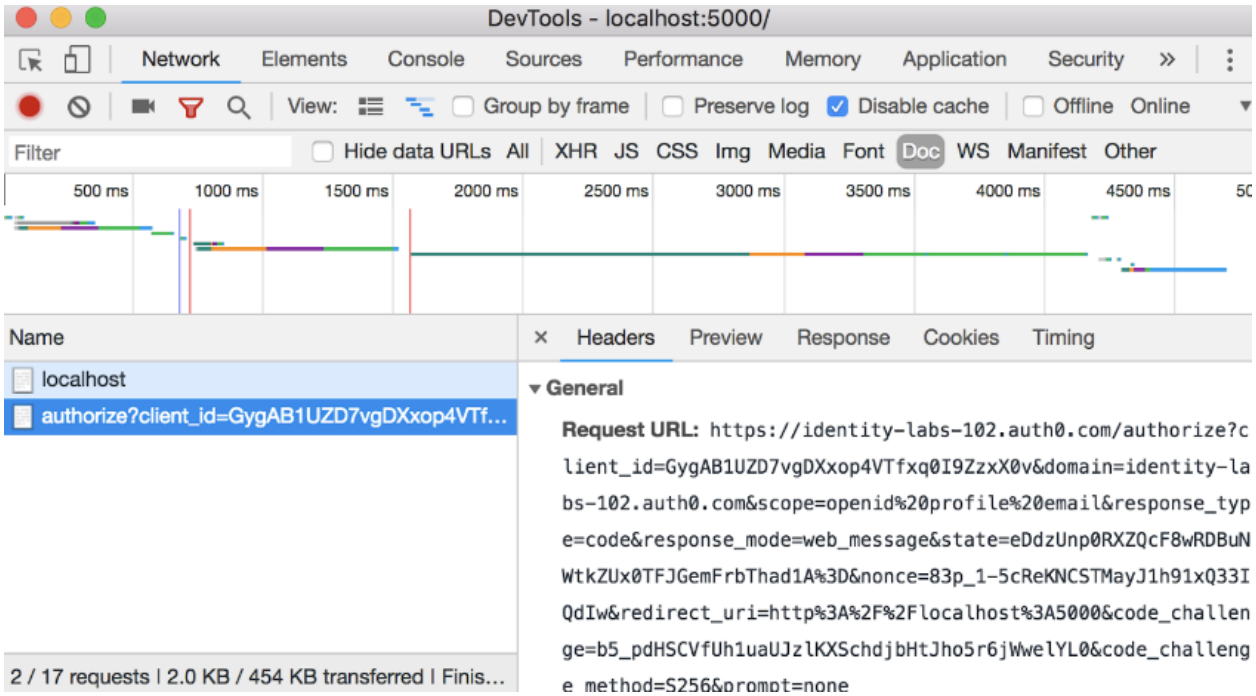


Note: If you are using a content blocker or browser setting that blocks third-party cookies, you will notice that in the step below, when authenticated, you need to log in again after refreshing the page. In that case, try changing your content blocker settings

to allow your Auth0 domain (or turning it off altogether for localhost). Blocking all third-party cookies is not generally recommended as it is known to cause issues in some Web sites. This problem does not occur when Custom Domains are used.

31. If you refresh the SPA and change Developer Tools to filter requests by Doc (Documents), you will see a request called *authorize*. This request is similar to the one issued after the authentication process with a few differences:

    - The request uses a different *response_mode*, in this case *web_message*. This is part of a strategy used to renew tokens silently.
    - The request defines a new query parameter called *prompt* set to *none*. As defined in the OpenID Connect protocol, this parameter is used on authentication requests that must not display user interaction. This parameter is also part of the silent authentication process.



For this to work properly, the silent authentication process requires the referrer URL to be whitelisted. This is why you added http://localhost:5000 to the Allowed Web Origins field for your Auth0 Application. Otherwise, the silent authentication process would fail, and your users would need to log in again interactively.

Lab 4, Exercise 2: Calling a Protected API

In this exercise, you will learn how to make your SPA consume, on behalf of the user, the private endpoint exposed by the API.

Note: You can continue using the source code from the previous exercise, or if you are starting from scratch, use the code in the *exercise-02/begin* folder. Make sure you run steps 1-6 in exercise 1 either way.

To make your SPA consume the private endpoint on the user's behalf, it must fetch an access token first, then call the protected API. The first time your application asks for an access token for an API, the authorization server will request explicit consent from your users.

To request this consent, your application will open a popup that will load a page from the authorization server. On that page, your users will learn what type of access your application is requesting, and they will be able to grant access or deny it.

1. Open *spa/scripts/expenses.js*. You will see a few constants that reference DOM elements (like *loadExpensesButton* and loadingExpenses). Right after these constants, add the configuration code for the API.

```
// spa/scripts/expenses.js
(async function() {
  // ... other constants
  const loadingExpenses = document.getElementById('loading-expenses');

  // Add the code below 👇
  const expensesAPIOptions = {
    audience: 'https://expenses-api',
    scope: 'read:reports',
  };

  // ...
})();
```

The *expensesAPIOptions* constant is a configuration object that will tell the SPA SDK the audience and scope needed in the access token your application will request. The SDK will try to fetch access tokens capable of consuming the *https://expenses-api* API with the *read:reports* scope.

2. Request the access token required to retrieve the expenses from the protected API on behalf of the user with the code below.

```
// spa/scripts/expenses.js
(async function() {
  // ... code from the previous example

  // Add the code below 👇
  try {
    const accessToken = await auth0Client.getTokenSilently(expensesAPIOptions);
    await loadExpenses(accessToken);
  } catch (err) {
    if (err.error !== 'consent_required') {
      alert('Error while fetching access token. Check browser logs.');
      return console.log(err);
    }

    loadExpensesButton.onclick = async () => {
      accesstoken = await auth0Client.getTokenWithPopup(expensesAPIOptions);
      await loadExpenses(accesstoken);
    };

    consentNeeded.style.display = 'block';
    loadExpensesButton.style.display = 'inline-block';
    loadingExpenses.style.display = 'none';
  }

  // ...
})();
```

The lines above are nested inside a try/catch block and are executed when the expenses view is requested. First, the application calls the *getTokenSilently()* method (provided by the SPA SDK) to see if your application is able to fetch a token without involving your user. If your app fetches the access token successfully, the application calls the loadExpenses() function with this token to load and display expenses (you will define this function in the next step).

If your application is not able to fetch an access token (an error occurs, or the user is not logged in), the application checks if the problem is *consent_required*, which means that the user has not given consent to access the API yet. If that is not the case, it means an unknown error has been raised, and your application will alert the user and log the error to the browser's console.

If *consent_required* is indeed the problem, then you define the behavior of the loadExpensesButton and show the *consentNeeded* and loadExpensesButton DOM elements. These DOM elements are responsible for letting the user know that they will need to give your application explicit consent to consume the API on their behalf. More

specifically, after reading the message, if your users click the *loadExpensesButton*, your application will trigger the SDK-provided *getTokenWithPopup()* method to open a popup where your users will be able to give consent.

3. Call the API using the access token with the code below.

```
// spa/scripts/expenses.js
(async function() {
  // ... code from the previous example

  // Add the code below 👇
  async function loadExpenses(accesstoken) {
    try {
      const response = await fetch('http://localhost:3001/', {
        method: 'GET',
        headers: { authorization: 'Bearer ' + accesstoken }
      });

      if (!response.ok) {
        throw 'Request status: ' + response.status;
      }

      const expenses = await response.json();
      displayExpenses(expenses);
    } catch (err) {
      console.log(err);
      alert('Error while fetching expenses. Check browser logs.');
    }
  }

  // ...
})();
```

After fetching an access token (silently or explicitly through the popup), your application will invoke the function above to issue a request to the private API on the user's behalf. Note the authorization header passed to the *fetch()* function; this header includes the access token required to consume the expenses API.

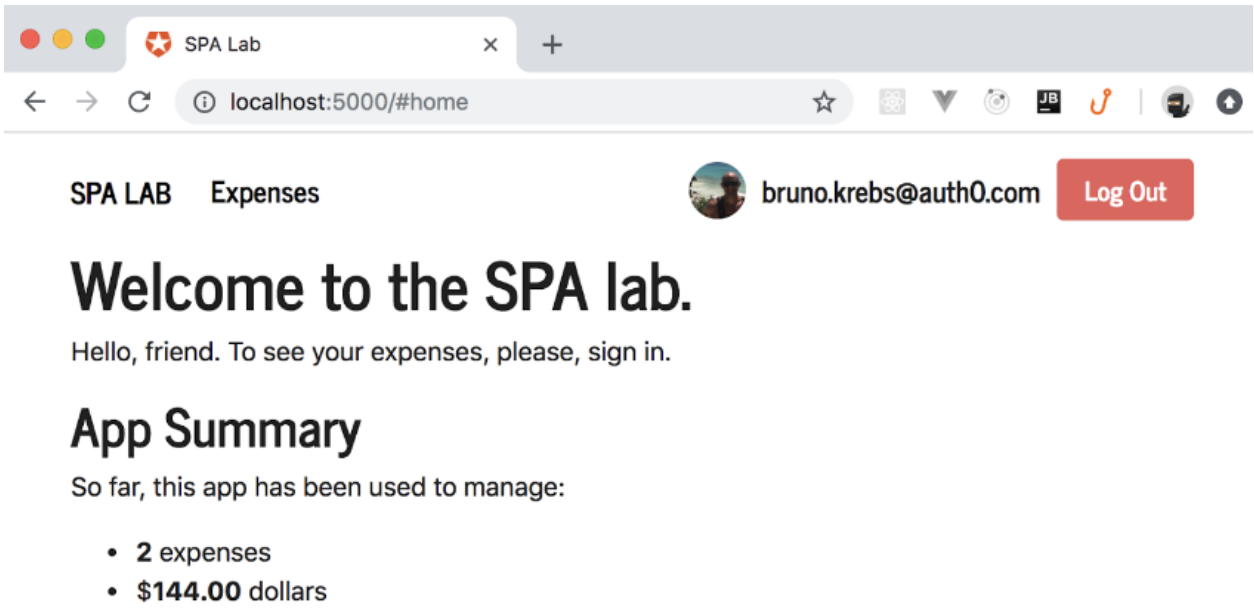
After executing the request to the API, if successful, the *displayExpenses()* function is called. This function creates the DOM elements on the page to represent the expenses and is already defined in the *spa/scripts/expenses.js* file at the bottom.

4. Open *spa/scripts/navbar.js* and search for the block that gets executed when the user isAuthenticated. Inside this block, right after changing the display property of the

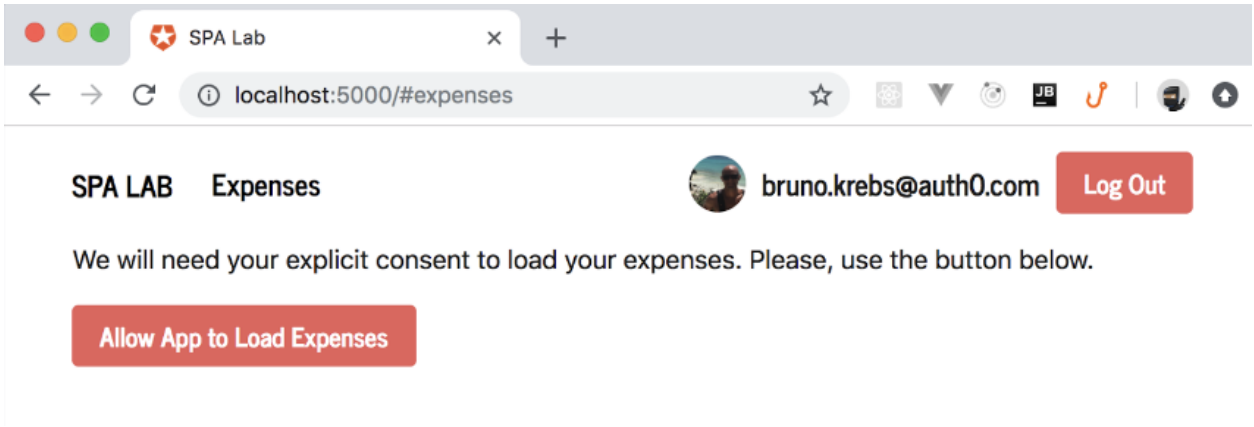
logOutButton, add the code below. This will display a link to the expenses view in the navigation bar when the user is authenticated.

```
// spa/scripts/navbar.js
(async function() {
  // ... code from the previous example

  const isAuthenticated = await auth0Client.isAuthenticated();
  if (isAuthenticated) {
    // ...

    // Add the line below 👇
    expensesLink.style.display = 'inline-block';
  } // ...

})();
```

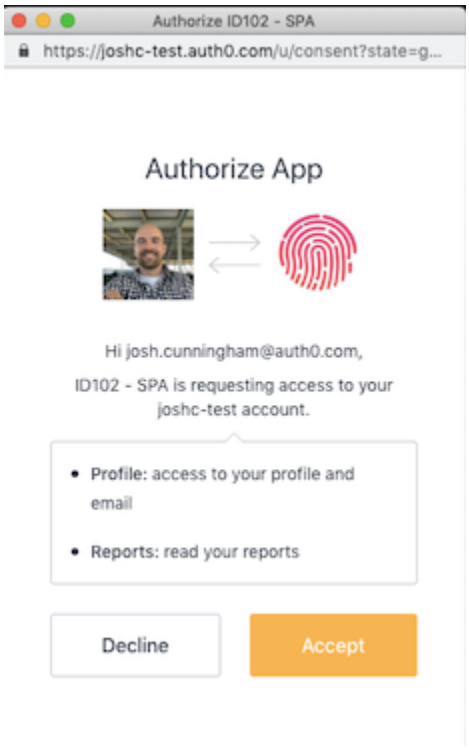
5. You are now ready to test the new version of the application. Save all the changes and reload the application in your browser. You should see the Expenses link in the navigation bar at the top.



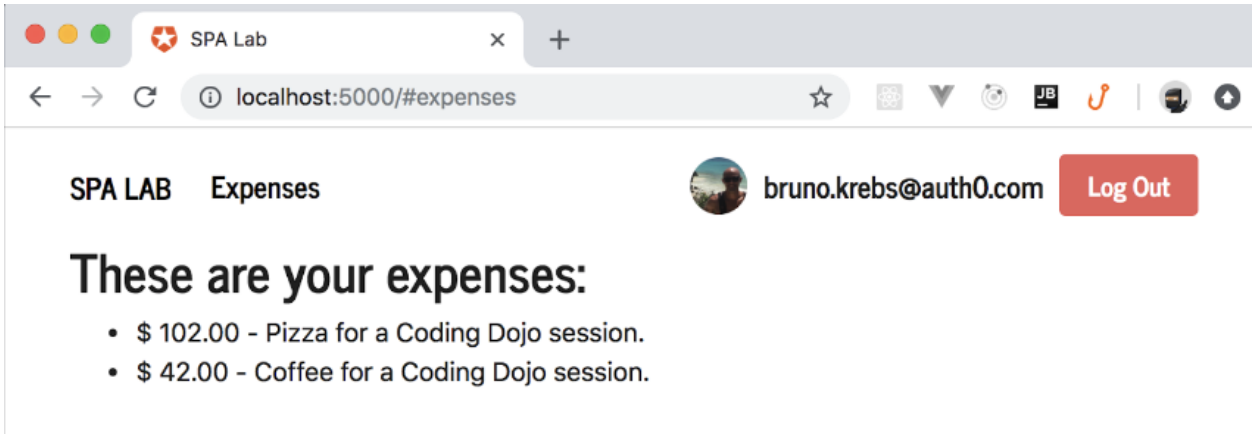
6. Click Expenses link, and you should see the consent prompt.



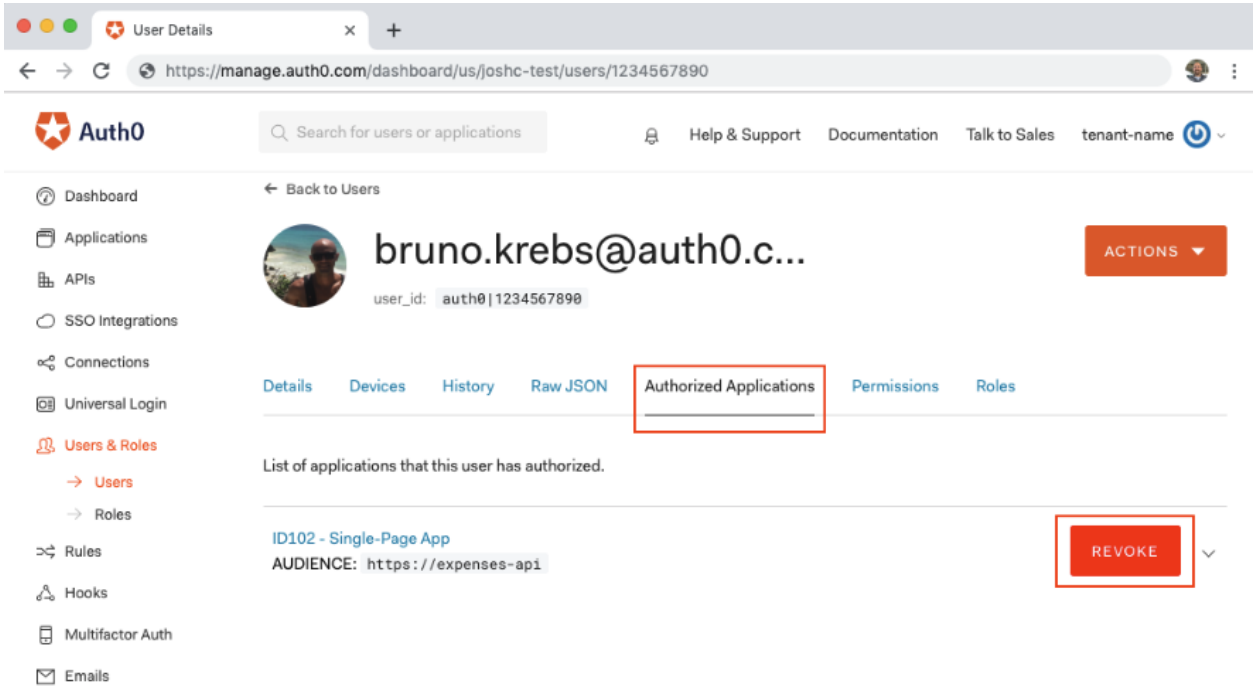
7. Because you have not provided consent yet, you will see an Allow App to Load Expenses button. Clicking it will open the consent popup; note that the Reports scope is now included.



In the popup, click the Accept button to give consent. The popup will close, and your application will get the access token it needs. With this token, the SPA will call the *loadExpenses()* function and show the data retrieved from the API.



Note: If you want to recreate the scenario where consent is needed, go to the Users screen of the Auth0 Dashboard, view your test user, click the Authorized Applications tab, and click Revoke for the single-page application with the Expenses API audience.



🎉 You have completed Lab 4 by building a single-page application calling a secure API! 🎉