Securing React Apps with Auth0

by Cory House

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hi everyone. I'm Cory House, and welcome to my course, Securing React Apps with Auth0. I'm the principal consultant at reactjsconsulting.com. Getting security right is hard. There's a dizzying amount of jargon and tradeoffs to consider. So in this course, we'll forge a clear path to successfully securing a React app by using Auth0, the world's most popular auth platform. Some of the major topics that we'll cover include modern security protocols like OAuth 2, OpenID Connect, and JWT tokens, login and signup, authorization approaches, security concerns for single-page apps, and techniques for securing API calls. By the end of this course, you'll be prepared to secure your own React app with rich rules for authentication and authorization. Before beginning the course, you should be generally familiar with modern JavaScript and React. I hope you'll join me on this journey to learn how to secure React apps with the Securing React Apps with Auth0 course, at Pluralsight.

Authorization and Authentication Standards

Intro

Hi there, and welcome to Securing React Apps with Auth0. In this course, we'll explore how to implement authentication and authorization for your React apps using Auth0. In this module, we'll review popular auth providers and consider why they're worth using. Then, we'll explore the modern security standards that we'll use to handle authentication and authorization including OAuth 2.0, OpenID Connect, and JSON Web Tokens. Alright, let's get started.

Why Use an Auth Provider?

When considering security for your React app, you have two core concerns, authentication and authorization. Authentication answers the question who are you? Authorization answers the question what are you allowed to do? So for example, in our React app, authentication is when someone logs in with their credentials, typically with an email or password. And authorization is when we determine if the user has the rights to do something in our app, such as delete a record or load a certain page. Security is complicated and it's critical that you get it right. So today, developers are increasingly trusting auth providers to help secure their apps. Auth0 and Okta are the two most popular providers. So, why should you consider using an auth provider? First, security is complex and it's easy to get wrong. It can feel like a maze getting all of the options right. By choosing a provider that specializes in security, you can trust that they know what they're doing and spend your time focusing on building your React app. Second, as you'll soon see, you can customize auth providers to your needs so that they feel integrated with your application. So what about price? Well, pricing varies by provider, but some providers offer a generous free tier that works for many apps. In this course, we use Auth0 because it's the most popular, it has superb documentation, it offers excellent mature libraries for creating single-page apps, and the price is right because it offers a generous free tier. Auth0's free tier supports up to 7, 000 active users. All the features that we use throughout this course are included in this free tier. If you're curious what features are available in the paid tiers, check out the pricing page, but for many apps, the free tier is all that you'll need. But before we jump in and set up Auth0 with React, let's learn some key security technologies.

OAuth 2.0

Let's explore OAuth. Throughout this course, we use OAuth 2.0, which is the latest version, but I may just say OAuth for the sake of brevity. Before we begin, let me set some groundwork to avoid confusion. Auth0 is the security provider that we'll be using throughout this course. OAuth

is a security protocol that we're going to use throughout this course as well. Yes, their names are similar, and I suspect that that's no accident. Anyway, in this clip, we're going to discuss the OAuth security protocol, so let's dive in. So remember the two core security concerns that I mentioned a moment ago. Authentication is about who you are and authorization is about what you're allowed to do. OAuth 2.0 is about what you're allowed to do. OAuth gives us access to third-party data. Imagine that you've built a React app that needs to access each of your user's basic Google account data. With OAuth, you can ask the user for permission to access their data. This is an example consent screen. Here you can set to sharing specific information. In OAuth terms, these permissions are called scopes. Scope is another word for permissions. Scope is a string that denotes a type of access like read-only access to Google Drive, write access, etc. The list of scopes your application requires drives this consent screen. As a side note, if you go create a personal access token on GitHub, you can see a good example of scopes in use. When you create a token, you select which scopes the token should be able to access. A scope is just a string, so you can decide how to format scope names, but typically a scope has a colon in the middle, and a verb on one side, and a noun on the other. For example, read:user or edit:product. So OAuth is a standardized authorization protocol. What makes OAuth special is you can authorize a user on your web app without exposing their password. And as we just saw, OAuth allows applications to get info from third parties. Your users just have to click consent to grant your application access. To set up OAuth, you need to register your React app with the service provider such as Google, Facebook, or any other provider that you'd like to integrate with. You'll give them info like your app name, website, and callback URL. Now since we're using Auth0 in this course, you won't have to manually go to each one of these providers, Auth0 will handle that footwork for us. So how does OAuth work? Well, first we need to understand four key roles in OAuth. The resource owner is the account owner, so if you're a user trying to log into a React app, you're considered the resource owner. The second role is the client. The client is the app that wants to access the user's account. The user must give the client permission to do so. Third is the auth server, which handles auth and provides access tokens. So for us, the auth server will be Auth0. The fourth and final role is the resource server. The resource server holds some protected information. This is the API that your application wants to access. So, for example, if your application wanted to access user data, it would make an API call to that resource server. And these four roles interact to handle authorization.

Choosing an OAuth Flow

So let's walk through a high-level overview of a typical authorization flow with OAuth. This all begins with the client app, which will send an authorization request that the user must accept. The user will see a screen that requires them to consent to sharing certain data. After the user accepts the request, the authorization grant is returned to the app. The authorization grant is data that proves that the user clicked yes to allow access. Now this is just temporary, and it's exchanged for an access token. Step three is that the app sends this grant to the authorization server. In this payload, the app includes both the authorization grant and authentication of the app's identity. Now after the app's identity is authenticated and the auth grant is validated, the auth server sends the app an access token. The access token proves that the client is allowed to call the app's protected APIs. This means that we can now use the access token to call some APIs. So let's go ahead and request some user data. The resource server is where the APIs are hosted. This could be a server you control or Auth0 itself if you're storing user data there. The access token is included in the request to prove that it's authorized. Now as long as the token is valid, the resource server returns the requested data. Now the flow above is just a general example. The exact flow will depend on the method used by your app to request authorization and the grant types supported by the API. Now OAuth specifies a number of different grants. A grant is a way to receive an access token. And remember, an access token is used to request access to APIs. A grant describes a workflow, so you'll also it called a flow in Auth0's documentation. And we just looked at a general overview of an OAuth flow, but the details can vary because OAuth specifies a number of different grants for different applications. So the OAuth flow that you choose will vary depending on the type of app that you're building. Now Auth0 provides this handy flow chart to help you choose an OAuth flow. So let's walk through this. The first question is about whether the party that requires access is a machine. An example of machine to machine communication is a scheduled job. There's no human involved. The scheduled job is the resource owner in such a case. React apps are not typically scheduled jobs, they're utilized by users, so this flow doesn't apply. React apps typically execute on the client, not the server, so this flow doesn't apply for us either. Now since React apps run in the browser, the client can't be trusted with user credentials, so this flow doesn't apply. These final two flows are most likely to apply to your React app. If you're building a React Native app, then the authorization code grant may be the best fit. But most React apps are traditional client-side rendered SPAs, also known as single-page applications. So if you're securing a React app, the Implicit Grant flow is most likely to be the best fit. So we're going to use the Implicit Grant flow in this course. Let's quickly walk through how the implicit flow works. First, your app directs the browser to the Auth0 sign in page where the user authenticates. Then Auth0 redirects the browser back to the specified redirect URI along with access and ID tokens as hash fragments in the URI. And finally, your app extracts these tokens

from the URI and stores the relevant authentication data in local storage. And this is precisely the flow that we're going to implement in our React app. Now for a quick history lesson. When OAuth first came out, people tried to use if for authentication as well, but OAuth is an authorization protocol, so it doesn't specify how to handle authentication. You'll sometimes see people use OAuth for authentication as well, but it's not recommended because there's no standard for handling authentication with OAuth. Let's review where we are. With application security, there are two key concerns, authentication and authorization. Authentication is about who you are, and authorization is about what you're allowed to do. Now as we just discussed, OAuth 2.0 handles authorization. OAuth allows your web app to access another web app's data, and OAuth keeps track of what you're allowed to do by issuing you access tokens that contain scopes. But we're leaving out a key point here, we need to authenticate the user too. We'll handle authentication using a technology that sits on top of OAuth called OpenID Connect. So, let's discuss authentication with OpenID Connect next.

OpenID Connect

Let's shift our focus now to talk about authentication. For that, we're going to use OpenID Connect. OpenID Connect solves two core problems. First, OAuth is for authorization. It provides no standards for scopes or user information requests; however, when OAuth was first introduced, people tried to use it to handle authentication too, and this led to confusion, security issues, and of course inconsistency in implementations. Second, ideally, we'd like to avoid managing passwords ourselves since it exposes us to risk and it's another thing that's easy to get wrong. So the solution to these two problems was to establish a clear standard for handling authentication called OpenID Connect, or OIDC for short. With OpenID Connect, we can authenticate users without having to manage their passwords. OpenID Connect increases security by putting the responsibility for user identity verification in the hands of expert service providers. Now OpenID Connect is a standard that sits on top of OAuth, so there are three layers here. OAuth uses HTTP to handle authentication. It answers the question what are you allowed to do? OpenID Connect sits on top of OAuth, and it answers the question who are you? So in summary, OpenID Connect is a standard way for handling authentication with OAuth 2.0. So OpenID Connect adds three key items. An ID token, which is typically a JSON web token that we'll discuss here in a moment, an endpoint for requesting user information, and a standardized set of scopes. A scope is a fancy word for permissions. So although you could add these things on your own, OpenID Connect is a standard authentication approach that integrates with OAuth. Let's step back and talk history for a moment. OpenID has actually been around for a while, but OpenID 1 didn't get much usage due

to various issues. They got the ball rolling though. OpenID 2 was much more popular, but it lacked native support and it was clunky to work with because it used XML. OpenID Connect is today's current standard. It uses JSON that's encoded in a standard called JSON Web Tokens, or JWT for short. Now any programmer who knows how to send and receive JSON messages over HTTP could implement OpenID Connect from scratch using standard crypto signature verification libraries. And you can check out openid.net for a list of OpenID Connect implementations. Fortunately, most of us won't have to as there are good commercial and open-source libraries that take care of the authentication mechanics for us. We'll use Auth0 so that we don't have to implement OpenID Connect from scratch ourselves. Let's also establish a few OpenID Connect terms. The identity provider is the entity that holds the user's information. So for example, Google, Microsoft, Facebook, or Twitter. Your app is the relying party because it's relying on this information. The idea here is simple. If your users already have an account with one of these companies, then you don't need to ask your users to choose a username or password. You can trust these identity providers instead. Now I mentioned earlier that OpenID Connect utilizes an ID token called a JSON Web Token for authentication. So, let's explore that in the next clip.

JSON Web Tokens

As I just mentioned, OpenID Connect standardized an approach to authentication on top of OAuth, and JSON Web Tokens are a token standard. They're used to pass important data around using JSON. A JSON Web Token is an access token. It's typically used for authorization and secure information exchange, so it often contains user information. Now JWT stands for JSON Web Token, but people typically just pronounce this as jot. The content inside can be verified and trusted because it's digitally signed. You can trust the data inside because you can trust that it came from the sender and that its content hasn't been tampered with. The content can be encrypted to assure security, but that's less common, so note, don't put secret data in the payload or the header elements unless the JWT is encrypted. And in this course, we will not be encrypting our JWTs. Alright, we're back here at the familiar Two Concerns slide. Now OpenID Connect authenticates who you are using an identity token. These days the identity token is typically a JWT. OAuth 2 determines what you're allowed to do and it uses an access token. The access token is often a JWT as well, although depending on your settings, it may just be a plain string. Why a JWT over SAML? Now JWTs have become the most popular authentication token, but you might have heard of SAML or used it previously. So, why has JWT become the most popular authentication token approach? Well, because JSON is less verbose than XML, so not surprisingly an encoded JWT is smaller than the equivalent SAML token too. JWTs are easier to

digitally sign than a SAML token. And perhaps most importantly, JSON is easy to parse on almost any platform, especially the web browser since it's JavaScript. A JWT has three parts: a header, a body, and a signature. The header specifies the type of the token and the hash algorithm that was used to create the token's content. The body contains the identity claims. Now I find the term claim confusing, but a claim is a simple idea. A claim is some info about a subject, typically a user. It's called a claim because it's information that the JWT is claiming is true about the user. So this example contains the claim that this user is John Doe. Common data in the body includes the user's name and the time that the JWT was issued. Finally, the signature verifies the sender and assures that the JWT's content hasn't been tampered with. Note that many of the property names are short abbreviations in order to keep the JWT compact. The header, body, and signature are each Base64 encoded and then they're separated by dots. Here's an example of a JWT. A JWT looks like a bunch of random characters because remember, that's because it's Base64 encoded. An easy way to decode a JWT is to use jwt.io. Here's an example JWT that's been decoded. Now we can see the header, which describes the algorithm and the token type, the payload, which contains the actual data such as the user information, and the signature, which is used to verify that the JWT is valid. I mentioned earlier that OAuth and OpenID Connect use two different tokens. OIDC uses an identity token and OAuth 2 uses an access token. Let's compare the information provided in each token. This is a decoded ID token. Each piece of information inside is called a claim. The identity token contains a variety of user data as you can see here. And you could also store custom claims in here as well. In contrast, the access token focuses on scopes. Remember, scopes are permissions. The only user data in the access token is the subclaim. So, this access token says that I have permission to read the user's basic profile, as well as read product data from what one could assume would be a product API. And one quick note. Unlike cookies, JWTs can't be revoked. Why? Well, because they're trusted by the client without a callback to the server. So it's important to set their life span to a relatively short period. This is why Auth0 defaults to about 10 hours.

Summary

Let's summarize by reviewing how these technologies fit together. Login is handled by OpenID Connect. You'll receive a token called a JSON Web Token, or JWT for short. This token is for the application. Since the JWT is cryptographically secure, your app can trust the content. Authorization is handled by OAuth 2.0. When you authorize an app to use your data, you receive an access token. The access token isn't for the app, it's only for API calls. Note that the access token may be a JWT or just a plain string. Later when you need to call an API, you pass your

access token for authorization. Remember that there are two types of tokens. Use identity tokens for authentication. These tokens contain user information. And use access tokens to access APIs. These tokens contain scopes which subscribe the user's permissions. In this module, we discussed the benefits of using auth providers like Auth0 instead of building your own auth solutions from scratch. And we learned some key technologies that we're going to use throughout the rest of the course including OAuth 2.0 for authorization, so it answers what the user is allowed to do, OpenID Connect for authentication, so it answers who the user is. Remember, OpenID Connect sits on top of OAuth and adds an ID token, standard scopes, and a UserInfo endpoint. Finally, we looked at JWTs, which are tokens that contain key information such as user and permission data. Alright, now that we've set the foundation, let's code. In the next module, we'll build a simple React app from scratch.

Create a React App

Intro

Now we're familiar with the core technologies that we'll be using in this course including OAuth for authorization, OpenID Connect for authentication, and JWT tokens for identity tokens and access tokens. So in this module, let's quickly build a simple React app so that we can put these technologies to use. In this module, we'll begin by creating a new React app. We'll use create-react-app, which is most popular way to create a new React app today. We'll add in React Router to handle navigation between pages.

Tools We're Using in This Course

Before we start coding, let me briefly introduce the tools that we're going to use. We're going to use create-react-app to generate the initial application structure. Create-react-app is the most popular way to create new React apps today, and it's supported by the Facebook team. To follow along, install the latest version of Node from nodejs.org. You can install either the long-term support version or the current version. Either should work just fine. I'm going to use VS Code as my editor, but you're free to use whatever editor you like. I currently prefer VS Code because it's fast, it offers a built-in terminal, and it has a huge ecosystem of extensions. VS Code also provides excellent autocompletion when working in React, and best of all, it's free.

Create App via create-react-app

Let's start coding. Through the rest of this module, we'll create the demo app structure using create-react-app and React Router. And if you're already familiar with create- react-app and React Router, then you can safely skip past this module. Just open the exercise files for this module, and then look inside the README.md for instructions on how to get started. To get started, we're going to generate our app using create-react-app. Make sure that you're running at least Node 6 and npm 5.2. Then, open up the command line and run this, npx create-react-app, and I'm going to name my app react-auth0. And when I hit Enter, you'll see create- react-app installs our dependencies, and this could take a few moments. After it completes, we should be able to cd over to react-auth0, assuming that's what you called your project, and then say npm start. You may receive a notification like this if you already have something else running on port 3000, it's okay to yes and let it open on a different port. And now we can see our React app running. I'm going to open the app in VS Code. Okay, let's review what we have here. We have a package.json file, which shows our dependencies, as well as a few scripts that we can run. And our dependencies are just react, react-dom, and react-scripts. Over in src, our entry point is right here at index.js, and there's not much to see here. It calls out to the app component. So the app component is where our application example sits. And you can see that it pulls in the logo and App.css in here, remember, webpack can handle CSS as well. Now we're going to create a couple of components that will be used as our application's pages. We're first going to create a component called Home.js. And when I come over here, I'm going to say rcc, and this won't work for you because you probably don't have this extension installed. But when I hit Enter, it generates the component for me. Now if you're curious what did this interesting magic, if you come over here into my extensions, you will see React code snippets right here. So this is the plugin that I'm using that gives me this handy little shortcut. I could also create a stateless component by saying rsc, and it will give me a stateless style like this. Let's go back and we'll keep this home page component. For now, I'm going to put in a header that says Home and hit Save. And I'll come back over here, and let's create one more component, and this file will be called Profile.js. And again, I'm going to type rcc, and it automatically populates the name here. I'm going to change this to an h1, and inside put Profile, hit Save. Another little thing you'll notice, when I hit Save, you can see that it reformats my code. That's because I am running Prettier, which is a code re-formatter. And it's running every time that I hit Save. So if you'd like to have that behavior, you can install the Prettier extension that I'm showing right here. And then, if you go into your Preferences, if you add this setting, editor.formatOnSave : true, then every time you hit Save, Prettier will autoformat your code. Totally optional, but I find that it helps me code more

quickly since I don't have to think about formatting the code. So let's go over to App.js and we can clean some things out here. We no longer need the App.css or the logo import here. And that means we can also delete this logo.svg over here. And we can delete App.css over here. And we can clear all of this out for the moment. I will just leave the div by itself. Now next up, we need to install some dependencies. So let's open up the built-in terminal, which you can also get to by going up here to View, Integrated Terminal. So I hit Ctrl+backtick to toggle the terminal. Now I'm going to install a number of different dependencies that we're going to use throughout the course. And I'm specifying the exact versions that I'm using here. Now you don't need to type this long list on your own. If you go out to this GitHub gist, then you can copy the command from the gist. And I'm going to update this gist as I confirm that newer versions work fine, so you might find that you're using newer versions than what you saw me install here on the screen. And now that we have our dependencies installed, in the next clip we can begin configuring routing.

Add Routing via React Router

Now that we have our dependencies installed and two pages created, let's use React Router to navigate between these two pages. To do that, we'll go over to index.js and make some tweaks. First, we want to import the BrowserRouter as Router from react- router-dom. So we're going to alias BrowserRouter as Router. And now we need to wrap our app with the router component and declare a single route that always renders the app component. So what we'll do is call Router. And then inside of here, declare a Route component. And we'll declare that the component that we're going to render is our App, which will reference as a variable instead of putting jsx here. I'll close out my Route and then close out Router. And when I hit Save, you'll see it reformat. Again, that's Prettier just doing its magic here. So now we have wrapped our app component within the Router. And this way all components will be able to interact with React Router. So now we're ready to declare our routes, which we can do over within App.js. First at the top, we need to import Route from react-router-dom, and then we need to import each of the components that we'd like to work with here. So. /Home, and then also import Profile,. /Profile. And now down here within render, we can declare routes for each one of these components. I will declare a Route for the root, and I will specify the exact prop here, so only this exact path will match. So this will take us to Home. And then I will add a second Route with a path to profile, and we will call the Profile component when this path matches. So if URL is just a slash, it will load Home, and if the URL contains profile, it will load Profile. So with the exact prop, the Home component will only render when the path exactly matches. In this case, when the URL path is empty. Now, as you can see, we have an error because we can only have a single top-level component, so I'm going to put in a

fragment to wrap this using the fragment syntax. And as long as you're using at least Babel 7, which is included in the latest version of create-react-app, you don't have to import fragment, this will be complied down by Babel for us, and this is valid syntax in jsx now, but this is equivalent to me importing fragment. And now if we open the terminal, we should be to run the application with npm start. But, as you can see, it fails. And this one confused me for a moment, and then I realized my mistake. Now I have yarn installed. If you're not familiar with yarn, it's an alternative to npm. And since I have yarn installed, yarn is what installed the initial dependencies when we created create-react-app. However, remember, when I installed the rest of our dependencies, I used npm install instead of yarn install. So happened was our node_modules now has some dependencies that were installed by yarn and some that were installed by npm, and that creates problems. It also means that now I have an npm style package lock and a yarn lock. So I need to decide which way to go to move forward here. So if you happen to come across this, keep in mind you just need to choose a package manager and be consistent throughout. So I am going to standardize on npm and that's what I would recommend that you do as well to follow along with me. So I'm going to delete my yarn lock, I'm going to delete the package lock, and just start over. It means I'm going to come down here, I'm going to delete node_modules with an rm -rf node_modules. Just clear everything out and do a reinstall right here. Now I could have also just come over here into the file list and deleted node_modules that way, of course, this will only work in a Bash style prompt. Now I'm going to hit npm i now, and this will reinstall our application dependencies. Great, now we can see that it installed. And if I hit npm start, looks like Route is undefined. And we can see that I have a missing import in index.js, so let's go back over here. I needed to import Route as well right here. And now when I hit Save, that error goes away. We can come over here, and refresh, and now we can see the home page is rendering. And I should be able to come up here, put profile in the URL, and see the Profile page. So great. We have routing working. In the next clip, let's add a navigation bar so that we can navigate between pages.

Add Navigation Bar

Now clearly, we're going to want to navigate between these pages, so next, let's add a simple navigation bar. We're going to go over here and create a new file and call it Nav.js. And again, I'm going to use rcc here. We're going to have some links inside this component, so I'm going to import Link from react-router-dom. Then down here in render, I'm going to declare our navigation as an unordered list. And that first list item is going to have a link to the home page in it, so I'll put a slash here, say Home. And that second Link is going to link to the profile page. Now let's jump back over to App.js. We can import our Nav component right here at the top. We can display our

Nav here above our Routes. This means that our Nav will display on every page. There we go. Now we should be able to click and navigate between our two pages. But this is looking pretty plain, so let's fix that next.

Style App

This is looking pretty plain, so let's jump over here and add some styles to index.css. Now I'm going to paste over the existing styles. You can find this CSS by looking in the exercise files, so you can save yourself some typing here. It's about 50 lines of styling. After I save these styles and come back over, we can see that the header now has a different color, but our navigation still hasn't changed much. For that, let's go back over to nav, I should've wrapped this in a nav component. That's semantically preferable, and also my styles assume that my nav is wrapped in this nav component. Once I do, look at that. Now we've got this very Pluralsight colored navigation bar, and we should be able to click between the two. One more minor tweak to make just to give us a little bit of padding. Let's go over here and wrap our Routes within a div that has a className equal to body. Hit Save, and there we go. Now we can see that our header is indented to line up right here, there we go. Developers, this is design 101. We want things to line up. Great, so now we have a working application shell. I'd say that this application's now ready for login, logout, signup. So we can augment this in the next module with Auth0. Let's wrap up with a quick summary.

Summary

Alright, app skeleton complete. In this module, we created a new React app from scratch using create-react-app. And then we created our initial app structure, and set up React Router to handle routing, and we declared our first routes. With this foundation in place, we're ready to start securing our application with Auth0. So in the next module, we'll set up login, signup, and logout.

Configure Auth0

Intro

We've laid the foundation. We have a working React app with routing and navigation set up, and we're generally familiar with OAuth for authorization, OpenID Connect for authentication, and JWT tokens as identity tokens and access tokens. So in this module, let's configure Auth0 to work with our app. We'll begin by signing up for Auth0, and then we'll make two key decisions. We'll choose an OAuth flow, and we'll select a login and signup integration approach. We'll wrap up by creating an Auth0 app that points to our React app, and we'll configure our new React app to consume the environment-specific configuration values that we set up.

Auth0 Versions Used in the Course

And a quick note before we move on. In this course, I'm using the following versions of Auth0's APIs and tooling. We'll use Auth0.js 9 and Lock 11.

Sign up for Auth0

To get started with Auth0, go to auth0.com and click on SIGN UP. On the Sign Up form, you can either register with your email, or if you already have an account with GitHub, Google, or Microsoft, then you can sign up using your existing account by clicking one of these buttons. And don't worry, you can use Auth0 for free for up to 7, 000 active users, so this won't cost you anything. Now after sign up, you need to choose a tenant domain. So, what's a tenant? It's a logical isolation unit. Now that sounds complicated, but the point is that no tenant can access data in another tenant. So I like to think of a tenant like a software development environment. And in fact, Auth0 recommends creating a separate tenant for each one of your environments, such as development, production, and QA. This allows you to make changes to one environment without affecting others. Now this might feel like a big decision because as it's mentioned here, you can't change the tenant name, but don't worry, you can create more tenants under your account if you decide that you don't like the domain that you happen to choose here. Now I'm going to create a domain for my development environment, so I'll append it with -dev, so reactjsconsulting-dev. Later, after I log in, I can create separate tenants for QA and production. On step two, you'll be asked some basic information. And for this final question, it doesn't matter what you choose since we're not going to use the onboarding tutorial. Once you log in, you're taken to the Auth0 dashboard. This is where you can manage all your settings. You may see a prompt for starting a tutorial on the right. We're going to start from scratch in this course, so you can skip the tutorial. You may see a message at the top about a trial period. Again, the free plan is

quite generous and everything we're doing in this course is included in the free plan, so don't worry about the limited trial warning.

Key Auth Decisions

Before we move forward, we need to discuss two key decisions. First, which OAuth flow should we use? And second, how should we implement login and signup? Let's begin by reviewing the OAuth flow decision that we discussed in a previous module. Remember how we walked through how to choose an OAuth flow? If you're building a React Native app, then you may want to use the authorization code flow, but since we're building a traditional client-side rendered React app in this course, we're going to use the implicit flow. Implicit flow is the recommended grant type for client-side rendered web apps. As a quick refresher, here's how implicit flow works. Your app directs the browser to the Auth0 sign in page where the user logs in. And after login, Auth0 redirects the browser back to the specified redirect URI along with access and ID tokens as a hash fragment in the URI. Then your app extracts the tokens from the URI and stores the relevant authentication data in local storage. Your app can now use these tokens to call the resource server, for example, an API, on behalf of the user. Okay, so that covers the first decision, we'll use the implicit OAuth flow. The second decision is how should we implement login and signup? There are three ways to handle login and signup screens with Auth0: Universal, Embedded, or Custom. Both Universal and Embedded use the Auth0 lock widget, so let's discuss the lock widget for a moment. Auth0 provides an optional lock widget that handles login and signup. There are a number of benefits to using it. The lock widget easily integrates with Auth0. It adapts to your configuration settings in the Auth0 dashboard, and it looks great on any device. It remembers the last used connection for a given user, and it automatically handles things like internationalization. It also implements detailed password policy checks for signups and it honors the settings that you specify in the Auth0 dashboard. As you'll see, it's quite customizable so you can tweak it for your company. You can customize the lock widget to match you color theme and customize a variety of behaviors to assure that it matches your needs. So again, these first two options utilize the lock widget, but the difference is who is hosting the widget. Universal is hosted by Auth0. It's easy to implement and the most secure since it avoids cross domain issues. Embedded is not surprisingly embedded within your app. It places the login form inside your React app. Embedded lock is inherently less secure than Universal though because it requires cross domain calls between Auth0 and your app. It's also recommended to set up a custom domain when using Embedded lock for optimal security. Now the final and third option is the Custom UI. Custom UI requires you to build your own UI and call the Auth0 APIs. Of course, this

option is the most work, but relatively straightforward since Auth0 provides an SDK and a wide variety of APIs. In this course, we're going to use the Universal login because it's recommended by Auth0, it's the most secure, it provides a single consistent login approach if you're implementing single sign on across multiple apps, and finally, it's less work to implement and maintain since a single login can serve all your apps. The single centralized login approach is quite popular today. Google is a great example of the Universal login approach. You're directed to a single centralized login page regardless of which Google property you're using. Now that we've picked a login approach, in the next clip, let's create a new app in Auth0.

Create an App in the Auth0 Dashboard

To create a new app in the Auth0 dashboard, click on NEW APPLICATION, and then we need to choose a name. I'm going to call my app react-Pluralsight. And then for application type, I'm going to choose Single Page Web Applications. Remember how we decided to use the implicit OAuth flow since it's recommended for client-side apps? Well, the implicit flow is automatically configured by Auth0 when we select this single page application type. Handy. And click CREATE. And then it's going to ask me which technology I'm using to create my app. It doesn't matter what you choose here since it merely presents you with a corresponding tutorial. We're not going to use the tutorial since I'll walk through the specific steps that apply to us here in this course. I'm going to click over here on Applications. And we can see that we have a default app that was already there and then the app that we just created. So I'm going to click on this app so that we can go into settings. And so these are the default settings for single page application. We can see our domain. Of course, yours will be different than mine. Your client ID will differ, but your name should be the same assuming you chose that. We're going to scroll down because one setting we do need to make here is specifying the Allowed Callback URL. Now this is the URL that Auth0 will call when it's done authenticating a user. So here enter http://localhost:3000/callback. So this is the URL that will get called after the user is authenticated. And we're going to specify this same callback URL within our application as we set it up. So scroll to the bottom and hit SAVE CHANGES, and you should see a confirmation that that has been saved. Now you can of course navigate around in the Auth0 dashboard to get comfortable with it. We've already looked at the applications. We don't need to install any addons or connections at this time. The defaults should be just fine. We're going to be storing any users that authenticate with us directly in a database that Auth0 supports. Later in the course, we'll set up some APIs, we'll see some users that get authenticated, we'll also customize some rules so that we can add our own behaviors in, and we'll

customize some posted pages as well. So we'll use a number of other sections in here. But for now, we're going to jump back over into the editor and begin writing some code.

Configure Environment Variables

Now let's begin integrating authentication with Auth0 in our application itself. To do this we're going to use the Auth0-js npm package, which we installed earlier with all the other packages. This package contains functions for interacting with Auth0. And the full docs for the Auth0.js library are available at this address. Now to configure our machine, we're going to create a .env file. So go to the root of the project, right-click, and say .env. So if you're not familiar with .env files, this is a common convention for specifying environment-specific variables. So you can imagine that you would have this file declared in each of your environments. So this file's going to hold our unique Auth0 information. We're going to declare a few different items in here. First, we're going to have REACT_APP_AUTH0_DOMAIN. And this needs to be set to the domain that you just selected when you registered. I'm going to show my settings page on the left-hand side so that you can see where I'm copying these values from. I'll copy the domain and set it. By the way, these settings that I'm specifying start with React app deliberately because create-react-app will automatically expose these to our application if they start with REACT_APP. Then next environment variable we need to declare is REACT_APP_AUTH0_CLIENTID. Okay, the Client ID is right here, and again, yours will be different than mine, so be sure that you pull up your settings and paste that in. The third and final setting we need to set up is REACT_APP_AUTH0_CALLBACK. For this, it's going to be exactly what we specified right down here in the previous clip. And again, this is the URL that is going to get called by Auth0 after it finishes authenticating a user. So these are our three environment variables. We can save this file. In the next clip, let's put these to use.

Create Auth Object

Now that we've stored our unique info, we're ready to start interacting with Auth0 using the Auth0 package that we installed earlier. So, let's create a new file. And first we'll create a new folder called Auth. And then inside this new folder we'll create a new file, and I'll call it Auth.js with a capital A because this is going to contain a class that we instantiate. I'm going to go ahead and go back full screen for now. And let's begin by importing the Auth0.js library. So we will say import auth0 from auth0-js. You can find the full docs for Auth0.js at this address. Then we're going to declare a class that is going to handle our authentication. So we will call this class Auth.

Inside here, we're going to declare a constructor. This constructor is going to accept history as an argument, and we're passing in the React router history. So this will allow us to programmatically interact with React router since we'll have a reference to that history object within our Auth object here. So since we're passing this in on the constructor, we will say this.history = history so that we have reference to that as an instance variable. And then we're also going to instantiate the Auth0 WebAuth. So to do that we will say auth0 = new auth0.WebAuth. Now WebAuth accepts a number of different arguments. You can see that it accepts an options object. So we are going to declare a number of different values here. First, we're going to declare the domain. And what we're going to do is reference process.env and then reference the REACT_APP_AUTH0_DOMAIN environment variable that we just declared. I'm going to copy this line and paste it twice because we're going to also pass in the clientID, which will be referencing AUTH0_CLIENT_ID. And we will also specify the redirectUri. So we will set this to AUTH0_REDIRECT_URI. I'm sorry, I called it URL over in our environment variables. I'm going to hit save and it will slightly reformat here, but if we come over here, oh, it's CALLBACK URL is what I had intended to call this. So let me add CALLBACK_URL here. I'm going to copy this, hit save, and come back over, and this will just avoid any typos here. So this should be REACT_APP_AUTH0_CALLBACK_URL. So recognize that we are able to say this process.env and reference these in our code because we prefixed our environment variable names with React app. So create-react-app will expose these automatically for us. Now there are two other pieces of configuration that we need to put here in our objects. We need to specify the response type that we would like to get back when we authenticate a user. So we want token and also an id_token. Id_token, remember when we talked about OpenID Connect, this will authenticate the user, and then token is an access token, so we'll be able to use this access token to make API calls later in the course. So we're requesting both types of tokens. And the final option that we want to specify is scope. Scope is where we specify permissions. For now, all we want is openid and we would like basic profile information on the user. Since we've requested openid in the scope, this means that we want to use openid for authentication. So the JWT that we get back will include the standard openid claim such as audience, expiration, and issued at. And since we're requesting the user profile scope, we'll receive user profile data such as name, picture, nickname, and the updated date. The exact data we receive depends on how the user signs in. Here's an overview of the user data returned from different identity providers. Now there is one notable piece of information that's not included in the profile, which is email. So we'll go ahead and add that in and request it as well. So when the user signs up for the first time, they'll be presented with a consent screen so that they can approve our use of these scopes. And we'll add some other items to the scope later.

Summary

In this module, we signed up for Auth0. Then we made two key decisions. We chose the implicit OAuth flow since that's what's recommended for client-side apps. And we're going to use the Universal login approach where Auth0 hosts the login widget for us. We created an Auth0 app that points to our React app, and we've configured our new React app to consume the environment-specific configuration values that we set up. Now we're ready to implement login in our React app, so let's tackle that next.

Implement Login

Intro

We now have a working React app and we've configured a new Auth0 application that configures our React app to work with Auth0. So, now we're ready to implement login. In this module, we'll spend our time coding. We'll implement login in our React app using the auth0-js npm package and the Auth0 Dashboard. Alright, let's get back to coding.

Setup Login

And now let's add our first method to our auth class, and we'll call this method login. Now, I'm going to use the class property syntax here because then we don't have to worry about the this keyword binding. And inside, I'm going to call this.auth0 .authorize. So this is a method that is available on the Auth0 WebAuth object. And if you're curious about which methods are available, again, you can go check their documentation. Now when authorize is called, it will redirect the browser to the Auth0 login page. And now that we've wired up the auth class, we need to share an instance of this class with our app so that we can call it from our components. So let's instantiate this authentication object over in App.js. What we can do is declare a constructor on App.js, and make sure that it will accept props, so we need to call super props on the first line. And in here, we're going to say this.auth = a new Auth object. And what we're going to pass to this auth object is this.props .history. Now notice up here that it was smart enough to add in a reference to my auth class that I just declared over here, so make sure to put this import in as well. And a quick side note. You could put the auth object in state if you want, but it's not necessary for our use in this course, so I just made it an instance variable. Now let's talk for a moment about this call. History will automatically be injected into this component on props

because in index.js we have wrapped the app in a route. So React Router will do that for us. And if we go over to index.js, the fact that App is wrapped within Router means that it's going to get the history object from React Router. So that's why we have access to this on props. And we're passing it into Auth so that Auth can interact with React Router. So now we need to pass our auth object down to our home component. And with React Router, passing props down can seem confusing at first. Since we're declaring the component right here as a prop, how can we pass a prop to the component? The solution is to use a render prop instead. So instead of saying component here, I need to say render, and what render receives is props. And then within the body we can use JSX instead to call our Home component and then pass whatever props we'd like to pass to the component, which in this case is going to be this.auth. I'm also going to use the spread operator here to pass any other props down to the component. And when I hit Save, it does reformat because Prettier sees that this line got a little bit too long. So the core change that I made was instead of doing a simple reference to a component like this, we now have a render prop. And that render prop will receive whatever props are available here on this component, and then we're telling it to render this component. And I am passing down auth, and I'm spreading any other props that are passed into this component, which in this case would be the router props that are getting passed down. So now our home component will be able to work with our authentication object. With this change, we should be ready to open our Home.js and add a login button, so let's do that right here. Say button onClick= this.props .auth .login. And I will label it Log In and close the button. And to clarify, this login method that we're calling here is the login method that we declared over within Auth.js. So we are going to call that login method when someone clicks this Log In button. Alright, well let's open the terminal and try this out. I'll hit npm start. And you can see that I get an error that clientID option is required. So, it looks like I've made a typo in my configuration. So over here I said AUTH0_CLIENTID. I forgot to put an underscore right here. After I've changed that, I need to kill the application and restart it, so hit Ctrl+C to stop the app, and then rerun npm start. When I do that, come back over here, great, now the application starts up. So now let's click Log In and see if it works. When I click Log In, great. I'm redirected to the Auth0 Log In page. Now you can see that we have a few different options. I could log in using Google or I could click on Sign Up to sign up that way. Now, I'm going to go ahead and log in via Google since I have a Google account. We're presented with a consent screen that asks us for permission to access some of our data. And then you also see a broken image up here because we haven't uploaded a logo for our application at this time. It informs me that the react-Pluralsight app is requesting access to this particular tenant. Remember that a tenant is a bit like a development environment. And we can see that it is accessing our profile and our email because remember back when we set up Auth0.js, we requested the email scope and

the profile scope, so that's why we're being notified that that's the information that our application is requesting out of Google. So when I click down here, we'll be redirected back to our application. Now that we're redirected, you can see that we were redirected back to the URL that we configured in Auth0's Dashboard, that localhost:3000/callback. What we get is a number of different useful pieces of information up here in the URL. Let's explore what's returned in this callback URL in the next clip.

Review Callback URL and JWT

Now let's pull this URL out and put it into a text document just so we can read it better. So I'm going to paste this in. And what you can do is look for the different parameters in the URL. You should find access token, and each time you find an ampersand, come in here and put in a blank line so that you can read through your URL because it will have slightly different information than mine. But it should have the same query string parameters in here. You should see access_token, which as you can see here is an opaque string, so this is not a JWT. Later, when we end up specifying a different audience for our authentication, then we will get a JWT for the access token. We can see that it expires in 7200 seconds, so that's the default. The token_type is Bearer. And then we have a little bit of state. The state parameter is an encrypted secret value used by Auth0 so that they know that we are the originating application. And finally, this is our JWT token here. So if I copy this entire JWT token, I can go over to jwt.io and check it out. So if I scroll down, I can paste it in over here on the left-hand side, and when I do, then I can see the information on the right-hand side. I can see that the type of token is JWT. That its algorithm is RS256. Here's the key ID. And then down below is all the payload data. So now we can see the data that has been returned from Google when I logged in. My name, my nickname, my picture, my gender, my email, whether my email has been verified. Who issued this token, which this token was issued from my Auth0 domain. My subscriber ID through Google OAuth. And then down below, we can also see information on the signature. So this is used to verify that the information within the JWT token can be trusted. So now we're getting a valid response back from Google when we log in, however, our application's not doing anything with it. We're stuck here at the callback page, so for the next step we need to create our callback page and parse this URL.

Parse Callback URL in Callback Component

Now that we're receiving a valid response on this callback URL, we need to handle it. We're seeing this plain white screen because we haven't created the callback page yet, so let's do that

now. I'm going to create a new file over here in src. Now we'll call it Callback.js. And inside I'm going to create a new React component called Callback. Inside, I'm going to put an h1 tag and the word Loading. You could do something flashy like a loading spinner if you prefer, but I'll keep this nice and simple. Now for this to actually work, we need to add a route over here to App.js, so let's add a new route for our callback. And that route's going to look pretty similar to our Home route, so I'm going to copy it and paste it. Again, we're going to use a render, but the path is going to be set to callback. And we can remove the exact prop. And here, we need to call a Callback instead. Now that does mean we need to import Callback up here at the top. Okay, so we have our route set up, but we need to go back over here to Callback.js because we're not quite done yet. We have a loading message rending here, but when the component mounts, we need to handle reading the URL. So let's add a componentDidMount lifecycle method. And what we want to do here is handle authentication if expected values are in the URL. So what we'll do is say let's look for these expected values using a regex. And we're going to look for the access_ token and the id_token. So I'm going to separate these by pipe characters and error. And then call .test on these. And what we're going to look at is this.props .location .hash, so this will give us the current URL. And I do have a typo right here, I need to remove this parenthesis, there we go. Okay. If those values are there, then we can go ahead and call this.props .auth .handleAuthentication. Now to clarify, we haven't created this handleAuthentication method, so we'll do that here in a moment. If for some reason we don't get the values that we expected in the URL, then we can throw a new error and say Invalid callback URL. As I mentioned, we need to implement this handleAuthentication method, but before we do, we need to decide how we're going to store our tokens. So in the next clip, let's take a step back and discuss our options for storing tokens.

Pick a Token Storage Approach

We need to decide where to store the JWT tokens that we're receiving from Auth0. We can use local storage, session storage, cookies, or just store the tokens in memory. This decision is complicated. As you can see, there are tradeoffs involved. Cookies are vulnerable to cross-site request forgery attacks, which is when a malicious website, email, or blog causes a user's web browser to perform an unwanted action on a trusted site where the user's currently logged in. Cookies also have a much smaller limit on storage, and they're sent on all requests, which wastes bandwidth. That said, if you have a dedicated server for your React app, then storing tokens in an HTTP only cookie with the secure flag enabled is generally the recommended approach. The primary benefit of storing tokens in HTTP only cookies is it helps protect from cross-site scripting since such cookies can't be accessed by JavaScript. This is why you have to generate these

cookies on a server. So, let me step back and explain cross-site scripting. Cross-site scripting is when an attacker injects client-side scripts onto your page. The risk comes from mishandling user content. For example, if the user provides content, you need to HTML encode it for display in the browser. Here's an example of a cross-site scripting attack. Imagine your app reads this data from the query string. If an attacker places some JavaScript in this query string and you display the result without HTML encoding it, then the attacker's JavaScript will run as part of your app. If unencoded content is displayed in the browser, the attacker can write JavaScript that steals another user's session by reading from local storage or cookies that aren't marked HTTP only since global JavaScript can access these items. Thankfully, React helps protect from cross- site scripting by design since it automatically escapes variables. However, there are some risky behaviors that you should avoid. If you use dangerouslySetInnerHTML to render unencoded HTML, you need to assure that the content you're displaying isn't dangerous. And it's also risky to pass user supplied values to props like hrefs on anchors. Now that we understand our options and the implications of cross-site scripting, how does Auth0 recommend storing tokens? It's not recommended to store tokens in local storage since local storage is vulnerable to cross-site scripting attacks. Ideally, we'd handle tokens on the back end since doing so provides superior cross-site scripting protection. However, there are two reasons I'm not using this approach in this course. First, if you're building a single-page app using React, you many not have a corresponding back-end server. And second, the process for handling tokens via a back-end server varies depending on your back-end technology. Of course, the implementations in Java versus .NET, Node, or Python would all look significantly different. And since this course is about React, I'm going to avoid implementing a back end to support a server-side auth flow. However, if your app has a corresponding back end, the authorization code flow is worth considering. For single-page apps without a back end, Auth0 recommends storing tokens in memory. You might be wondering, wait, if I store tokens in memory doesn't the user have to log in again if they open up the app in a new tab or close the browser? Not necessarily. There's a way to avoid this downside using a technique called silent auth, which I'll show in the final module. Bottom line, token storage is a complex decision, so I encourage you to research the tradeoffs and decide for yourself. For simplicity, I'm going to store tokens in local storage in this module. Then in the final module, we'll switch to storing our tokens in memory with silent auth, which is the approach that Auth0 recommends for single-page apps without a dedicated back end. So just keep in mind that the in-memory approach that I show in the final module is preferred.

Implement Handle Authentication

So now we're ready to implement and handle authentication. Let's open up Auth.js and we'll add this new method to our class. And we'll call it handleAuthentication. And it is going to accept a history object as its sole argument. And again, I'm going to use a class property so that we don't have to worry about the this keyword binding inside. And I'm going to say this.auth0 .parseHash. So this is a function that is built in to the auth0.js library, and what it does is, not surprisingly, parses the hash from the URL. And what we get from it is both an error object and a result, so we will handle that inside an arrow function. And first, we will check to make sure that we have received what we expected. So we're expecting to receive an authResult and we're expecting that that authResult should have an accessToken, and also that that authResult should have an idToken. And if we received the information that we expected, then we are going to create our session and store some data in local storage. So I'm going to declare that logic in a separate method below called setSession. Let's go ahead and call that method first, and we'll pass it the authResult data. So we'll implement setSession here in a moment. But in the meantime, after the session is set, we will call this.history. And a slight correction, I just realized, we aren't passing any parameters into this because we don't need to pass in the history object, we're going to have it available here as an instance variable. So I can call this.history .push right here. This is a way to programmatically tell React Router that we want to redirect to a new URL. So this is going to redirect the application back to our home page. Now let's go ahead and handle any errors that might have occurred. So we'll say else if err. We'll come in here, and again we'll go ahead and push to the home page if there's an error. But we will also do an alert and we will display the error that has occurred. So I'll say $ err.error. And I'm using a template string here so that I don't have to escape in and out of string context here. Oh, I need to remove that backtick. And I'll say Check the console for further details. And I will write to the console right here, console.log the err. So this way we'll be able to see anything that has happened. So let's summarize what we just wrote again. We're going to get the data that was passed over in the URL, we'll parse that out and get individual pieces out of it, and then we'll write that data to our session. So we need to now go declare this setSession method down here below. So let's come in and say setSession. And setSession is going to receive an authResult. And inside here, first we need to set the time that the access token will expire. So to do that, let's declare a new variable called expiresAt. And to declare this we will call JSON.stringify. And we will use the authResult.expiresIn, and multiply that by 1000, and then add information on our current date and get time from this. I'll hit Save, that will reformat. So what this is giving us then is the time that the access token will expire so that we can write this to our local storage. And this will be a handy way for us to check whether the JWT token that we're storing in local storage is still valid. Now our goal here is to calculate the UNIX Epoch time when our token will expire. The UNIX Epoch is the number of milliseconds since

January 1st, 1970. So, to calculate when our token will expire in UNIX Epoch time requires three steps. We start with authResult.expiresIn, and this contains the JWT expiration time in seconds. We multiply this by 1000 in order to convert it into milliseconds, and then we add it to Date.getTime, which is a function that returns the current UTC time in UNIX Epoch format. So this gives us the UNIX Epoch time when the token will expire. So now we're ready to save a few useful pieces of information to local storage. So we'll come down here and say localStorage.setItem. And the first thing we'll save is the access_token, which we will set to authResult.accessToken. And then I'm going to copy this line because we have two other pieces that we're going to save. We're going to also save the id_token, which we'll get from authResult.idToken. And finally, we will set expires_at to the value that we just set up above. The logic that we're placing here within Auth.js could also be placed within your React component if you prefer, however, by centralizing it here, we keep our React components simple and we allow many different React components to leverage this logic. So I prefer having this separate dedicated class that is solely focused on our authentication logic. And with this change we should be able to open up the terminal and restart the app. If I click on Log In, I'm going to select LOG IN WITH GOOGLE. Great. I ended up back at the home page. This tells me that the callback redirect worked as we expected. Now to prove that this actually worked, let's right-click anywhere on the white space and select Inspect so that we can open up the browser dev tools, and then click on Application. Now we're looking at the local storage for our app. We can see that three values have been written. By the way, if you have this window too small, then you won't actually see the list, it will only show the keys. So you might have to size this window up a bit so that you can see all these. Another thing you can do is size this window down to make more room for this list. So we have an access token, our expires_at value, and our JWT token right here for our ID token. Great. So now we've stored the authentication information that we received back from Auth0 successfully. So in the next clip, let's put this information to use.

Check if User Is Authenticated

Now that we're logged in, we don't have any visual indicator that we are logged in. So let's add a method to our auth class that can determine whether the user is authenticated. So we'll go down here to the bottom and add a new method called isAuthenticated. And inside here, we're going to check whether the current time is passed the access tokens. So let's say const expiresAt is going to equal JSON.parse. Now we're going to parse is that expires at value that we saved in local storage. So we will call getItem and get the expires_at data from local storage. And then what we can do is return a Boolean based on the Date.getTime, which remember, returns the UNIX Epoch,

and it should be less than the expiresAt that we just pulled from localStorage.getItem expires_at. So you are authenticated if we have this data here. Now given, you might wonder then, couldn't I go in and change local storage? Well, you're absolutely true. This is really a convenience method, but keep in mind that if we made a call to an API, that API is also responsible for validating any data sent over. A JWT token will have a signature that our server validates to make sure that it's valid. So, this extra data that we've stored in local storage is really just here for our own convenience. So now we can use this function on the home page to display a link to the profile page when the user is logged in. So let's go to the top and import Link from react-router-dom. Down here, we will only show the log in when the user isn't authenticated. So we are going to check this.props .auth .isAuthenticated. And if you are authenticated, then what we will do is display a Link to the profile page. And then I'm going to move this curly brace down because otherwise what we will do is show the Log In button. I'll hit Save and Prettier will reformat this for us. Now one thing I like to do just to shorten our calls is de-structure up here at the top. So I'm going to say const isAuthenticated and Login are equal to this.props .auth. And this way we can shorten our calls down here. Okay, and with this we should be able to jump over to the browser. And now we see this View profile link, so we know that we're logged in. If I come over to Profile, we're still not showing any of the user's profile data yet, but we do at least know we're logged in because we see this View profile. And in fact, we can prove that this works by coming in here and then deleting all of our local storage, and when I hit refresh, hmm, still shows me logged in. Well there's a surprise, let's go look. Oh, that's why. I need to call the function, isAuthenticated is a function. So now when I come over here, there we go. So now the Log In button shows, and there we go. Now I've got View profile. And I should find that if I come down here and clear this out and hit refresh, now the Log In button comes back.

Summary

Great. Login is working. In this module, we wired up login using the Auth0 Universal approach and the auth0-js npm package. In the next module, we'll fill in the obvious missing pieces. We'll implement logout, check out signup, and display the user's profile.

Logout, Signup, and User Profile

Intro

We're off to a great start. We can log in just fine. Now let's implement some key missing features. In this module, we use the auth0-js npm package and the Auth0 Dashboard to implement logout. Then, we'll try the signup process and review options for how the user data is stored. We'll finish up by displaying the user's profile when they're logged in. This module is all coding, so fire up your editor.

Implement Logout

We can log in, so it's clearly time to support logging out too. To do that, let's go over to our auth class, and down here we'll declare a logout function. What I'm going to do is copy these three lines from our setSession because we're going to undo what we set right here. Now I'm going to hold down Shift and Alt, which allows me to select a column within VS Code and then I can replace the set with remove. Just a nice little shortcut. And then I'll come over here and remove the second parameter because removeItem does not accept a second parameter. With that, we've now removed the items that we stored within local storage, which will log the user out. Now finally, we want to navigate to the home page after logging the user out, so we will do a this.history .push, which tells React Router to redirect. So now that we have this function, let's add a log out button to the Nav bar. I'm going to paste the button in as another li. If the user's authenticated, then it will call logout, otherwise it will call login. And the label on the button will change based on whether the user's authenticated as well. Notice that I'm making short calls here, so we do need to go up above to the top of render and add in a de-structuring statement for supporting these short calls. Since we need the auth object, let's also open App.js and pass the auth object down on props to the Nav component. And with those changes, let's try it out. We can see I'm logged in. If I click on Log Out, then I log out and the label changes as expected. If I click Log In, then an unexpected thing happens. If you notice the URL changed for a moment, but I was logged in without seeing the Log In page presented. If you're like me, you're likely thinking, what? Why? Well, because we're actually still logged in on the Auth0 server. If we jump back over here to the logout function in Auth.js, you can see that we are merely removing local data. This approach is a soft logout. By soft I mean it doesn't actually kill your session on Auth0's server. This is convenient for single sign on scenarios since it will maintain your session on other applications that are using the same Auth0 tenant. Remember, Auth0 checks your session cookie, which is stored in your browser under your Auth0 domain to determine if you're logged in. To see the session cookie that's being checked when you click Log In, load your JSON Web Key address on your Auth0 domain. Be sure to replace this section with your Auth0 domain. Under Cookies,

the Auth0 cookie with my domain is my session cookie. So Auth0 is checking this, and if it finds my session is still active on a server, it doesn't display the Log In screen. Instead it automatically logs me back in. In fact, if I delete this cookie and then come back over here to the application and click Log Out, and then log back in, notice that I'm now presented with the Log In screen again like we'd expect. Why? Because without the cookie on my Auth0 domain, Auth0 no longer knows who I am. All that being said, if you're building a single stand-alone application, then you'll likely prefer to log the user out on the server in a more traditional way. So, let's do that. I'm going to replace the call to this.history .push with a call to this.auth0 .logout. We need to pass the ClientID for our application, as well as an address that Auth0 will redirect to after the logout is completed. Now if I come back to the application and Log In, then click on Log Out, uh-oh, it breaks. Well the reason is we need to set up one more setting in the Auth0 Dashboard. Click on Applications, find your application, and scroll down until you find Allowed Logout URLs. We need to paste the logout URL that we've specified right here for security. Once we do that, we can scroll down and save our changes. After changing that, we should be able to come back over and reload our application. I'm going to Log In, now when I click on Log Out, it works. Success. Now that we have login and logout set up, let's review the signup process in the next clip.

Review Signup Process

So far, I've logged in using my existing Google credentials. Now Auth0 also supports your user signing up directly. To do so, click on Log In again, but this time we'll click on Sign Up over here. And to sign up, I'm going to enter my email address and a password that meets the default requirements. Once I do, I'm presented again with this consent screen. Again, I have a broken image here because I haven't uploaded a logo for our application yet. Now as you can see, it's asking for permission to access my profile and my email, so I'll consent. And now I have signed up using a personal email. So let's go over to Auth0 and log in to the Dashboard. Now If I come over here to the left-hand side to the Users tab, we can see that there are two users in my application. My Gmail account, which I was using earlier in the course, and now the new reactjsconsulting account that I just signed up for. And for either of these, I can click and view more information about this user. I can see the devices that they might have linked, raw JSON about their particular profile, and any applications that they have authorized. I could also, if I wanted to, come in here and delete the user or block the user, send a verification email. And for that matter, if you go check your email, you should find that you received an email from Auth0 that allows you to confirm your account. And as a side note, you can edit the content of that email as desired. If I click back over here on Details, I can also see when my latest login was and metadata about my

account, which at the moment is empty, but we'll see how we can save our own user metadata which users can have read/write access to. These are things like personal settings and data that they might have read-only access to, such as permissions information, which might say who is an admin versus a read-only role. And as a side note, by default, it is saving users out to Auth0's database, but you could choose to go to Connections, Database, and start writing user data to your own database. I'm not going to walk through that in this course, but I want to clarify that you're not forced to keep your user data out in Auth0. In an upcoming clip, I'll also show how we can use rules to augment the logic that is run each time a user signs up or logs in. In the next module, I'll show how we can turn on different identity providers, such as Facebook, Twitter, and Microsoft. Now let's wrap up this module with a quick summary.

Get User Profile

Alright, our final step for this module is to actually display some profile information when the user is logged in. To make that happen, let's go over to Auth.js and we'll add some functionality to support this. To display the user's profile, I'm going to add two functions down here at the bottom: getAccessToken and getProfile. GetAccessToken will get the access token from local storage and throw an error if the access token isn't found. Then, down here in getProfile, it will return the user's profile if it's already found. Now you notice that I'm referencing an instance variable here. This means that we should initialize it up top in the constructor. So if we already have a user profile, then we will immediately return the user profile on the provided callback to the getProfile function. If we don't have a user profile, then we will call the userInfo endpoint on Auth0 and pass it the access token. The second parameter is a callback function that receives an error in a profile. If we get a profile, then we will set this.userProfile. And finally, on 72, we call the callback and pass it the userProfile and any error that might have occurred. Finally, since we're storing the user profile in an instance variable, we should clear it out upon logout. Let's tweak the routing configuration for the user profile page on the next clip.

Configure Profile Page Route

Let's go over to App.js because right now our profile component is not receiving the auth object, so we need to pass that down. And for that matter, it would make sense for us to redirect to the home page if the user isn't authenticated. So let's do some redirects here. It's illogical for anyone to try to load the profile page when they're not logged in. So, for us to be able to pass some prompts to the profile component, we need to use a render prop instead, so I will say render

props, and then delete this out. I'll use JSX to declare our profile. I will pass it our auth object, this.auth. Then I'll also pass it any other props. And I'll hit Save, and again, Prettier just reformats a bit for me. But the thing is, if the user isn't authenticated, we don't want to load the profile page anyway. So when we come in here and have these render props, what we'd like to do is add a check right here that says this.auth .isAuthenticated. And if they are authenticated, then what we would like to do is show this user profile page. But if they are not authenticated, what we'd like to do is redirect, and we would want to redirect them to the home page. Now this redirect is currently failing because we need to import up here on react-router-dom. So we'll come up here and add Redirect like this. Also down here, I need to remove this initial curly brace and hit Save. There we go, now we have it. So, if the user's authenticated, we will show the profile page, otherwise, we will redirect them to the home page. So now you can't load the profile page unless you are logged in.

Display User Profile

So as our final step, we are now prepared to go over to the profile page and display some profile data. To begin, we're going to store user profile data in state. So we're going to store the profile and initialize it to null, and any error that we receive back will be stored as a string in state as well. I'm using the class field syntax here since it's more concise than the constructor, but feel free to use a constructor if you prefer. Let's load our user profile data using componentDidMount. However, as I discussed in my Clean Code Course, I prefer to extract well-named functions from magic functions like componentDidMount. This helps more clearly convey intent. So, in here I'm going to call this.loadUserProfile. Now we can declare this separately as loadUserProfile. Inside we're going to call this.props .auth .getProfile, which is the function that we just implemented, and remember, that function over in Auth.js expects a callback. So we'll declare our callback right here, we're going to receive profile and error. This function will use those to call this.setState and set profile and error. And notice I don't need to declare the right-hand sides here, I'm using the object shorthand syntax. With these changes, we're all set to show some data down here in render. I'm going to wrap render in a fragment so that we can have multiple child elements side by side in here. Let's put in a paragraph, and we will put the profile.nickname inside this paragraph tag, and let's also add an image tag and I'm going to add an inline style here to set the maxWidth to 50 and the maxHeight of 50 just to keep the image from being too large. I'll also set the src to profile.picture, and I'll set an alt to profile pic. Close this, and when I hit Save it reformats at bit. Finally, just so we can see all the other data, JSON.stringify, and I will call our profile and a null, and then 2. So this will display all the data that is within our profile as a string of

JSON to the page. All right. Moment of truth. If your app's not running, open up the terminal and hit npm start, and uh-oh, profile isn't defined. I have forgotten some work here. Ah, yes. So I'm referencing profile. I was planning to de-structure it and I didn't do so. So let's de-structure profile up here at the top and say this.state. Also, I should do a little check here. I should check and if the profile isn't set yet, then we'll return null. We now have my nickname displaying. I don't have an image associated with this email, so I get this standard gravatar approach. And then I can see all the other data that I could also display that's available as part of the user's profile. Now keep in mind, the fields you see on the profile page may differ from what you see here. Some identity providers collect more information than others, so you'll get different data depending on how you signed in. Let's wrap up with a quick summary.

Summary

All right. Authentication complete. In this module, we finished implementing authentication using the Auth0 universal approach. We added logout, we reviewed the signup process, and were successfully displaying the user's profile. Next up, let's turn our attention to a key concern for every react app, API calls. We'll create secure API calls that require users to have different levels of authorization.

API Authorization Fundamentals

Intro

Virtually every React app makes calls to web APIs, so this module is critical for building real world React apps. In this module, we'll implement a common use case, working with secure APIs. We'll use Node with Express to create a few simple API calls that we can host locally. We'll integrate our API server into create- react-app so it starts automatically and utilizes the same environment variable structure. And we'll configure Express to parse and validate JWTs in order to authorize users. Then, we will create two endpoints on the API, a public endpoint that anyone can call and a private endpoint that requires the user to be logged in. This module is all code, so back to the editor.

Create Environment Variables

We'll begin by adding two new environment variables to our .env file. These variables will be used for API work. First, we have the AUTH0_AUDIENCE. This is the identifier for the API that we're going to create. I'll explain the AUTH0_AUDIENCE setting more in an upcoming clip. And second, we have the API_URL. This is the URL that we're going to host our API on locally. You might think it's odd that I'm setting two different environment variables to the same value, but I'm just doing this for clarity since these two values aren't required to match. And remember, we can access these environment variables in our app because create-react-app will automatically expose them to our app because they begin with REACT_APP. Now we're ready to create our first API using Node and Express.

Create API with Node and Express

Let's create a new API using Node and Express. To begin, let's keep it simple. We'll create an API that contains a single public endpoint that anyone can call. Of course, you can host the API using any technology, but I'm going to use Node because it's quick and easy to set up using Express, and we already know JavaScript, so it's quite approachable. First, let's create a new file in the root of our project, so outside of the src directory, and we're going to call it server.js. Should find that it's in the same path as package.json. This will keep it separate from our React code. To begin, we're going to import Express. Express is a very popular Node-based library for creating APIs. Express is popular because it takes very little work to create an API. I'm also going to require dotenv.config. This package will automatically give us access to environment variables within this file. Now let's instantiate Express. At this point, I'm ready to declare my first endpoint, so I will say app.get, and what we want to do is declare an endpoint that is public. So I'm going to set the path for this endpoint to public. The final argument for each endpoint is a function that receives request and response. Inside of here we will declare what our response will be. For our first endpoint's response, we'll return some JSON under a message, and our message will be Hello from a public API. Just two more lines and we're all set. Now I'm going to call app.listen to declare which port we're going to listen on. We're going to listen on 3001. And finally, I will console.log so that we can see that it's running. We'll say API server listening on, and then I'm going to reference process.env .REACT_APP_AUTH0_AUDIENCE. And that's all it takes to configure Express in our first endpoint. So next, let's set up an npm script to start this up.

Start Express API Server via npm

We're going to use npm scripts to automate some work. If you're not familiar with npm scripts, we can put some commands right here in the scripts section of package.json, and then we can call these from the command line. To start the API server, let's create a new script called start:server, and we will call node on server.js. Let's try this out and see if it works. Say npm run start:server. Then go to localhost:3001/public. Great, it works. There's our first message from our API. Now that we've created the API, it would be handy if it started automatically when we run npm start. To do that, let's use npm run all. I'll come up to start, and I'm going to rename this to start:client, and then create a new script up above called start. In this script, I'm going to call run-p, which is the command that comes with npm run all and says to run the following scripts in parallel. We want to run the client and the server. Now come down to the command line, hit Ctrl+C to kill the previous running process, and say npm start. We should see both our application and the API start up. And there's the app running as expected. I should be able to come up here to 3001/public, and there's our API running at the same time. In the next clip, let's put React to use to call our first API.

Call Public API via React

To try out our API with React, let's create a page that will call it using Fetch. I'm going to create a new page over here in the src directory and call it Public.js. I'm going to begin by adding a new class in called Public. This class will contain some state, so I will declare some state here at the top that will hold a message. I'm also going to declare a lifecycle method, componentDidMount. In here, we're going to call fetch to call our API. We're going to call fetch and tell it to hit our public endpoint that we just configured. Fetch has a promise-based API. So I will say .then and handle the response that we received by declaring an arrow function. With Fetch you check whether the response is okay by looking at the ok property on the returned response. And if it is okay, then we need to convert that response into JSON explicitly. If the response wasn't ok, then we're going to throw an error. We'll say Network response was not ok. Now that we've converted the response to JSON, then we can call .then and handle that JSON response. What we'll do with that response is call setState to set the message to the response.message. We'll also catch any errors that might be returned and call setState to set the message to the error that we receive. Finally, our render will be very simple. I'll have a paragraph that calls this.state .message. Since we've added a new page, we should open Nav.js and add a new link. So I'll place it here below profile. We'll go to public when you click on the Public Link. Since this is a new page, we also need to open App.js and add a new route. (Typing) Be sure to add the import for public here at the top. And with those changes, we should be able to navigate to the new page. I can see that I have a problem.

And this is a very unhelpful error message, but if you come in here and hit Inspect, what you realize under the Network tab is if I refresh, I am making a call to localhost 3000, but remember that my API is hosted on 3001. If we go over to the way that we configured Public.js, we put in a relative path here. Now, create-react-app comes with support for a proxy feature, so we can jump over into package.json and add another line right here called proxy. This line tells create-react-app to proxy any calls to localhost 3000 over to localhost 3001. This conveniently avoids cross-origin resource sharing issues and error messages in development. And it allows us to have nice, short, relative URLs in our Fetch calls. With this change, should be able to come down here, stop the application with Ctrl+C, hit npm start, and try again. Excellent. Now it's working. Hello from a public API. That was simple enough, but this course is about securing React apps. So next, let's create some secure APIs that check if a user is authorized.

Create Auth0 API

To secure an API call with Auth0, we need to tell Auth0 about it. And to do that, we can create a new API definition in the Auth0 dashboard. Click on APIs over here, then click on CREATE API. We need to enter a name and an identifier. You can choose to be specific or create a separate API entry for each unique API that your app will consume. I prefer to create a single API entry since it's simpler. Handling multiple endpoints via a single Auth0 API entry means that you can use a single authorization request that gives the user an access token that will work for multiple APIs. If you create separate APIs in the Auth0 Dashboard, then you'll need to handle multiple authorization requests so that your client receives an access token for each separate API. Let's call the API Demo App API on localhost. For the Identifier, a URL is recommended. A logical choice is your API's URL. We're hosting our API locally on localhost 3001 in our dev environment, so let's use that as our identifier. When you create APIs in different environments like QA or production, you would typically set up a separate tenant for each environment, and then create separate API entries and use their corresponding URL as the identifier. This way you can experiment with configurations in the Auth0 Dashboard without affecting your production API configuration. We're going to leave the default Signing Algorithm selected, and then click CREATE. We're now presented with a quick start page that shows us how to work with APIs and different technologies. I'll walk through the necessary steps for us using Node.js. If we click on Settings, we can see the basic settings for our API including the identifier. Notice that this matches what we set up in our environment variables at the beginning of this module. Scrolling down, there's a number of different settings that you could potentially configure, but nothing that we need to change right now. We will declare some scopes in an upcoming clip, but I'll leave

these as is for the moment. Instead, let's jump back over to our code. Now that we've configured our new API in Auth0, we need to tweak the settings in Auth.js. We should specify the audience. We can do so by adding a line to the configuration object that we're passing to the WebAuth constructor. Say audience is equal to process.env .REACT_APP_AUTH0_AUDIENCE. So this is consuming the environment variable that we set up at the beginning of this module. Auth0 will send us an access token that is tied to a particular audience. The audience for our API is the audience that we've specified over here within settings. We're going to use this Identifier as the audience parameter on our authorization calls. Now to protect an API, we need to configure Express to handle JWTs, so let's do that next.

Configure Express to Parse JWTs

We called a public API in a previous clip. It can be accessed without being logged in. Now let's configure Express to handle JWTs so that we can create some secure API calls that require the user to be authorized. To begin, open server.js. We need to enhance our Express configuration to support reading a JWT. So first, I'm going to add two imports. Express-jwt validates JWTs and sets req.user. And jwksRsa will retrieve the RSA keys from the public JSON Web Key set that Auth0 exposes under our domain. Yes, that's a mouthful. I'll explain more in a moment. Next, I'm going to paste in a function that we'll use to validate our JWTs. This validates that the information inside the JWT is valid and ensures that the JWT was generated by Auth0. It uses the public signing key that Auth0 exposes for our domain. Now I know that this is intimidating looking, so let me step back and briefly review how a JWT is validated. Verifying a JWT involves two steps. First, you verify the signature, and then you validate the claims. To verify the JWT's signature, Auth0 exposes a JSON Web Key set at a dedicated URL for your domain. You can view your JSON Web Key set by going to this address on your Auth0 domain. Now this is an abbreviated example of what's returned. A JSON Web Key is a JSON object that represents a cryptographic key. The members of the object represent properties of the key including its value. Our API will use this information to verify the JWT's signature. This assures the JWT is valid. Now if you're curious, the Auth0 docs describe what each of these properties represent, but for now just understand that this data will help us verify JWTs. After validating the JWT token signature, we need to validate three standard claims in the token's payload. The expiration holds the expiration date, so you need to confirm that the expiration date is in the future. Iss holds information on who issued the JWT. And since we're using Auth0, this field should contain our app's Auth0 domain. And finally, we'll verify that the aud field holds a valid audience. Here's the code where we're validating the audience and the issuer. And we specify that the algorithm that we're going to use is RS256. This

was the default algorithm that we also selected when we set up our Auth0 API. Now we've configured Express to handle JWTs, so let's create a new endpoint that requires a JWT. Scroll down, and we'll copy the existing public endpoint and use it as a starting endpoint for our private endpoint. Set the path for this endpoint to private and then we'll say Hello from a private API. To actually make this endpoint private, we need to add a second parameter here that will check the user's JWT. Express supports declaring multiple arguments here to check the request. If any checks we declare here in the middle fail, then the request will fail. So adding checkJwt here assures that the user is authenticated because it checks that a valid JWT has been sent before they can receive a response from our private API. In the next clip, let's call this endpoint with React.

Call Private API

To call our new private API, let's create a new file called Private.js, then let's copy from Public.js and paste it into Private.js because it's a good start. We'll make a few changes. Be sure to rename. We're going to call the private endpoint instead of public. The other key change we need to make is to pass an object to configure Fetch further. The second parameter to Fetch is a configuration object, so we're going to specify the headers that we need to pass. What we need to pass is an Authorization header and we will pass a Bearer token. The token that we pass for API calls is the access token, so we will call this.props .auth .getAccessToken. This line will send the content of our access token as an Authorization header in our fetch call. We need to add a new route for this page to App.js. I'll copy our public route and change it to say private. We also need to import, so I will add the import. Finally, we need to add a link to this page to Nav.js. So again, I will copy the public link, paste it here, change this to Private. We're ready to try it out. And uh-oh, it fails. If I try loading private, I don't have access to the auth object. It's not getting passed down on props. And that means that we need to go back to where we declared our route for private. We need to pass it down, so we need to use a render prop instead. I'm going to copy this style right here so I don't have to retype. And I will reference private like this, and change the path, and then delete the previous route that we declared. Now that we're using a render prop, we can pass down that auth object on props. Let's try again. Now it's loading, but the Network response is not ok, so let's check out the Network tab. Click on Network and reload. Then if we find the private call what we see is, ah, it's what we'd expect, 401 Unauthorized. And we're unauthorized because I'm not logged in, so why don't I do that? (Logging in) And now when I click on Private, success. Hello from a private API. Great, we've called our first private API. Since we know the private page is only for logged in users, let's jump over to the Nav component and check whether the user is

authenticated. What we want to do is hide the private link if I'm not authenticated. That should do the trick. Now if I click Log Out, the Private link disappears. Let's go back to App.js and enhance this route a little bit further. If the user's authenticated, then great, go ahead and render the private component. But otherwise, let's call this.auth .login. So this will redirect the user to the log in page if they happen to try loading the private route directly when they're not logged in. Excellent. So now you know how to lock down an API call using React and Auth0. However, this is just the beginning since we're merely requiring the user to be logged in. We can have much more control by using scopes. So, let's explore that next.

Summary

In this module, we learned how to implement and authorize APIs in React using Node and Express. We configured Express to parse and validate JWTs in order to authorize users. Then we created two APIs including a public endpoint that anyone can call, and a private endpoint that requires the user to be logged in. But this module was just the beginning. In a real lab, you're likely to want more granular control. So in the next module, we'll use scopes and rules to apply different permissions to each user.

API Authorization with Scopes, Rules, and Roles

Intro

We just created our first secure API, which requires the user to be logged in. In this module, we'll enforce more granular authorization settings with scopes, rules, and roles. We'll begin by exploring OAuth2 scopes and the problem that they're designed to solve. Then, we'll create a custom scope in Auth0 and configure Express to check the scope to validate that the user is authorized to call the endpoint. We'll use Auth0's JavaScript-based rules to assign users to roles. And finally, after trying scopes and roles, we'll compare authorization approaches to clarify when each is useful. This module will give you the power to create granular permissions for each user. Let's dive in.

Intro to OAuth Scopes

OAuth2 is a protocol that allows you to share your information with another app without sharing your username or password. Each permission that you grant is called a scope. To understand OAuth2 and scopes, let's consider an example of OAuth in action. To help grow my LinkedIn network, LinkedIn asks for permission to access my email contacts. I can click the Gmail logo here to sync my Google contacts with LinkedIn. When I click on a button to sync my Google contacts, I'm presented with a consent screen from Google. So basically, LinkedIn is asking, can we have your Gmail contacts? As you can see here, LinkedIn wants permission to see, edit, download, and delete my Google contacts. These are called scopes. Scopes are permissions that you grant to an application. If I click Allow, LinkedIn will be able to access my Google data even though it doesn't have my Google password. When I click Allow, Google returns an access token to LinkedIn. This token includes the scopes I just granted, such as read:contacts, edit:contacts, and so on. Now that I've granted LinkedIn the requested scopes, LinkedIn can request my contacts from Google. Google confirms that the access token provided has the read:contacts scope and then it returns my contacts to LinkedIn. All of this happens via the OAuth2 protocol, which provides us a standard way to grant application's permission to work with our data. Each permission that we grant is called a scope. To try this out, let's create a demo that shows how to create, request, consent, and check scopes. Here's the plan. Let's pretend that you're a developer at Pluralsight. Your task is to share a list of each author's courses via OAuth. This way, Pluralsight authors can display a current list of their Pluralsight courses on their own website. So, we'll create an endpoint that returns an author's Pluralsight courses. The endpoint will require the read:courses scope to assure that the author has granted our React app permission to read their course data from Pluralsight. Here's where I ask you to pretend. We're going to add the course's endpoint to our existing Express API. This way I can show how to implement an API that checks scopes. But I'm going to ask you to pretend that the course's endpoint that we're about to create is actually hosted by Pluralsight. Here's how it'll work. When the user logs into our React app, we'll ask for consent to read the author's course data. Once the author has consented to sharing course data, our app can request the course information. Our endpoint will confirm that the user has the read:courses scope and then return the list of requested courses. Again, we're going to host the course's endpoint ourselves, but to help you imagine a real-world example, please pretend this endpoint is actually hosted by Pluralsight and returns a current list of an author's Pluralsight courses. Okay, we have a plan. In the next clip, let's implement our first scope.

Create a New Scope

We just created an API that requires the user to be logged in in order to call it. Now let's go a step further and show how we can set up fine-grained authorization using scopes. Scopes delegate permission. Scopes specify the actions an app can perform on behalf of a user. To begin working with scopes, I'm back here in the Auth0 Dashboard under the APIs section, and I'm going to click on the Demo API that we set up earlier. To configure scopes, click on the Scopes tab. We're going to declare a single scope. We'll name it read:courses. And this will grant our React app permission to read the current user's Pluralsight courses. Now we can jump back over to our code into Auth.js. We need to request the scope we created when we set up the API in Auth0. So let's add our new scope right here, read:courses. We can store the user's scopes as a separate value in local storage so that it's easy to access. So let's come down here where we set local storage and add some code. I'm going to paste in a little snippet here. And I'm going to store scopes by either looking at the scope that was returned on authentication, and if it's not set, then I will just use the requested scopes that we declared up above. And notice, I'm referencing them as this.requestedScopes. So, we need to pop back up here and change this line. I'm going to cut it and say this.requestedScopes here. And then declare this variable up above. So now we have our requested scopes in a single spot so that we can reference them here in the constructor for WebAuth, and down here when we're about to set localStorage. To set scopes to localStorage, we will set the scopes item, and then call JSON.stringify on the scopes. It's worth noting that we're storing scope information in localStorage merely for convenience. Otherwise, we'd have to parse the JWT every time we wanted to check if the user had a certain scope. You might think that this is a security problem since users can write whatever they want to localStorage, but there's no security issue here because the server will access the token and the token's content can't be tampered with since it has a cryptographic signature that's validated on the server. Since we're writing scope data on login, we should also clear it out on logout. Now we're properly requesting, storing, and removing scopes, but let's create a function down here at the bottom that checks if the user has a certain scope. I'll just paste this in is and we can talk it through. The function is called userHasScopes, and it accepts an array of scopes. It checks for the list of granted scopes by looking in localStorage for the list of scopes. If there's no scopes in localStorage, then it defaults to an empty string. It then splits on that string since scopes is a space delimited array of strings. Finally, it uses scopes.every to iterate over each scope and it returns true if every one of the scopes that have been passed into this function are found within the list of scopes stored in localStorage. In a moment, we'll call this function from our React component, but first, let's go configure Express so that it can validate scopes.

Configure Express to Validate Scopes

To try out our new admin setup using scopes, let's add a new endpoint that returns a list of courses. First, we need to import one final package, and that's going to be set to the variable, checkScope. I'm going to import express-jwt-authz. And what this does is validates JWT scopes. Now let's scroll down, and to create our new endpoint that will return a list of courses, I'm again going to copy the private endpoint we created earlier. This time, the endpoint will be courses. We're going to add another argument here so that we'll have an additional check that occurs as part of this endpoint. What we'll do is call checkScope, and then specify in this array the scope that we're requiring to be able to make a call to this endpoint. So now we're requiring that the user has the read:courses scope in their access token. Instead of returning a message, we're going to return an array of courses. I'll just quickly type in two. (Typing) Now we have an endpoint for courses that will return an array of two course objects as long as the user that requests has the read:courses scope. So we're finally ready to call this secure API from React. Let's do that in the next clip.

Create React Page That Validates Scopes

To call our new secured courses API, let's create a new page called Courses.js. And again, I'm going to copy from the Private.js file since it's a good starting point. But I will rename this component to Courses. We're going to fetch courses instead, so change it to fetch/course. And our response is not going to get a message anymore, it's going to get courses. Now down here within render, we're going to output a list of courses, so let's change that p tag to an unordered list. And we will map over the array of courses, so I will map over each course. And then in here return an li with a key set to the course's id. We'll display the course's title, and I'll hit Save. So this will display that list of courses. One other change we're going to make is we're no longer going to store message in state, we're going to store in array of courses. And I missed this left-hand side as well, this should say courses. I could write some code to display the error message, but I'll just leave it as is for now. To support routing to the new page, we need to add a route to App.js. To do that, I'm going to copy the private route and tweak it for our needs. Path will be courses instead, component that we load will be courses. And it would be a good idea for us to also check to confirm that the user has the requested scope. So I will say this.auth .userHasScopes, and check to assure that they have the read:courses scope. And if the user isn't both logged in and granted this scope, then they will be redirected to the login page. And again, I want to emphasize that even though we're checking for things here on the client, it's critical to make sure that these same security checks are on the server where your API resides. You should never trust the user. These

client-side checks are merely here to hide features that the user shouldn't be able to access. It's the API's responsibility to verify the JWT's contents in order to assure that the user has access to any requested data. Since we're referencing the course's component, it's important to come up here and import Courses. Now let's jump over to Nav.js and create a link to this page. Again, I'm going to use the private link as a starting point. It will now route to the course's page. And again, we should check whether the user has the requested scope. (Typing) I do need to scroll up here and add userHasScopes to my de-structuring statement on line 6. So now we should be able to try it out. Let's jump over to the browser. And let's try logging in. Notice now that I get a new consent screen that is asking for permission to read courses. When I do, I'm returned to the app, and I can see the new Courses link displayed up here. Now when I click Courses, I don't see a result. Looks like I got a 404 on that request. I made my request to /course. Let's go look at server.js. And that's the mistake I've made. I set the endpoint to courses, but I was making a request to course. Traditionally, APIs would state the singular here, so I'm going to change it to /course, and now this matches what I'm requesting right here in componentDidMount. Now that I have changed my configuration, we need to restart our web server. So hit Ctrl+C and restart. And there we go, now we have success. We're now viewing our courses. And that means that we must have the scope that was expected. Let's go over into Application, scroll down here, and here's our access_token. I'm going to double-click this access_token and go over to jwt.io, and then paste this access token in. We can see that I now have the read:courses scope as expected, and that's why I'm able to read the courses in the application. We've just seen how to handle authorization via scopes. In the next clip, let's look at a different authorization approach by assigning users to roles via Auth0 rules.

Assign Admin Role via a Rule

Auth0 allows you to customize the authentication pipeline's behavior using rules. Rules are written in JavaScript, and they run as part of the authentication process. Click on Rules over here in the Auth0 Dashboard and then click on CREATE YOUR FIRST RULE. Now we could start with an empty rule, or we can select from a long list of existing templates. These prewritten templates allow us to enhance access control by forcing email verification, linking accounts with the same address, or setting roles for certain users. Scrolling down, we can enrich the user's profile by adding a user's country, adding roles that are stored in a relational database, or even call a third party like FullContact to get additional information about the user. We could use webhooks to integrate with third parties, and prewritten rules exist for popular providers like Sendgrid and Firebase. Finally, down here at the bottom, there are prewritten rules for multi-factor

authentication and debugging. Let's scroll back up to the top and click on Set roles to a user. We're presented with a default implementation for this rule. Let's walk through what it does. Rules are written in JavaScript, so this should be pretty straightforward to understand. Each rule is a JavaScript function that's passed three arguments by Auth0: the user information, the context, and a callback you can call to set new values or throw an error. For more information on the data structures that you can access under user and context, check out this URL. First, the rule assures that the user has an email and that their email has been verified. Side note. If you haven't verified your email, go check your email. And if you can't find the verification email, go to your user account, click on ACTIONS, and select Send Verification Email. This rule will store the user's role in app_metadata, which the user can't change. The addRolesToUser function checks if the user's email address ends in a certain domain and then assigns the admin role if they happen to have a matching email. Otherwise, the user's role is assigned as a plain user. Now as a side note, this implementation obviously isn't very scalable, but you can look at other rules that allow you to call out to your own relational database to store this data. On line 20, the result of the function that we just looked at is stored in the roles constant, and then on 22, we assign the roles array to just set to the roles property under the user's app_metadata. Down here on 23, we call updateAppMetadata. This will write our role data to the user's app_metadata. Finally, on line 25, we set a custom claim in the idToken. This might look weird to you, but it's recommended to prepend custom claims and JWT tokens with your app's domain. This assures that your custom claims don't conflict with standard claims or new claims that are added to JWTs in the future. On 26, we call the callback, which passes our results on to any other rules that we declare. Now we only need to change two lines in this. First, up here on 12, instead of @ example.com, put in the domain name for your email. I'm going to put reactjsconsulting.com since that's the email address that I'm using to log in. Over here on line 15, I'm going to add a semicolon, that's making my eye twitch and was also causing an error over here. Finally, down here on the context token, we would logically put in the URL for our application, so I'm going to change this to localhost:3000/roles. This would of course change depending on the environment where you're running this rule. In the next clip, let's debug this rule.

Debug Rule

In order to test this rule, we can scroll down and click on TRY THIS RULE. We're presented with a dialog that contains some default test data. The top pane contains user data, and the bottom pane contains information about the context of the request. We can change any of these values to test our rule. In order to test our rule, I'm going to change the email to my email. Be sure to use

your email instead. I'm going to set the email_verified field as well since this is required within our rule. And I'm going to change the user_id to my user id. To find your user id, click on Users, find your user account, then click on Raw JSON, and copy your user_id from the Raw JSON. With this data set, we should find if we scroll down and click TRY, that the user's role is admin, and great. We can see that that's true. I could also scroll back up and change the domain, maybe if I use a Gmail domain instead. I should find if I scroll down and click TRY again that the role is set to user instead. And indeed, I see that's the case. So great. We now know that our rule is working. I could also come over here and use console.log to test any of this along the way. For instance, I could test right here and say console.log endsWith, and this will be output when I click on TRY THIS RULE. As we can see, my console.log was output here at the top of the window. Now if I scroll back up, I could also click on INSTALL REAL-TIME LOGS, and this will take me over to install an extension. When I install this extension, I'm taken over to the Extensions section, So we need to click back over on Rules, and then we can click on Edit next to our rule. Now we have a DEBUG RULE button that we can click, which will prompt us to log in. After we log in, we're presented with a window that will display a real-time log of any rules that are run. So this is a handy way to debug rules that are being run by multiple users in your environment. If I click back over here on Rules, we could rename this rule, and you could see if I have multiple rules, I could reorder those rules if desired. This can be useful when the order that your rules run is important, such as when data from one rule is utilized in a following rule. Also, note that when you're over here in the rule and you click on TRY THIS RULE, you are debugging this one rule in isolation. If you want to run all your rules at the same time, then click on TRY ALL RULES WITH and select an identity provider. And keep in mind, when you test a rule, it actually runs and changes real data in the tenant. So if we go to my user's account under the Details tab, and scroll down, we can see that my user has been granted the admin role, which is stored under app_metadata. The user can't change this data, it's read-only. You can also use rules to write user_metadata, which the user has rights to change. Typical data you might store here is user specific data that the user provides. So if you decide to enhance the signup form to request extra data, you can store the user submitted data in user_metadata so that you can access it later. Now that we've configured the rule, let's try it out in the app. If I click on Log In and use my reactjsconsulting.com email, which I set up as the admin, I should go over to Profile and see my new role. And there it is. I'm now listed as an admin. If I Log Out, and instead log in using my Google account, go to my Profile, we can see that I am just a user instead. So now our app could check for this value and display admin only functionality. But that's trivial, so I'll leave that as an exercise for you. The bigger question is what if I want to user the role on the API? That's trickier than you might expect. Let's explore that next.

Validate Roles in Access Tokens

If I log in to my admin account and paste my ID token into jwt.io, I can see that my role is admin as expected. However, check out my access token. The role isn't in here. Why? Because role is a custom claim. Custom claims aren't added to access tokens by default, and that's a problem for us because the access token is what we should typically send for authorization on API calls. Thankfully, the solution to this is easy. To solve this problem, I'm back over in the Rules section of the Dashboard and I'm going to add another rule. I'm going to start with an empty rule because what we need to do is tell Auth0 that each time a user logs in, we should add our custom claim to the access token. Remember, our custom claim had our domain name on the front, and then roles on the end. And we could read that data from app_metadata.roles. So this simple rule will Add roles to accessTokens. I'll scroll down and hit Save. I'm going to click back on Rules. And one thing that is important is that this Add roles to accessTokens is down here below Set roles to a user because we want it to run after the user's roles have been set since it relies upon this data. Now if I come back to the app and log out, and then log back in, I can inspect, and let's pull my access token out and check it out. Excellent. Now we can see that my access token is receiving the roles custom claim as expected. Now let's jump over to server.js and create an API that requires the admin role. To create our admin-only endpoint, I'm going to copy the private endpoint as a starting point, and then change a couple things. We'll make the endpoint admin, and we'll say Hello from an admin API. I'm also going to extend the middleware by calling another function called checkRole. And we will pass checkRole the role that we are going to require for this particular endpoint. Now checkRole isn't defined yet, so let's add that. Thus far, we've used npm packages to handle Express middleware, but Express middleware isn't scary. We can write our own. So let's declare this checkScope function up above. The checkRole function accepts a role to check. Express middleware must return a function that accepts three arguments: a request, a response, and next, which passes control on to the next item in the middleware chain. We check if the user's assigned roles includes the role that's been passed in. And if it does, we return next, which signifies success and allows processing to continue. If not, we throw an HTTP 401 unauthorized error and tell the user that they lack the required role. See, Express middleware isn't so scary after all. I'll hit Save here. And then to try this out, let's jump over to the Courses page and add another call. I'm going to copy the fetch course call and make a couple tweaks. We're going to call the admin endpoint instead. And instead of calling setState here, let's just console.log the response. Alright, let's open the terminal and try it out. Need to kill the process since we have changed our API, so hit Ctrl+C to stop. So here I am back in the browser. And if I click on Courses, success. I can see Hello from an admin API. And if I Log Out and instead come

over and log in as my non-admin account, which for me is my Google account, if I come back over to Courses, now I can see the 401 unauthorized, so we know that the Express middleware that we just declared to read roles is working successfully. We've just tried two different approaches to authorization, scopes and roles. In the next clip, let's step back and consider how to choose an authorization approach.

Authentication Approach Options

We just saw two approaches for handling authorization: scopes and roles. But session cookies are a popular third option to consider. Let's briefly discuss the merits of each approach. Session cookies are the traditional way to handle authentication and authorization. With this approach, you set a session cookie that contains a unique random identifier via the server, and the user stores this cookie in their browser. The session id looks like a bunch of random characters, but it's a unique identifier that identifies the user's session which is stored on the server. So each time a user calls an API, the server uses this session identifier to determine who the user is. Then it queries the database to determine the authorization rights that the user has. This approach is simple and it's secure when you use HTTP only cookies over HTTPS. It's been around for years. However, the downside is session cookies are merely an identifier. They handle authentication, but not authorization. You could think of a session cookie like an ID that you carry around. It tells somebody who you are, but not what you're allowed to do. It's your server's job to perform the lookup to determine what actions the user is authorized to perform on each call. This impacts performance since you have to query the database to look up the user's rights on every call. Admittedly, this downside can be mitigated by storing user sessions in an in-memory cache to avoid the database call overhead. The second option is to create your own custom scopes and assign different scopes to different users stored in a JWT. However, scopes were designed to handle a very specific problem, specifying what an application is allowed to do with third- party data on behalf of the user. But over time, people have tried to use custom scopes to handle general authorization within their app, and Auth0 does support this. You can assign different scopes to different users when they log in using Auth0 rules. Since JWTs are cryptographically secure, you can trust the scopes provided within the user's access token. This can improve performance by avoiding a database call to check on a user's right for each API call. However, there's a tradeoff here because using scopes for granular permissions quickly leads to bloated JWTs that contain dozens of scopes. If your app is simple, this approach can work, but you're really using scopes in a way that they weren't originally intended, and this leads to consequences. Imagine that you're building an Ecommerce app using scopes. You'd have scopes for

delete:product, edit:product, add:product, but that's just the beginning. It can quickly get more complex if different users have different permissions. You may need scopes for edit:price, edit:product-description, edit:product-sku, add:product-photo, delete:product-photo, the list goes on. The list of scopes can quickly get very long, which leads to a bloated JWT that you need to send with every request. It also leads to confusion since the scope alone doesn't convey the user's exact rights. For instance, a user may have the delete:product scope, but they may only have rights to delete a product that they created. So scopes don't scale well for large, complex authorization scenarios. And that's a good segue to our third option, which is roles. Roles group users by permissions. You grant different permissions to each role. You can include role information within your access tokens, so there's a single line to check that declares the user's role. The role approach is simple and scalable. Roles scale better than scopes because they don't lead to bloated JWTs. A single role encapsulates a long list of permissions. It's fast as well, since unlike traditional session cookies, roles are specified in a JWT that you can trust since it's cryptographically secure. This means you don't have to call the database to determine a user's rights. Remember that list of scopes for an Ecommerce site? Well, a single role can encapsulate all these scopes. Roles are also more maintainable over time. With scopes, your code can become littered with hard coded references to all of these scopes. In contrast, the role is an abstraction layer. You can add and remove permissions from a role over time without having to touch your application code. Bottom line, use scopes for their original purpose, which is for delegating permissions for your application to interact with third- party data. And use roles for handling your own app's permissions. Alright, let's wrap up this module with a summary.

Summary

In this module, we saw how scopes are an OAuth2 feature that allow us to delegate permissions to applications. With scopes, apps can utilize third-party data without needing to share a password. We created the read:courses scope in Auth0. And we used Express to authorize users by checking for the existence of the read:courses scope. We saw how rules give us the power to customize the authentication pipeline using JavaScript. We created a rule to assign users to different roles based on their email address. And we wrapped up by comparing authorization approaches to clarify when each is useful. We're almost done. In the final module, let's check out some interesting ways that you can customize React and Auth0 for your needs.

Customization and Enhancements

Intro

We've covered login, signup, and API calls. In this short, final module, let's explore some customizations and enhancements that can streamline our app's code base and configuration. We'll begin by focusing on React enhancements. We'll redirect the user to the previous page upon login. We'll create a PrivateRoute component to streamline our route declarations. We'll share the auth object via React's context so that we don't have to manually pass it down to all components. Then we'll store our tokens in memory instead of localStorage and utilize silent auth and token renewal to keep the user logged in. I'll close out by quickly showing options to tweak Auth0 to your needs including identity providers, custom login and signup, emails, database integration, hooks, extensions, and more. Alright, let's go.

Redirect to Previous Page Upon Login

Right now, after login, the user is always redirected to the Home page. Notice that I'm currently over here on the Public page. But if I click on Log In, go through the login process, I'm taken back to the Home page. It'd be convenient if it remembered where I was so that it would return me back to the Public page after I logged in. To do so, we can store the current location and localStorage before they're redirected to the login page. This means that we need to enhance some code over here in Auth.js. It should write the current location to local storage before the redirect occurs. To begin, we need to store a value in localStorage. So, let's call localStorage.setItem. And we will set the redirect_on_login in localStorage. What we will store in here is a stringified version of this.history .location. We can access the location via React Router's history object since we passed it in the constructor when we instantiated the auth function. We'll hit Save and it reformats. However, I'm going to copy this value and place it up top in a constant. I'll call this constant REDIRECT_ON_LOGIN. We are going to use the string in a few other places, so this will avoid typos and give us IntelliSense support along the way. Now that we're storing the redirect on login location in localStorage, we can read this location after the Auth0 authentication is complete. Down here after we setSession, let's store the redirect location. I'll declare a constant, redirectLocation, and look within localStorage by calling localStorage.getItem. We will want to get the REDIRECT_ON_LOGIN item. And if that item is undefined for some reason, then we will

redirect to the Home page. It should be defined since we're setting it, but this is just a safety check. If it is defined, then we will call JSON.parse and get the value from localStorage. If I hit Save, Prettier will clean this up for us. So as long as there is a value in localStorage, then we want to redirect to this location. So change this line to now use the redirectLocation that we just pulled from localStorage. We'll no longer always be pushing to the Home page, we'll redirect to a location that was stored in localStorage instead. Finally, we can clean up localStorage once we're done. So we'll call removeItem and pass it REDIRECT_ON_LOGIN. Just to see this work, I'm going to add a debugger right here in handleAuthentication. And now let's go over to the browser, open up the dev tools, and log out. I'm going to go to the Public page so that we can see whether we're redirected there as expected. I'll click on Log In, and log in, it hits my debugger as expected. Now if I go over to the Application tab, we can see that it wrote the data to localStorage. We can see that our React history object is stored here and has information on the previous path name. So now we have the information we need to redirect. If I go back to the Sources tab and hit F8, and I will move past the debugger, and great. It worked. I landed on the Public page as expected. And while we're over here tweaking React's behavior, in the next clip, let's streamline our routes by declaring a private route component.

Create PrivateRoute Component

I'm looking at the render function in App.js where we declare our routes. The code in here could be more streamlined. Notice that the route declarations for profile and private are nearly identical, except on profile I redirect to the Home page if the user isn't logged in, but on public, I'm redirecting to login. This is both inconsistent and ugly. The courses route down here is also quite similar, but has one extra check for the user scope. To simplify this code and enforce consistency, let's create our own private route component that wraps the plain route component. This way we can centralize our authentication and authorization logic to make our route configuration simpler and more declarative. First, let's create a new file, which we will call PrivateRoute.js. To save time, I'm going to paste this in and we can talk it through. It's about 40 lines of code. We import react, react-router-dom, and we declare some prop-types at the bottom. This is a functional component and we are de-structuring the props getting passed in to this component. I am aliasing the component prop with a capital C because we're going to reference this component to render it down here on line 24. We're also de-structuring the auth prop, any scopes passed in, and then we're using the rest operator here to store any remaining props that are passed in on this object, which we're calling rest. We call React Router's Route component and we pass it the props that we spread on rest. Then, we declare our render prop much like we did over in App.js, which

received some props. Our first step is to check, and if the user is not authenticated, redirect them to the login page. So it will redirect to login if the user is not logged in. If the user is logged in, then we're going to display a message if the user lacks one of the required scopes. This component will accept an array of scopes. And if there are any, and the user doesn't have those required scopes, then we will display a message that the user lacks the necessary scopes to view this page. Finally, if our scopes check and our login check have worked out, we know the user's logged in and has the correct scopes, we can render the component. We pass it the auth object and the other props that we have spread. Down at the bottom, our PrivateRoute component accepts the component that we want to render, the Auth0 auth object, and an optional array of scopes, which we default to an empty array. And that's that. In 40 lines of code, we've now centralized our logic for handling private routes. This means we can come back over to App.js and do some simplification. Let's import PrivateRoute. Then, down here we can change Route to PrivateRoute. We no longer need a render prop here. We can instead declare that the component that we want to render is the Profile component. So we can delete the rest of this code, hit Save. This just got much more declarative. Same story with our private page. Change this to PrivateRoute, change the prop to component, and then specify Private as the component that we'd like to render. Finally, for our courses page, we will call the PrivateRoute, change the render prop to component, we will render the Courses component. Instead of manually calling this.auth .userHasScopes, we can pass an array of scopes right here. So we can delete the rest of this code, hit Save. Ah, I like it. This is beautiful, much more declarative. Now let's jump back over to our app and make sure things work. Uh-oh. This fails because I forgot to pass the auth object down to these components. So I'll copy this prop and place it on our other new PrivateRoutes. With that change, I should be able to jump over here and navigate around as before. I should also find if I log out and I try to go to the Profile page, that now I'm redirected to the login page as expected. It's annoying that I have to pass the auth object down to every one of these PrivateRoute components. So in the next clip, let's use React's context to eliminate the need to manually pass the auth prop down.

Create AuthContext

Notice how App.js currently passes the auth object down to every child component. You can imagine how this can get tedious on large apps with many nested components that need to access the auth object. We might have to pass the object down on props through many layers to get to the component that needs it. This problem is commonly called prop drilling. Two popular solutions to this pain point are Redux and React's built-in Context. I explore Redux in detail in

Building Applications with React and Redux. Redux is powerful and popular, but it has a lot of moving parts, so it's intimidating to learn and configure. In contrast, Context is quite simple and it's built in to React. React has technically had Context since the beginning, but it was considered unstable and thus not recommended for use. That changed with React 16.3 when they added the new Context API. The Context API is useful for sharing data and functions that are used across your React app. We can expose the auth object via Context so any component in our application can access it. Your application can have as many contexts as you like. Typically, it makes sense to put unrelated data in separate contexts. To begin, let's declare a context in a new file under src. We'll call it AuthContext.js. First, import React. We'll call this context AuthContext. And we'll instantiate the new context by calling new React.createContext. Finally, we will export default the AuthContext that we just declared. We could optionally pass some default values to our context, but I'll leave them blank. And that's it. We've now declared a context that will hold our authentication data and functions. There are two remaining pieces to set up: the provider and the consumer. The context provider provides information and the context consumer consumes information. In the next clip, let's move on to step two and declare the provider.

Declare AuthContext Provider

We'll declare the provider first, but we have to make a decision about where to declare the provider. Typically, the provider is declared near the app's entry point. This way all your child components can access the data that that provider exposes. However, if only a portion of your app needs the data, you can declare the provider farther down the React component tree so that only a portion of the application can access the data. Our entire app is likely to need the authentication data, so let's declare the provider in App.js. First, we need to import the AuthContext that we just declared. Instead of storing auth within an instance variable, we're going to store it in state instead. So change this equal sign to a colon, or remove this semicolon, and close out my this.state call. Down here within render, I'm going to de-structure our reference to this.state. And this will allow us to shorten our calls down here in render. So we have a number of spots we can shorten now. We can say auth instead of this.auth. So I'll update all the spots down here where I'm saying this.auth. (Typing) If I click up here on the auth object, you can see the six spots that I just updated. Notice that they all just say auth instead. Now we're ready to put our context to use. Let's call AuthContext.Provider. The provider accepts one prop, which is value. We can declare here the value that this context will expose to its children. We're going to expose the auth object. We're going to come down here to the bottom and close out AuthContext.Provider. Now that we've declared this Provider, any of these child components in

here will be able to automatically access the auth object by importing the AuthContext.Consumer. This means that we no longer have to pass auth down on props to all of these different components. They can instead choose to use the AuthContext.Consumer. The best example of where this is useful is if we utilize Context in our PrivateRoute component. By doing so, we won't have to pass this auth prop down every time that we use the PrivateRoute. So in the next clip, let's update our PrivateRoute component to consume our AuthContext.

Consume AuthContext

Let's update PrivateRoute to consume the AuthContext so we don't have to pass the auth object to it. First, let's come up here and import AuthContext. Since we're importing the AuthContext, we will no longer need to pass auth in on prop, so I will remove it as a prop from this component. Instead, what I will call is AuthContext.Consumer. Context consumers expect us to declare a render prop and pass us whatever data was passed on value from the provider. So I'm going to declare a render prop. And I'm going to use a parenthesis to wrap my route component. So we can see here, we have our parenthesis that wraps the route component. So this is implicitly returning the route component from our render prop. Here's our closing curly brace. The hard part about render props is getting your syntax right. So now that we've closed the curly brace, we can see it highlighted in gray up on line 9. Now we're ready to close our AuthContext. Now that we've closed it, we need a parenthesis to close our return statement from line 7. And then finally, a curly brace to close our function itself. When I hit Save, Prettier reformats, so we know we did this correctly. So up here on line 9, this auth object will be available because we passed that auth object over as a value from our Provider within App.js right here. The fact that we declared this value means that any consumer will receive this value as a render prop. And with this work, we should be ready to test in the browser. Let's see if it works. I can click on Log In. And I can navigate around just like before. But now if we jump back over to App.js, I no longer have to pass auth down to these PrivateRoutes. This prop is no longer utilized. So now I can hit Save. If we come back over, we can still navigate the app just as before. So now we have streamlined our routes through a combination of the PrivateRoute component that we declared and the use of context, which allows us to access our auth object from PrivateRoute without needing to pass that auth object down every time we use this component. Quite handy. Let's compare the result to our work over the last few clips. Here's a before and after comparison. The result is quite compelling. Our PrivateRoute component combined with the use of context has eliminated redundant code, increased readability, and enforced consistency in our authorization approach. You could use a similar approach to remove the duplication that currently exists in our

standard route components as well. For example, you could name that component PublicRoute. Now let's shift gears. Throughout the course, we stored our tokens in localStorage, but in the next clip, let's switch to storing our tokens in memory instead.

Store Tokens in Memory

So far, we've stored our tokens in localStorage for simplicity, but when using the implicit flow, it's recommended to store tokens in memory to reduce the attack vector for a cross- site scripting attack. So, let's change our approach to store tokens in memory. To make this happen, I'm back in Auth.js. To store tokens in memory, let's declare some private variables up here above the class. Some people prefer to prefix private variables with an underscore to help convey intent, so I'll do that here. Now, scopes and expires are merely here for convenience on the client, so they're really not a security concern, but I'll move them to in-memory as well since as you're about to see, it's actually simpler. To help us find the places we need to update, I'm going to search for localStorage. So let's scroll down and start seeing what places we need to change. We can leave login as is because that's just dealing with the redirect after you're logged in. Same story with the references and handleAuthentication. That's all related to redirect on login, so we can leave it as is. Scrolling down to setSession, here's where we can begin making our updates. We can reference expiresAt directly. We don't need to call JSON.stringify anymore since we're going to be storing it in a variable. And now this becomes a one liner. Same story with scopes. This also gets simpler. We will set it to the variable at the top, and that means we don't need our call to localStorage down here for scopes or for expiresAt. Set the id_token and access_token. I'm going to hold down Alt+Shift and put an underscore on the front of this. We will set the accessToken = authResult.accessToken and the id_token = authResult.idToken. We also need to change the casing here to match JavaScript casing standards. Working our way down. This line can actually go away and we can set it on a single line. The logout section gets much simpler. We can delete all the calls to localStorage now. We can also delete this line. I should have deleted this earlier in the logout module since it's unnecessary when we redirect, it would clear out this instance variable anyway. We don't need to call localStorage here anymore, instead we can reference our private variable. And finally, down in userHasScopes, this call also gets much simpler and becomes scopes. If you have the app running and check the console, you'll see that we have a linting error from line 6 because idToken isn't used, so we can add a comment like this right above the idToken, and that will make that one go away. Of course, in a real lab, you may choose to use the idToken upon login to parse and receive the user's data, but we're calling the user info endpoint instead. And with this change, we should be ready to jump over to the browser and

make sure we haven't broken anything. If you happen to be logged in like I am, then you'll want to come over here to the Application tab, and I'm going to delete all of my localStorage since we're no longer writing to localStorage, but remember, we also removed the code that clears from localStorage. So, clearing here will help me prove that our code is now working. Now I will log back in. And if I open the dev tools and go to Application, we can see I'm logged in, but there's no longer any data written to localStorage. So that's working as expected. However, there is a quirk with the change that we've made. If I open a new tab at the same address, notice I'm not logged in even though I do have a valid session. And that's because it's stored in memory. And those in-memory values, of course, don't travel between tabs. Same story if I close a tab where I'm logged in and reopen it. I have to log in again even though my session hasn't timed out yet. So, this is secure, but not very convenient. In the next clip, let's solve this problem using a standardized approach that's part of OpenID Connect called silent authentication.

Silent Authentication and Token Renewal

There's a problem with our new setup of storing tokens in memory. The user session is lost if they open a new browser tab or close the existing tab. Auth0's JWTs expire by default after 36, 000 seconds, which is 10 hours. So our app should honor that and keep the user logged in during that period when they open a new tab or close an existing tab without logging out. To address these concerns, we can use a technique called silent authentication to renew our tokens. That sounds complicated, but it's not. We make an HTTP call to Auth0 to check if we still have an active session on our Auth0 domain. If we do, we receive new tokens in response via an iframe. The tokens are then propagated to our app from the iframe using the post message API. This process is part of the OAuth2 spec. Thankfully, Auth0 supports this flow in their auth0-js library, so we don't have to code it by hand. Alright, let's make it happen. Open Auth.js and scroll down to the bottom so we can create a new function. This new function is renewToken. Auth0 gives us a function called checkSession that makes this call behind the scenes. The first parameter to checkSession allows us to specify the audience in scope, but we can omit this parameter since it defaults to the audience in scope that we declared when we instantiated the Auth0 WebAuth object above. If there's an error, we'll log it. But otherwise, we'll call setSession with the result that we receive back. This function also accepts an optional callback that's called after the response has been received. We need to call renewToken when the app first starts up, and we need to complete this check before the app renders. So, let's make that happen by opening App.js. First, let's add another piece of state called tokenRenewalComplete, which will initialize to false. This will keep track of whether our tokenRenewal request has been completed. Next, let's add a

componentDidMount function. Inside, we will call renewToken. And after the renewToken call is complete, we will call setState and mark tokenRenewalComplete to true. Now we have the information we need to suppress rendering until tokenRenewal is complete. So we can come down here before our return, and add a simple one-liner check. If tokenRenewal isn't complete, then we will just return the words Loading. Of course, in a real lab, you'd probably want to put in a flashy spinner. But the point is we can't show the app until we know whether the user is logged in or not, so we need this check to complete first. So now when the app loads, it will receive new tokens via an iframe behind the scenes if the user session is still active on the Auth0 server. Alright, let's give it a shot. I'm back in the browser. I'm going to log out first, and then click on Log In. I'm going to log in directly with Auth0. Be sure that you do the same to follow along with me. The Google login won't work, and I'll explain why in a moment. If I try to load the page, it fails. So, let's go down and hit Inspect. If we go to the Network tab, I refresh, we can see the failed call right here. And the response is a page from Auth0 saying oops, something went wrong. There could be a misconfiguration in the system or a service outage. Not a particularly helpful message. But thankfully, easy to fix. To fix this issue, come over to your application Settings, scroll down, and we need to set Allowed Web Origins to our app's address. Since we are making a call from this address, for security purposes we need to tell Auth0 that that's okay. After making that change, my application does load. I can click Log In, and again, log in using your Auth0 account. And great, I logged in successfully. And now if I hit refresh, I can see that I stay logged in, so it's working. Excellent. As a little performance tweak, you could write a value to localStorage upon login so that this check doesn't occur when someone is visiting for the first time or after they've clicked log out, since in those cases, they're clearly not logged in, so there's nothing to check. But I'll leave that as an exercise for you. Now note, there are some caveats with this approach, which is why I saved it for the end. First, this approach relies on setting third-party cookies via an iframe, so if you have disabled third-party cookies in your browser, this won't work. Safari blocks third-party cookies by default. To avoid third-party cookie issues, it's recommended to set up a custom domain with Auth0. Second caveat is if I try to log in with Google, it fails. I can log in successfully, but if I hit refresh, you can see that it looks as though I'm not logged in. That's because silent authentication doesn't work with identity providers by default. You need to configure your own keys with each provider. That said, this isn't actually any extra work. Before you go to production, you need to configure your identity provider keys anyway. And as a side note, I didn't cover that topic in this course. The default Google config that we've used throughout this course is configured with the development key from Auth0 that you will need to replace before publishing your app to production. One final related tweak we can make is silent token renewal. This way our app can request new tokens automatically when our current tokens expire. This requires very little

code, so let's jump back to Auth.js to make it happen. I'm going to add a schedule renewal function down here at the bottom. This will call renewToken when our token expires. And we should call this after the user is authenticated, so let's go back up to setSession and call it at the bottom. Great. This will ask Auth0 for a new token when our current token expires. So, this will keep the user logged in while the tab is open until their Auth0 session actually expires. Alright, that's a wrap. In the next clip, I'll show you a variety of customizations that you can consider.

Overview of Auth0 Customization Options

You might be wondering whether Auth0 is sufficiently customizable to your needs. The short answer is almost certainly yes. There are a huge number of customization options to consider, so let's briefly review a few. You can integrate with popular single sign on services including Active Directory, box, Salesforce, and New Relic. Under Connections, you can specify how to handle usernames and password policies. You can require a username, specify minimum username length rules, and disable signups so you can handle them behind the scenes. Under Password Policy, you can specify password strength rules. Move the slider to change the required password strength. And you can try different combinations over here on the right. You can also forbid using a previous password, disallow certain passwords, and disallow passwords that contain personal data. We stored user information with Auth0, but you can use your own database if you prefer by configuring database action scripts. Under Social, you can enable over a dozen different identity providers with a single click of the button. This is a huge time saver over building these integrations yourself. Users can log into your app using their Enterprise credentials such as Active Directory or PingFederate by integrating Auth0 with an Enterprise connection. And maybe you want to avoid passwords altogether. With this approach, the user clicks a button and is sent a one-time code in either a text message or an email each time they want to log in. The email and text message's content is fully customizable, as is the one-time password expiration and code length. We looked at Rules earlier in the course. Rules allow you to customize the authentication pipeline with your own logic written in JavaScript. You can also consider Hooks, which allow you to use Node.js to customize Auth0's behavior with code that runs before or after the user registration, or during the client credentials exchange. Auth0 supports multi-factor authentication as well via push notifications and SMS, as well as via Google Authenticator and Duo. And here's a section you'll likely want to tweak. You can customize the login, signup, and password reset pages as well. Click on Customize Page and you're presented with a window down here below that allows you to tweak the HTML, JavaScript, and CSS that are displayed on the login and password pages. You can change this code to tweak the look of the Auth0 page. You can change

the logo, the primary color and application icon, the title and placeholder text for each field, how the identity provider login buttons are displayed, and you can even add custom fields to the signup form. You can also select predefined templates such as the password list template that I've selected here. Click on Preview to see the results of the template that you've selected. Or come back over and select Custom Login Form for complete control over the look and feel of your login form. And if you'd like to embed the login form within your app, that's an option too. You can embed the Auth0 lock or embed a completely custom login that calls Auth0's APIs. You have similar customization options for the password reset page. Users are sent emails for various events including account verification, password resets, welcome emails, and more. You can control the content and subject of each email, and you're also free to select the provider that will send these emails as well. Auth0 provides detailed logs of actions performed in the Dashboard and authentication by users, as well as brute-force protection that is enabled by default to limit the number of signups and failed logins from a given IP address. Finally, you can enable Extensions to further enhance Auth0's capabilities including custom social connections, sending logs to Logstash or Azure, GitHub, GitLab, or GitBucket deployments, and a powerful authorization extension that adds granular support for groups, roles, and permissions. So, chances are you can tweak Auth0 to your needs.

Summary

In this final module, we enhanced our auth object to redirect the user to the previous page upon login. We created a PrivateRoute component to streamline our route declarations. We shared the auth object via React's context so that we don't have to manually pass it down to all components. And we stored our tokens in memory and kept the user logged in using silent authentication and token renewal. Then, I wrapped up by showcasing a variety of Auth0 features and customizations that you might find useful in your next React app. Now you're ready to handle authorization and authentication in your next React app. Thanks for watching.

Course author

Cory House

Cory is the principal consultant at reactjsconsulting.com, where he has helped dozens of companies transition to React. Cory has trained over 10,000 software developers at events and businesses...

Course info

Level	Intermediate
Rating	★★★★★ (81)
My rating	★★★★★
Duration	3h 18m
Released	30 Nov 2018

Share course