

Authentication and authorization

One of the first questions we're asked is usually about authentication and authorization in a serverless environment. Without a server, how does one authenticate users and secure access to resources? To help answer these questions, we introduce an AWS service called Cognito and another (non-AWS) service called Auth0. We also introduce the AWS API Gateway and show how to use it to create an API. We show you how to secure this API using custom authorizers and connect it to Lambda functions. Lastly, we show how to extend 24-Hour Video to provide sign-in, sign-out, and user-profile facilities by combining features of Auth0, API Gateway, and Lambda.

5.1 *Authentication in a serverless environment*

In modern web and mobile applications, authentication and authorization can take a number of forms. Allowing users to directly sign up with the application or sign in via an enterprise directory is important. It can be equally important to allow users to authenticate with a third-party identity provider (IdP) such as Google, Facebook, or Twitter. You might ask how one implements and manages all the required authentication, authorization, user sign-up, and user validation concerns without a server. The answer is by using services such as AWS Cognito and Auth0 and technologies such as delegation tokens. Before we discuss these services and technologies in more detail, you may want to look at appendix C. This appendix serves as a nice refresher on the topics of authentication and authorization, OpenID, and OAuth 2.0.

5.1.1 *A serverless approach*

Authenticating a user and then authorizing access to needed services may seem like a challenge without a server, but it isn't difficult once you understand what's possible:

- You can use services such as Cognito (<https://aws.amazon.com/cognito>) or Auth0 (<https://auth0.com>) to help implement an authentication system.
- You can use tokens to exchange and verify user information between services. In this chapter, you'll use JSON Web Tokens (JWT). These tokens can encapsulate necessary information (claims) about the user. Your Lambda functions can verify that a token is legitimate and then allow execution to continue if everything is okay. You can even check the validity of a token in the API Gateway before the relevant Lambda function is run (more on that in section 5.3).
- You can create delegation tokens using a Lambda function or Auth0. Delegation tokens can be used to authorize direct access to services from the front end.

Figure 5.1 shows what a possible authentication and authorization architecture may look like in a serverless application. The process of authentication is managed using Auth0, which takes care of authentication and creation of delegation tokens needed for direct authentication with other services. As you can see in the figure, the client can access the database directly or send requests to a Lambda function that can access the database using its own credentials. You have flexibility in choosing the best approach for your system.

JSON Web Tokens

Throughout this chapter we'll refer to JWT, which stands for JSON Web Token. The Internet Engineering Task Force (IETF) describes JWT as a "compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted" (<http://bit.ly/1Spxog6>). See the section on JWT in appendix C to learn more.

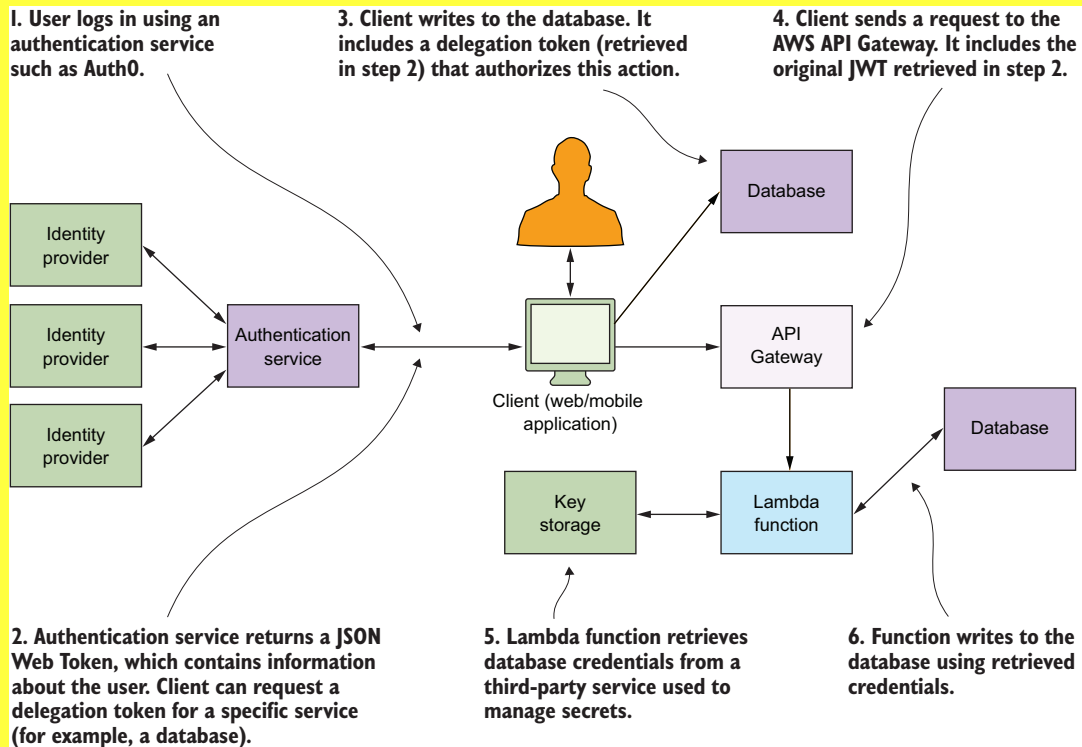


Figure 5.1 In a serverless architecture, you should let the client interact with services directly where it makes sense.

In chapter 1, we told you that in serverless architecture “the presentation tier of the application communicates directly with services, the database, or compute functions via an API gateway. Many services can be accessed directly by the front end. Some services need to be hidden behind compute service functions where additional security measures and validation can take place.” This description stands in contrast to many traditional systems where communication often flows through a back end that coordinates access to the database and services. So when designing your serverless system for authentication and authorization, remember the following points:

- Use an established, industry-supported method for authentication and authorization such as OpenID Connect and JWT.
- Make use of delegation tokens and allow the front end to communicate directly with services (and the database) when it makes sense (that is, do this only when an interruption to the client won’t place the system in an inconsistent state and only if it’s secure to do so).

Delegation tokens

Later in the book, in chapter 9, we're going to show how to use a JWT-based delegation token to authorize access to your database and to other services. JWT is great, but it's not supported by all services everywhere. There will be times when you'll have to use signatures or temporary credentials instead. In this chapter (and beyond) assume that delegation tokens are JSON Web Tokens. But if we use other ways of granting temporary access to services, we'll clearly mention it.

Making things easier in the long term

When building your serverless architecture, try to reduce the number of steps your system has to take to perform an action. Allow your front end to communicate with services directly if it's secure and appropriate to do so. This will reduce latency and make the system easier to manage.

Furthermore, don't come up with your own way of performing authentication and authorization. Try to adopt common protocols and specifications. You're likely to integrate with multiple third-party services and APIs that implement these as well. Security is difficult, so if you follow tried-and-tested models for authentication and authorization, you're more likely to succeed.

5.1.2 *Amazon Cognito*

As a developer, you can build your own authentication and authorization system if you wish to do so. OpenID Connect and OAuth 2.0 can help you support external identity providers. Add a Lambda function, a database, and a sign-up/sign-in page, and you can begin to authenticate users. But why build when someone else might have already done it? Let's look at existing services to see if they can reduce the amount of work you would normally have to do.

Amazon Cognito (<https://aws.amazon.com/cognito>) is a service from Amazon that can help with authentication. You can use it to build an entire registration and login system, and it can integrate with public identity providers or your own (existing) authentication process.

Authenticated and unauthenticated users going through Cognito are assigned an IAM role/temporary credentials. This allows users to access resources and services in AWS. Cognito can also save end-user data. This data can be synced and accessed across different devices. Figure 5.2 shows how a user can authenticate with an identity provider and then get access to a database in AWS. Cognito acts as an intermediary (see <http://amzn.to/1SmsmPt> for more information on Cognito authentication flows).

Cognito is a great service but it has a number of limitations. Useful features such as password reset require a bit of manual implementation and don't have some of the more advanced features such as log on via TouchID. Cognito is a great system but there's another alternative we should explore: a service called Auth0.

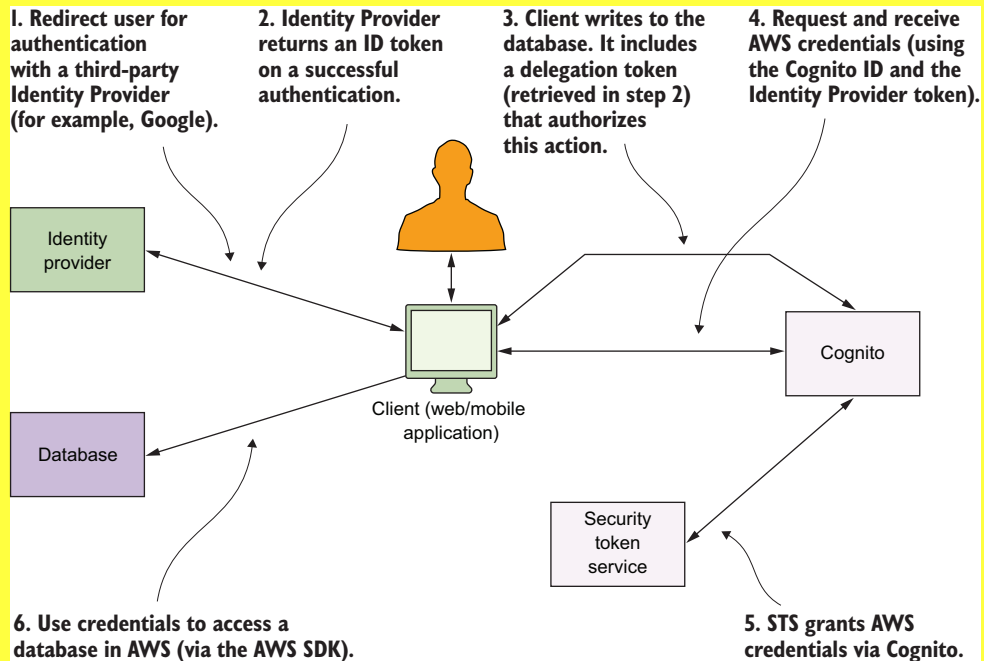


Figure 5.2 The (enhanced) authentication flow with Cognito and a security token service (STS), which grants temporary AWS credentials. Assume here that the client uses AWS SDK to invoke the resource (that is, the database in this figure) directly.

5.1.3 Auth0

Auth0 (<https://auth0.com>) can be labeled a universal identity platform. It supports custom user sign-up/sign-in with a username and a password, integrates with identity providers that use OAuth 2.0 and OAuth 1.0, and connects to enterprise directories. It also has advanced features such as multi-factor authentication and TouchID support.

When a user authenticates with Auth0, the client application receives a JSON Web Token. This token can be used in a Lambda function if it needs to identify the user, or it can be used to request a delegation token (from Auth0) for another service. Auth0 integrates well with AWS. It can obtain temporary AWS credentials to securely access AWS resources, so you don't lose anything by using Auth0 instead of Cognito (for more information about integration with AWS see <https://auth0.com/docs/integrations/aws>).

Cognito and Auth0 are both very capable systems. You should explore the unique features they offer and make an assessment based on the requirements of your project. In the next section, we'll explore how to handle user authentication in a serverless application using Auth0 and JWT.

5.2 Adding authentication to 24-Hour Video

In this section, you're going to add sign-in/sign-out and user-profile features to 24-Hour Video. You'll use Auth0 to handle user sign-up and authentication, and we'll show you how to secure access to Lambda functions. So far, we've focused only on building the 24-Hour Video back end and neglected the front end. You're now going to build an interface so that users can interact with the system (figure 5.3.)

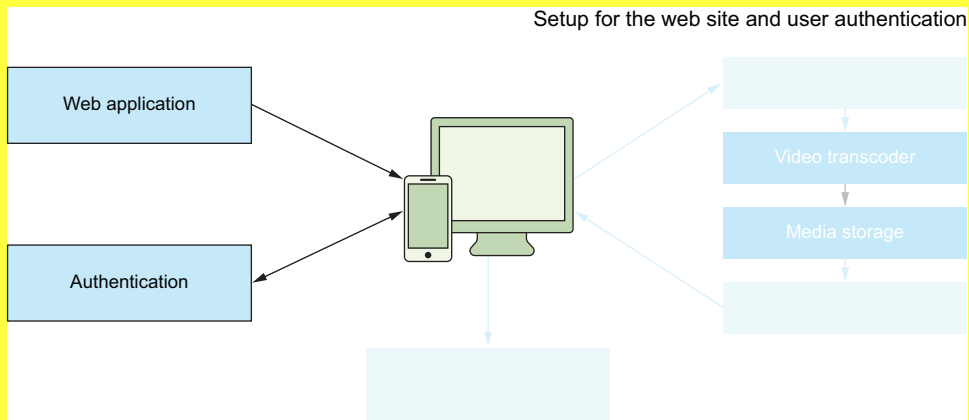


Figure 5.3 In this chapter you're going to add authentication and begin building your website.

Furthermore, we'll introduce the AWS API Gateway in more detail. You can use this AWS service to create an API between back-end services and the front end. The API Gateway is covered in more detail in chapter 7, so feel free to jump to it if you need further information or clarification as you follow this example.

5.2.1 The plan

The plan for adding an authentication/authorization system to 24-Hour Video is as follows:

- 1 Create a basic website to serve as a user interface. It will have sign-in, sign-out, and user-profile buttons. In later chapters, you'll add additional capability to this website such as video playback.
- 2 Register an application with Auth0 and integrate it with the website. Users should be able to log in via Auth0 and receive a JSON Web Token that identifies them.
- 3 Add an API Gateway to allow the website to invoke Lambda functions.
- 4 Create a user-profile Lambda function. This function will decode the user's JWT and invoke an Auth0 endpoint to get more information about the user. It will then return this information to the website via the API Gateway. For the moment, you don't have a database, so there isn't any additional information

you can store about the user. But after chapter 9, you'll have a database in which you can save extra user information.

- 5 Configure the API Gateway to invoke the `user-profile` Lambda function using an HTTP GET request.
- 6 Modify the API Gateway to perform JWT validation before the request hits an integration endpoint (that is, before the request reaches your Lambda function). You'll create a special Lambda function to validate your JWT and connect it to the API Gateway as a custom authorizer to run on every request.

Figure 5.4 shows the authentication/authorization architecture you're going to build in steps 1–5. Step 6 is described in more detail in section 5.3.5.

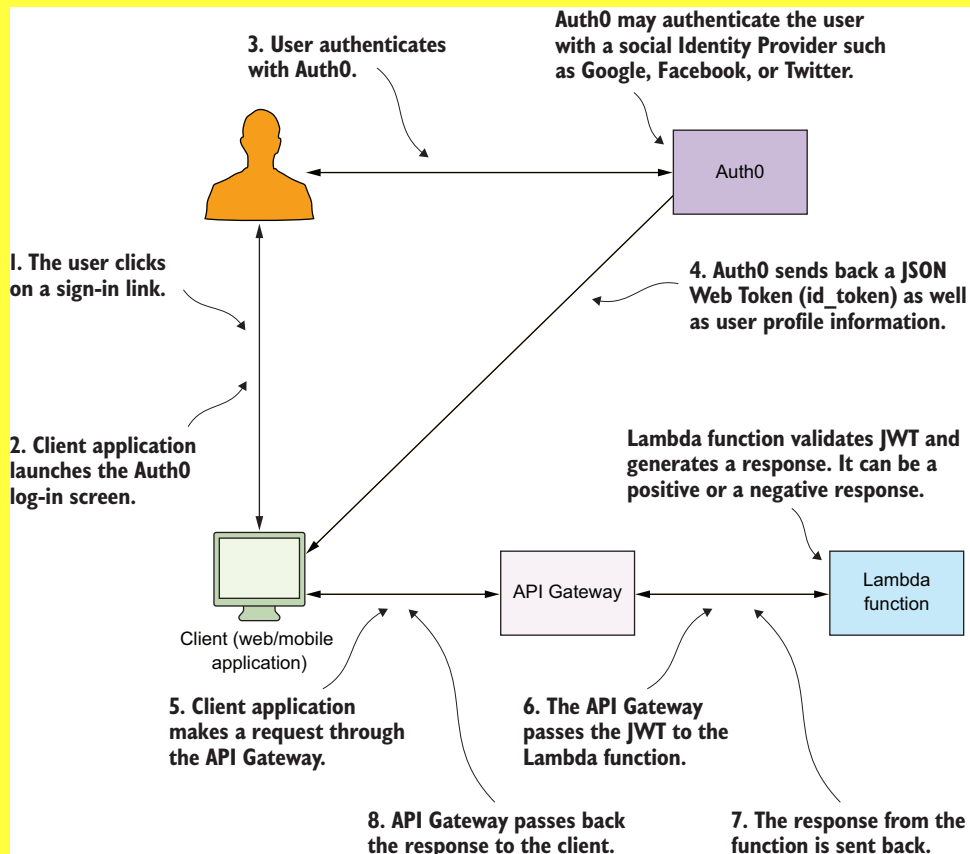


Figure 5.4 The basic authentication/authorization flow with Auth0 and JWT that you'll implement for 24-Hour Video

5.2.2 *Invoking Lambda directly*

At this stage, you might ask, why can't I get temporary AWS credentials and invoke Lambda directly from the 24-Hour Video website? Why do I need an API Gateway at all? Those are fair questions. You do have two ways of invoking Lambda functions. One way is to use the SDK; the other is to go through an interface created by the API Gateway. If you use the SDK approach, it would mean the following:

- The user would have to download a portion of the AWS SDK.
- 24-Hour Video would become coupled to specific Lambda functions. Changing these functions later could become painful and might require a redeployment of the website.
- It would be harder to prevent a rogue user from abusing the system and invoking Lambda thousands of times. With the API Gateway, you can throttle requests, authorize requests, and even cache responses.
- The API Gateway allows you to design and build a uniform RESTful interface that other clients can interact with using simple HTTP requests and standard HTTP verbs.

When it comes to Lambda and a web application, creating a RESTful interface using the API Gateway and putting your functions behind it is the way to go.

5.2.3 *24-Hour Video website*

If you're building a large web application today, you might choose one of the available single-page application (SPA) frameworks such as Angular or React. For the purposes of this example, you're going to create a website using Bootstrap and jQuery. The reason for doing so is to allow you to focus on the serverless aspects of the system rather than configuration and management of an SPA framework. If you wish to use your favorite SPA rather than vanilla JavaScript and jQuery, feel free to do so. You'll be able to follow this example with a few minor tweaks. Figure 5.5 shows what this basic website will look like initially.

A quick way to create a skeleton website is to download the Bootstrap version of the Initializr template (you can accept all default settings when downloading) from <http://initializr.com>. Extract the download to a new directory such as 24-hour-video. You're going to make changes to this website and install additional packages. To help manage dependencies and later to perform deployments, you'll use npm as you did for Lambda functions in chapter 3. Open a terminal window and do the following:

- 1 Change to the website directory and run `npm init` from it. Answer questions from npm to create a `package.json` file.
- 2 You'll need a web server to host your website. A good module you can use is `local-web-server`. Run the following command from the terminal to install it:

```
npm install local-web-server --save-dev
```
- 3 Modify `package.json` to look like the following listing. It will allow you to run `npm start`, which then launches the web server and hosts the website.

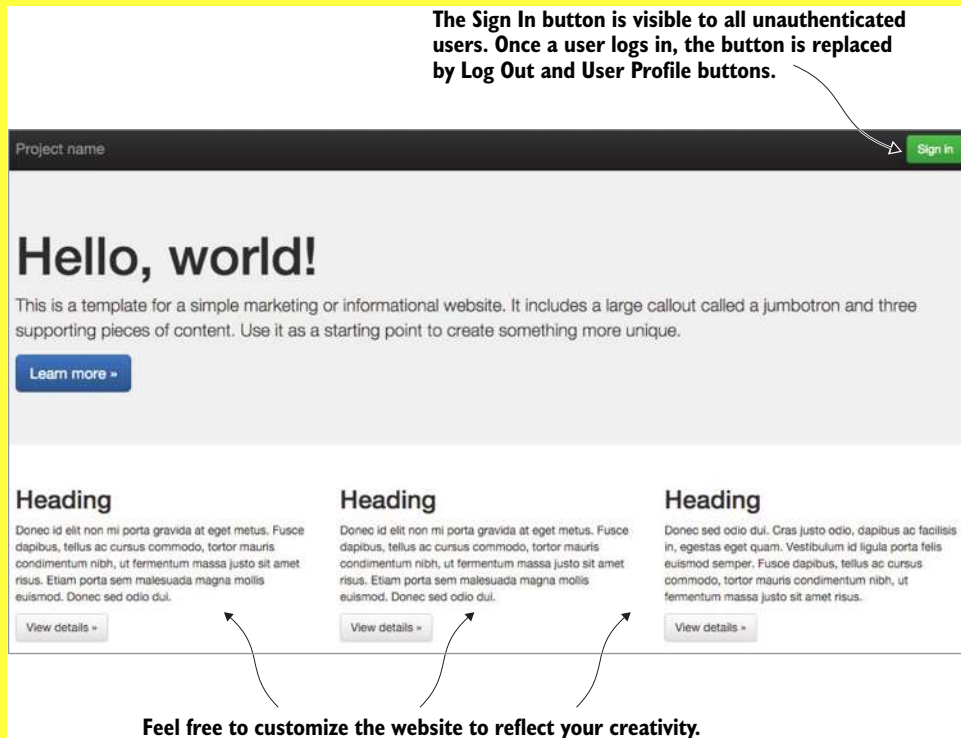


Figure 5.5 The Initializr Bootstrap template looks like this with an added Sign In button.

Listing 5.1 Package.json for the website

```
{
  "name": "24-hour-video",
  "version": "1.0.0",
  "description": "The 24 Hour Video Website",
  "local-web-server": {
    "port": 8100,
    "forbid": "*.json"
  },
  "scripts": {
    "start": "ws",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Peter Sbarski",
  "license": "BSD-2-Clause",
  "devDependencies": {
    "local-web-server": "^1.2.6"
  }
}
```

Port 8100 is unlikely to clash with other open ports on your system, but you can change it to anything you want.

Running npm start will launch the web server.

Your version number could be different but that's okay. Everything should still work.

Run `npm start` from the terminal and open `http://127.0.0.1:8100` in your web browser to see the website.

5.2.4 Auth0 configuration

Now you can integrate Auth0 with the website. Register a new account at <https://auth0.com>. You'll need to type in a preferred Auth0 account name, which could be anything (for example, your organization or website name) and select a region (choose US West). After creating the account, you might see an Authentication Providers pop-up. In this pop-up you can choose the types of authentication to offer to your users. They include standard username and password authentication, as well as integration with Facebook, Google, Twitter, and Windows Live. You can configure additional connections or remove the ones you've chosen later.

You'll start with a default app in Auth0 that you can use as a basis for 24-Hour Video. You'll be given an option to choose an application type (figure 5.6). Select Single Page App and then select jQuery. You'll be taken to a documentation page that describes how to configure Auth0 for your website. You can always refer to this page for additional information, and you should because Auth0 documentation is excellent. For now, however, click the Settings tab that's under the Default App heading.

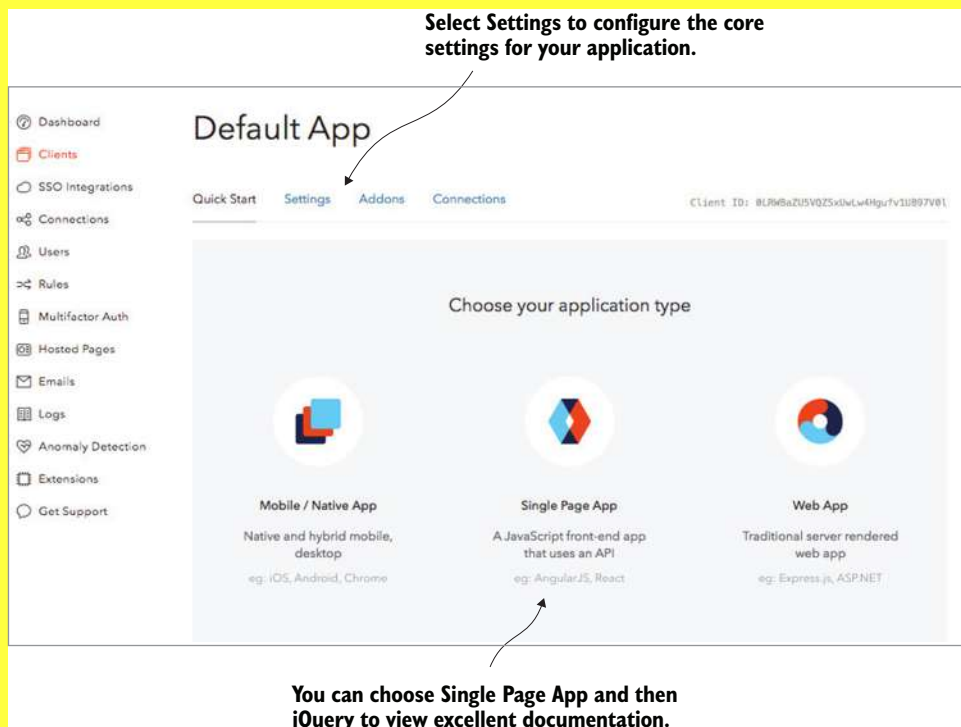


Figure 5.6 Auth0 has a sparse and easy-to-use dashboard. We love it.

In the Settings tab you'll need to configure a couple of options (figure 5.7):

- 1 From the Client Type drop-down select Single Page Application (if it's not already selected).
- 2 In Allowed Callback URLs type `http://127.0.0.1:8100`.
- 3 Click Save Changes at the bottom.

Auth0 will send responses to only the URLs that are specified in Allowed Callback URLs. If you forget to specify your website URL there, Auth0 will show an error during sign-in.

The screenshot shows the Auth0 Settings interface for a client. The 'Domain' is 'serverlessheroes.auth0.com'. The 'Client ID' is '0LRWBaZU5VQZ5xUwLw4Hgufv1UB97V0I'. The 'Client Secret' is masked with asterisks, with a 'Reveal client secret' checkbox and a note 'The Client Secret is not base64 encoded.' The 'Client Type' is set to 'Single Page Application'. The 'Allowed Callback URLs' field contains 'http://127.0.0.1:8100'. A 'Save Changes' button is at the bottom right. Two callout boxes provide additional context: one points to the 'Client Secret' field stating it will be used in a Lambda function to verify tokens, and another points to the 'Allowed Callback URLs' field stating that Auth0 will only respond to these URLs.

Domain: serverlessheroes.auth0.com

Client ID: 0LRWBaZU5VQZ5xUwLw4Hgufv1UB97V0I

Client Secret: [masked]

☐ Reveal client secret.

The Client Secret is not base64 encoded.

Client Type: Single Page Application

The type of client will determine which settings you can configure from the dashboard.

Allowed Callback URLs: http://127.0.0.1:8100

After the user authenticates we will only call back to any of these URLs. You can specify multiple valid URLs by comma-separating them (typically to handle different environments like QA or testing). You can use the star symbol as a wildcard for subdomains ('*.google.com'). Make sure to specify the protocol, `http://` or `https://`, otherwise the callback may fail in some cases.

This client secret will be used later in a Lambda function to verify the authenticity of the token.

Set the allowed callback URLs. Auth0 will not send a response otherwise.

Figure 5.7 Use the Auth0 Settings screen to get the client secret and set allowed callback URLs.

You should also look at Connections (on the left menu) to see which types of integrations, such as database-driven, social, enterprise, or password-less, you could use. If you click Social under Connections, you'll see a list of third-party authentication providers that you can enable for your web application (figure 5.8).

Having two or three social connections enabled is usually enough for most applications. Users will get confused and use multiple accounts to sign in to the system. When that happens, you'll get questions from people asking why their account is different or why things are missing. It's possible to link accounts together, but that's outside the scope of this chapter; see <http://bit.ly/1PRKiRe> if you need more information on how

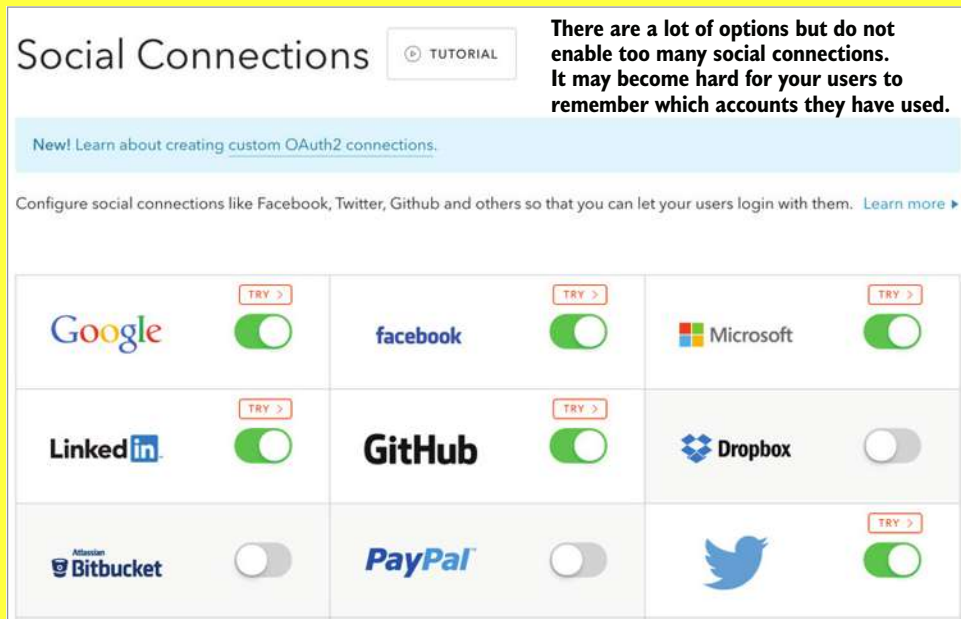


Figure 5.8 Auth0 supports social, database, and enterprise connections.

to do it. Note that the free Auth0 account supports only two social identity providers. If you want to use more, you'll have to go on a paid plan.

If you decide to enable integration with a third-party identity provider such as Google or GitHub, you'll need to do a bit of configuration. When you click an identity provider in Auth0, you'll see the information, such as an API key, that needs to be entered. Auth0 always provides a link to a page that explains how to obtain needed keys, client IDs, and secrets (figure 5.9). For 24-Hour Video, make sure to enable and configure at least one identity provider, such as Google or GitHub, to see how it works. An exercise at the end of the chapter will ask you to do this.

5.2.5 Adding Auth0 to the website

In this section, you'll connect the website to Auth0. The user will be able to register and sign in to Auth0 and receive their JSON Web Token. This token will be stored in the browser's local storage and included in every subsequent request to the API Gateway. The user will also be able to sign out, which will remove the token from local storage. Figure 5.10 shows this part of the workflow. Please be aware that in a real system, including this token in every request isn't a best practice. You should control where the token is sent so that third parties don't accidentally intercept it. An exercise at the end of the chapter asks you to address this problem.

Auth0 Lock is a free widget from Auth0 that provides a nice-looking sign-in/sign-up dialog box. It simplifies the authentication flow and has a few interesting features

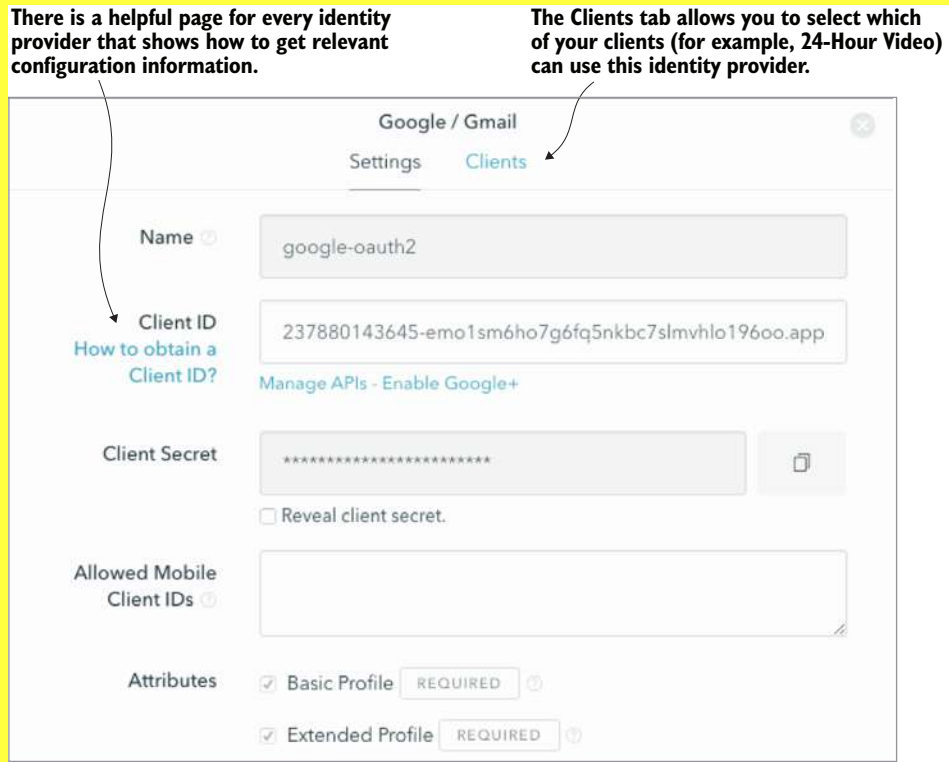


Figure 5.9 Auth0 has guides to help you find key information from third-party authentication providers.

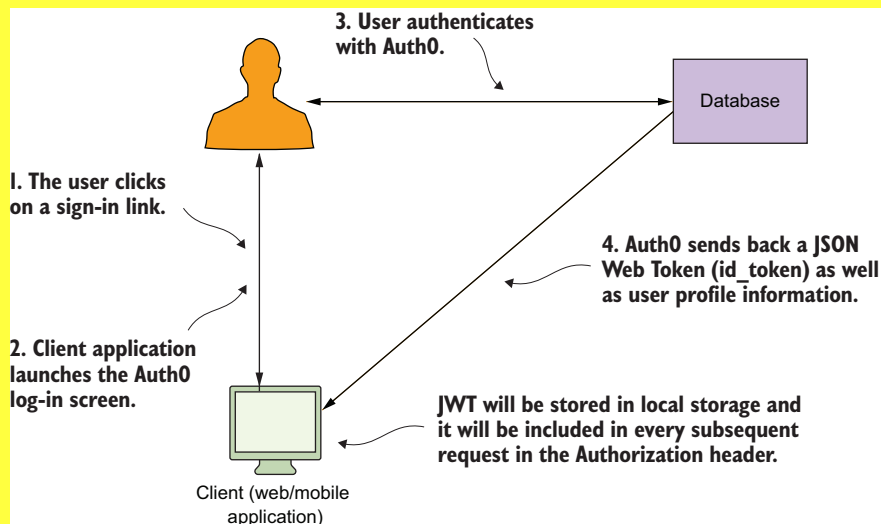


Figure 5.10 Now that you've completed this section, your users will be able to sign into and out of the website.

(for example, it can remember which identity provider the user used in a previous session). You're going to use this, so next we'll look at the following:

- Adding Auth0 Lock to the website
- Adding sign-in, sign-out, and user profile buttons
- Adding a sprinkle of JavaScript to show the login dialog and save the JWT token in local storage once the user authenticates

To add Auth0 Lock to the website, follow these steps:

- 1 Open index.html in your favorite HTML editor.
- 2 Add `<script src="https://cdn.auth0.com/js/lock-9.min.js"></script>` above the line that says `<script src="js/main.js"></script>` (which is at the bottom of the file).
- 3 To add buttons, remove the login form beginning with the line `<form class="navbar-form navbar-right" role="form">` and replace it with the code in the next listing.

Listing 5.2 Adding buttons to index.html

```
<div class="navbar-form navbar-right">
  <button id="user-profile" class="btn btn-default">
    <img id="profilepicture" />&nbsp;<span id="profilename"></span>
  </button>
  <button id="auth0-login" class="btn btn-success">Sign in</button>
  <button id="auth0-logout" class="btn btn-success">Sign Out</button>
</div>
```

The profile picture will be retrieved via Auth0.

These buttons will trigger the click event in user-controller.js.

You need to add JavaScript to wire up the buttons. Create the following two files in the js directory of the website:

- user-controller.js
- config.js

Now add the following lines above `<script src="js/main.js"></script>` but below `<script src="https://cdn.auth0.com/js/lock-9.min.js"></script>` in index.html:

```
<script src="js/user-controller.js"></script>
<script src="js/config.js"></script>
```

Copy the next listing to user-controller.js. This code is responsible for initializing Auth0 Lock, wiring up click events for the buttons, storing the JWT in local storage, and then including it in every subsequent request in the Authorization header.

Listing 5.3 Contents of user-controller.js

```

var userController = {
  data: {
    auth0Lock: null,
    config: null
  },
  uiElements: {
    loginButton: null,
    logoutButton: null,
    profileButton: null,
    profileNameLabel: null,
    profileImage: null
  },
  init: function(config) {
    var that = this;

    this.uiElements.loginButton = $('#auth0-login');
    this.uiElements.logoutButton = $('#auth0-logout');
    this.uiElements.profileButton = $('#user-profile');
    this.uiElements.profileNameLabel = $('#profilename');
    this.uiElements.profileImage = $('#profilepicture');

    this.data.config = config;
    this.data.auth0Lock =
      new Auth0Lock(config.auth0.clientId, config.auth0.domain);

    var idToken = localStorage.getItem('userToken');

    if (idToken) {
      this.configureAuthenticatedRequests();
      this.data.auth0Lock.getProfile(idToken, function(err, profile) {
        if (err) {
          return alert('There was an error getting the profile: ' +
            err.message);
        }
        that.showUserAuthenticationDetails(profile);
      });
    }

    this.wireEvents();
  },
  configureAuthenticatedRequests: function() {
    $.ajaxSetup({
      'beforeSend': function(xhr) {
        xhr.setRequestHeader('Authorization',
          'Bearer ' + localStorage.getItem('userToken'));
      }
    });
  },
  showUserAuthenticationDetails: function(profile) {
    var showAuthenticationElements = !!profile;

    if (showAuthenticationElements) {
      this.uiElements.profileNameLabel.text(profile.nickname);
      this.uiElements.profileImage.attr('src', profile.picture);
    }
  }
}

```

The Auth0 client ID and domain will be set in the config.js file.

If the user token already exists, you can try retrieving the profile from Auth0.

This token will be sent in the Authorization header in all future requests. Doing this may be insecure, so in section 5.5 we ask you to fix it.

```

        this.uiElements.loginButton.toggle(!showAuthenticationElements);
        this.uiElements.logoutButton.toggle(showAuthenticationElements);
        this.uiElements.profileButton.toggle(showAuthenticationElements);
    },
    wireEvents: function() {
        var that = this;

        this.uiElements.loginButton.click(function(e) {
            var params = {
                authParams: {
                    scope: 'openid email user_metadata picture'
                }
            };

            that.data.auth0Lock.show(params, function(err, profile, token) {
                if (err) {
                    alert('There was an error');
                } else {
                    localStorage.setItem('userToken', token);
                    that.configureAuthenticatedRequests();
                    that.showUserAuthenticationDetails(profile);
                }
            });
        });

        this.uiElements.logoutButton.click(function(e) {
            localStorage.removeItem('userToken');

            that.uiElements.logoutButton.hide();
            that.uiElements.profileButton.hide();
            that.uiElements.loginButton.show();
        });
    }
}

```

Auth0 Lock will display a dialog and allow users to register and log in.

Save the JWT token to browser's local storage.

Clicking Logout removes the user's token from local storage, makes the Login button visible, and hides the Profile and Logout buttons.

Copy the code that follows to config.js. Remember to set the correct client ID and Auth0 domain.

Listing 5.4 Contents of config.js

```

var configConstants = {
    auth0: {
        domain: 'AUTH0-DOMAIN',
        clientId: 'AUTH0-CLIENTID'
    }
};

```

The Auth0 domain and client ID can be obtained from the Auth0 dashboard (figure 5.6).

Copy the code in the next listing to main.js.

Listing 5.5 Contents of main.js

```

(function() {
    $(document).ready(function() {
        userController.init(configConstants);
    });
})();

```

Run the userController.init function to wire up events and set up Auth0.

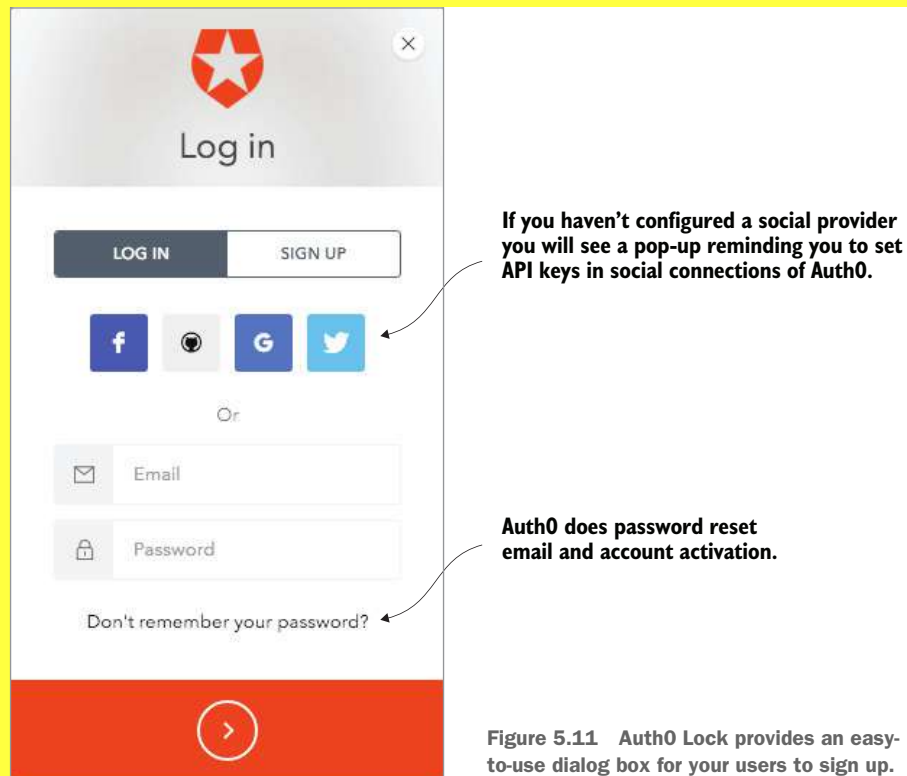
Finally, modify `main.css` (located in the `css` directory of the website) to have the styles given in the following listing.

Listing 5.6 Contents of `main.css`

```
#auth0-logout {  
  display: none;  
}  
  
#user-profile {  
  display: none;  
}  
  
#profilepicture {  
  height: 20px;  
  width: 20px;  
}
```

5.2.6 Testing Auth0 integration

To test Auth0 integration, check that the web server is running in the terminal. If it isn't, then run it by executing `npm start`. Open the page in the browser and click the Sign In button. You should see the Auth0 Lock dialog (figure 5.11). Sign up right now (note that you're creating a new user for the 24-Hour Video app; this isn't the same



user you used to sign up to Auth0 in the first place), and Auth0 should immediately sign you in to your website. The JWT should be transmitted and saved in the browser's local storage (if you use Chrome, you can open Developer Tools, select Storage, click Local Storage, click <http://127.0.0.1:8000>, and you'll see the `userToken`). Click the Sign Out button to log out and delete the JWT from local storage.

Go back to the Auth0 dashboard and click Users. You'll see all users registered with the site. You can contact, block, delete, view location, or even sign in as a different user. If you signed in successfully before, you should see your user details in the list.

If something didn't work and you couldn't sign in, open your browser's developer tools and inspect the Console and Network tabs for any messages from Auth0. Double-check that you set the Allowed Callback URL in Auth0 to be the URL of your website, and check that you have the correct client ID and the domain.

5.3 Integration with AWS

Now you're going to create a Lambda function that will accept the JWT from the website, validate it, and then request more information about the user from Auth0. You could issue a request to Auth0 straight from the browser and get information about the user that way. You don't need a Lambda function to do this, but this example is designed to show how to deal with JWT in Lambda and, a little later, some of the code will serve as a basis for your custom authorizer.

As we mentioned earlier, there are two ways to invoke a Lambda function: using the AWS SDK or via an API Gateway. We'll go with the second option, so you need to create an API Gateway. Your website will issue requests to an API Gateway resource and include JWT in the Authorization header of the request. The API Gateway will capture requests, route them to the Lambda function, and then send Lambda responses back to the client. Figure 5.12 shows this part of the workflow.

You'll now work on the custom API (figure 5.13).

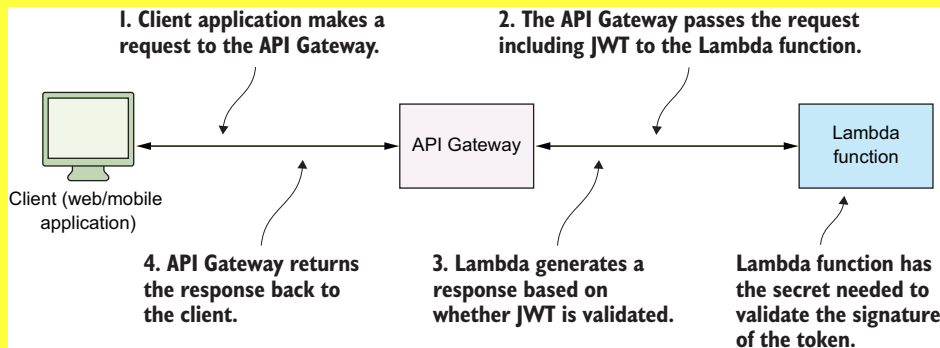


Figure 5.12 The website invokes a Lambda function via the API Gateway. The request includes JWT in the Authorization header.

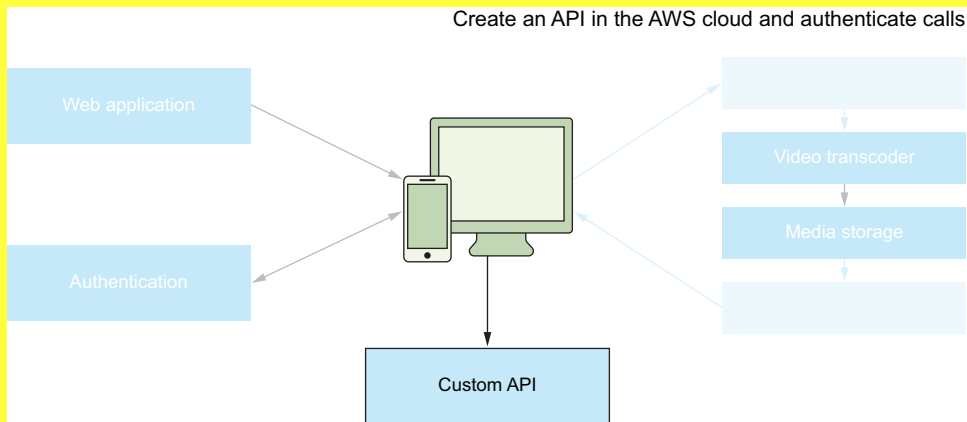


Figure 5.13 You'll build and use your custom API throughout the next few chapters.

5.3.1 User profile Lambda

Before implementing a user profile Lambda function, you should create a new IAM role for it. You could reuse the role you created earlier (`lambda-s3-execution-role`), but it has one too many permissions you don't need. So let's see how to make a new one with fewer permissions:

- 1 Create a new role in the IAM console.
- 2 Name it `api-gateway-lambda-exec-role`.
- 3 In step 2 of the role-creation process, select AWS Lambda.
- 4 From the list of policies, select `AWSLambdaBasicExecutionRole`.
- 5 Click Create Role to save.

Having created a new role, you can focus on the Lambda function. This function will do the following:

- Validate JSON Web Tokens.
- Invoke an Auth0 endpoint to retrieve information about the user.
- Send a response to the website.

Create the function in AWS right now:

- 1 Click Lambda in the AWS console.
- 2 Click the Create a Lambda Function button and select the Blank Function blueprint.
- 3 Click Next on the Triggers screen.
- 4 Name the function `user-profile`.
- 5 Select `api-gateway-lambda-exec-role` from the Existing Role drop-down.
- 6 Leave all other settings as they are, and then save and create the function.

On your computer, set up the function:

- 1 Make a copy of one of the Lambda functions you worked on in chapter 3.
- 2 Change the name and any relevant metadata in package.json (remember to update the function name or the ARN in the deploy script).
- 3 If you have the AWS SDK in the list of dependencies in package.json, you can remove it because you won't need it for this function.

You now need to add an npm module called jsonwebtoken. This module will help to verify the integrity of the token and decode it.

In a terminal window change to the directory of the function and run

```
npm install jsonwebtoken --save
```

Also, to make a request to Auth0 to retrieve user information, you're going to use a library called request. Install request by running `npm install request --save` from the terminal. Your package.json should look similar to the next listing.

Listing 5.7 Package.json for the user-profile Lambda function

```
{
  "name": "user-profile",
  "version": "1.0.0",
  "description": "This Lambda function returns the current user-profile",
  "main": "index.js",
  "scripts": {
    "deploy": "aws lambda update-function-code
    --function-name user-profile --zip-file fileb://Lambda-Deployment.zip",
    "predeploy": "zip -r Lambda-Deployment.zip * -x *.zip *.json *.log"
  },
  "dependencies": {
    "jsonwebtoken": "^5.7.0",
    "request": "^2.69.0"
  },
  "author": "Peter Sbarski",
  "license": "BSD-2-Clause",
}
```

← You can delete unused scripts (like the test script) and dependencies if they aren't needed in this function.

Your version numbers may be different.

Open index.js and replace its contents with code in the next listing. This code is responsible for validating and decoding the token. If it succeeds, it sends a request to the tokeninfo endpoint provided by Auth0. The JWT is included in the body of the request to Auth0. The tokeninfo endpoint returns information about the user, which is then sent back to the website.

Listing 5.8 Contents of the user-profile Lambda function

```
'use strict';

var jwt = require('jsonwebtoken');
var request = require('request');

exports.handler = function(event, context, callback){
```

```

    if (!event.authToken) {
        callback('Could not find authToken');
        return;
    }

    var token = event.authToken.split(' ')[1];

    var secretBuffer =
    new Buffer(process.env.AUTH0_SECRET);
    jwt.verify(token, secretBuffer, function(err, decoded) {
        if(err) {
            console.log('Failed jwt verification: ', err,
            'auth: ', event.authToken);
            callback('Authorization Failed');
        } else {
            var body = {
                'id_token': token
            };

            var options = {
                url: 'https://' + process.env.DOMAIN + '/tokeninfo',
                method: 'POST',
                json: true,
                body: body
            };

            request(options, function(error, response, body) {
                if (!error && response.statusCode === 200) {
                    callback(null, body);
                } else {
                    callback(error);
                }
            });
        }
    });
};

```

AUTH0_SECRET and DOMAIN are Lambda's environment variables. You can set and modify these in Lambda's console.

event.authToken needs to be split because it contains the word *Bearer* before the token.

The jsonwebtoken module can verify and decode at the same time. It's a useful utility if you need to check the integrity of a token and extract claims.

The request module is an excellent utility for performing all kinds of requests. If the error object is not null, you can assume that the request succeeded and send back its body via the API Gateway.

Environment variables

Environment variables are Lambda's way of storing configuration settings, database connections strings, and other useful information without having to embed them in a function. Saving settings in environment variables is highly recommended because it allows developers to update those settings without having to redeploy the function. Environment variables can be changed independently and in isolation from the function. The AWS platform makes environment variables available to the function via `process.env` (for Node.js). Furthermore, environment variables can be encrypted via KMS, which provides a good way to store important secrets. Chapter 6 has more information on this useful feature.

Deploy the function to AWS by running `npm run deploy` from the terminal. Finally, you need to create two environment variables for your Lambda function to store the Auth0

domain and the Auth0 secret (figure 5.14). Listing 5.8 uses these two variables to verify the token and issue a request to Auth0. To add these two variables, do the following:

- 1 Open Lambda in the AWS console and click the `user-profile` function.
- 2 At the bottom of the Code tab, you should see a section for environment variables.
- 3 Add a variable called `DOMAIN` followed by the Auth0 domain.
- 4 Add another variable called `AUTH0_SECRET` followed by the Auth0 secret. The domain and the secret can be copied from Auth0 (figure 5.6). It's easy to mix up the Auth0 client ID and secret, so double-check that you've copied the right value.
- 5 Click the Save button at the top to persist your settings.

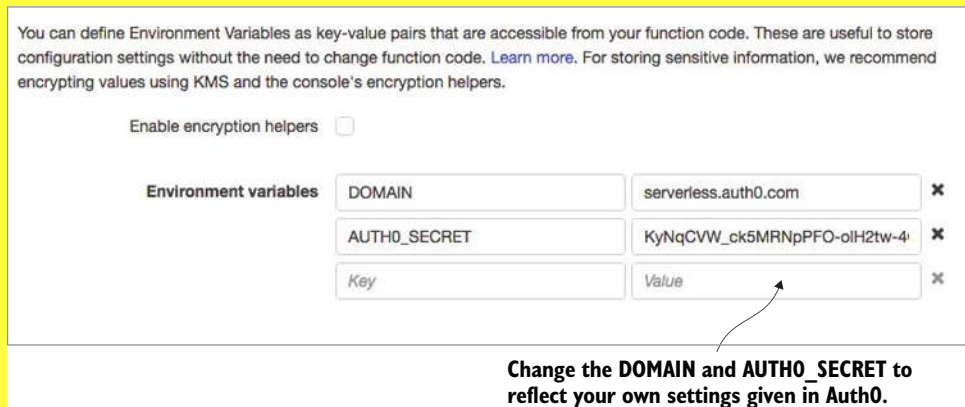


Figure 5.14 The `DOMAIN` and `AUTH0_SECRET` must be set for the Lambda function to run correctly.

5.3.2 API Gateway

You need to set up an API Gateway to accept requests from your website and invoke the `user-profile` Lambda function. You also need to create a resource, add support for a GET method, and enable cross-origin resource sharing (CORS):

- 1 In the AWS console, click API Gateway.
- 2 Type in a name for your API, such as `24-hour-video` and, optionally, a description.
- 3 Click Create API to create your first API.

APIs in the Gateway are built around resources. Every resource can be combined with an HTTP method such as HEAD, GET, POST, PUT, OPTIONS, PATCH, or DELETE. You're going to create a resource called `user-profile` and combine it with a GET method. In the API you just created, follow these steps:

- 1 Click Actions and select Create Resource.
- 2 Type User Profile in the Resource Name field. The Resource Path field should automatically fill in (figure 5.15).

New Child Resource

Use this page to create a new child resource for your resource.

Configure as [proxy resource](#) ☐ ⓘ

Resource Name*

Resource Path*

You can add path parameters using brackets. For example, the resource path {username} represents a path parameter called 'username'. Configuring /(proxy+) as a proxy resource catches all requests to its sub-resources. For example, it works for a GET request to /foo. To handle requests to /, add a new ANY method on the / resource.

Enable API Gateway CORS ☐ ⓘ

* Required

[Cancel](#) [Create Resource](#)

We will have to enable CORS but don't do it right now. You will do it once you've created a GET method for this resource.

Figure 5.15 Creating a resource takes a few seconds in the API Gateway.

- 3 Click the Create Resource button to create and save the resource.
- 4 The left list should now show the /user-profile resource.
- 5 Make sure the resource is selected, and click Actions again.
- 6 Click Create Method to create a new GET method.
- 7 Under the /user-profile resource, click the drop-down and select GET (figure 5.16).
- 8 Click the check mark button to save.

Resources **Actions** **/user-profile Methods**

▼ /

/user-profile ✓

- ANY
- DELETE
- GET
- HEAD
- OPTIONS
- PATCH
- POST
- PUT

You have a choice of different methods for your resource. For the user-profile Lambda function, select GET.

Figure 5.16 Select the GET method for the /user-profile resource. You'll use it to retrieve information about the user.

/user-profile - GET - Setup

Choose the integration point for your new method.

Integration type ☒ Lambda Function ⓘ

☐ HTTP ⓘ

☐ Mock ⓘ

☐ AWS Service ⓘ

Use Lambda Proxy integration ☐ ⓘ

Lambda Region us-east-1 ▾

Lambda Function user-profile ⓘ

Save

You can specify a Lambda function version or an alias.

Figure 5.17 You need to set up this integration request before CORS can be enabled.

Having saved the GET method, you should immediately see the Integration Request screen (figure 5.17):

- 1 Click the Lambda Function radio button.
- 2 Select your region (for example, us-east-1) from the Lambda Region drop-down menu.
- 3 Type user-profile in the Lambda Function text box.
- 4 Click Save.
- 5 Click OK if you're asked if it's okay to add permissions to the Lambda function.

Next, you need to enable CORS:

- 1 Click the /user-profile resource.
- 2 Click Actions.
- 3 Select Enable CORS.
- 4 The CORS configuration screen can be left with the defaults. The Access-Control-Allow-Origin field is set to a wildcard, which means that any other domain/origin can send a request to your endpoint. This is fine for now, but you'll restrict it down the road, especially as you get ready to roll out staging and production environments (figure 5.18).
- 5 Click Enable CORS and Replace Existing CORS Headers to save the configuration.
- 6 Click Yes, and replace existing values in the confirmation box that pops up.

Enable CORS

Cross-Origin Resource Sharing (CORS) allows browsers to make HTTP requests to servers with a different domain/origin. Specify which methods in the **/user-profile** resource are available to CORS requests. To define static values surround the value in single quotes (eg. 'amazon.com'). To define mappings use the syntax described in the Method Editor (eg. `method.request.querystring.myQueryString`).

Methods* ☒ GET ☒ OPTIONS ⓘ

Access-Control-Allow-Methods GET,OPTIONS ⓘ

Access-Control-Allow-Headers 'Content-Type,X-Amz-Date,Authorization' ⓘ

Access-Control-Allow-Origin* '*' ⓘ ⚠

▶ Advanced

Enable CORS and replace existing CORS headers

Leaving this header set to * will make the resource accessible from any origin.

Figure 5.18 CORS will enable you to access this API from your website.

5.3.3 Mappings

If you look at listing 5.8, you'll see code that refers to `event.authToken`. This is the JWT token passed in via the Authorization header from the website. To make this token available in a Lambda function, you need to create a mapping in the API Gateway.

Mapping templates

In listing 5.9 you're creating a mapping using the Velocity Template Language (VTL). This mapping extracts a value from the HTTP (method) request and makes it available to your Lambda function (via a property called `authToken` on the event object). A mapping template transforms data from one format to another. See chapter 7 for more information on mapping templates.

This mapping will extract the Authorization header and add it as an `authToken` to the event object:

- 1 Click the GET method under the `/user-profile` resource.
- 2 Click Integration Request.
- 3 Expand Body Mapping Templates.
- 4 Click Add Mapping Template.
- 5 Type in `application/json` and click the check mark button.

- 6 Select Yes, Secure This Integration if you see a dialog box titled Change Passthrough Behavior.
- 7 In the Template box type in the code in the next listing.
- 8 Click Save once you're finished (figure 5.19).

Listing 5.9 Mapping template for the token

```
{  
  "authToken" : "$input.params('Authorization')"  
}
```

Mapping takes elements out of a request and makes them available as properties on the event object.

Body Mapping Templates

Request body passthrough ☐ When no template matches the request Content-Type header ⓘ ☒ When there are no templates defined (recommended) ⓘ ☐ Never ⓘ

Content-Type

application/json

+ Add mapping template

application/json

Generate template:

```
1 {  
2   "authToken" : "$input.params('Authorization')"  
3 }
```

Cancel Save

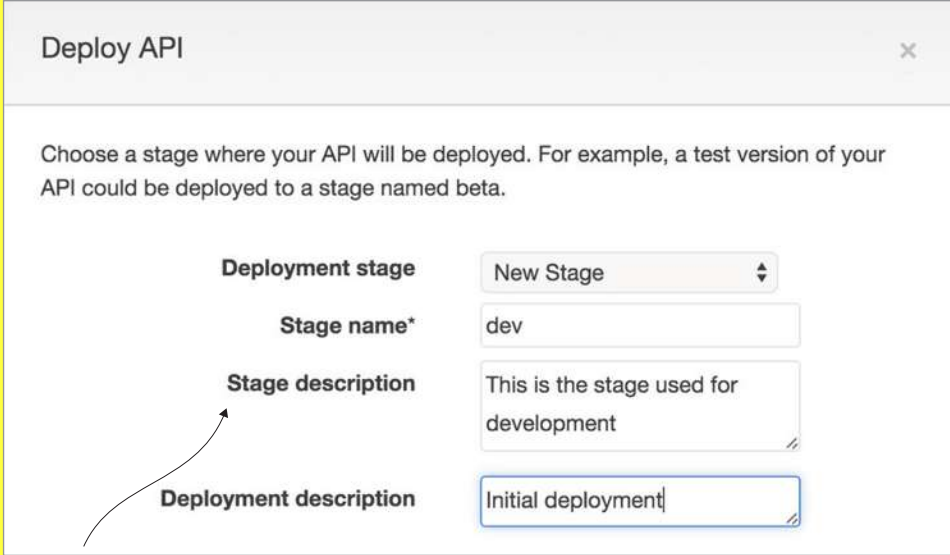
Figure 5.19 A mapping template can transform elements of a request to properties accessible via the event object in a Lambda function.

Lambda proxy integration

In figure 5.17, you might have noticed a check box labeled Use Lambda Proxy Integration. If you had enabled that check box, the incoming HTTP request—including all headers, query string parameters, and the body—would have been mapped and made available to the function via the event object automatically. This means that you wouldn't have had to create a mapping template as you did in listing 5.9 (the file-name would be accessible from `queryStringParameters` on the event object). The reason you didn't do this is because we wanted to show you how to create a custom mapping template and extract only the parameter you need (rather than passing the entire HTTP request to the function). In many cases, proxy integration is very useful and you'll certainly use it as you progress through the chapters. See chapter 7 for a more in-depth discussion on proxy integration versus manual mapping.

Finally, you need to deploy the API and get a URL to invoke from the website:

- 1 In the API Gateway, make sure your API is selected.
- 2 Click Actions.
- 3 Select Deploy API.
- 4 In the pop-up, select [New Stage].
- 5 Type `dev` as the Stage Name.
- 6 Click Deploy to provision the API (figure 5.20).



Deploy API [Close]

Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta.

Deployment stage	New Stage
Stage name*	dev
Stage description	This is the stage used for development
Deployment description	Initial deployment

Create different stages such as dev, test, and production for your API.

Figure 5.20 Every time you make changes to your API, remember to deploy them using the Deploy API button. You can deploy to an existing stage or create a new one.

This URL is needed to send requests to the API Gateway.

Invoke URL: <https://tizo7a7o9.execute-api.us-east-1.amazonaws.com/dev>

Settings Stage Variables SDK Generation Export Deployment History Documentation History

Configure the metering and caching settings for the **dev** stage.

Cache Settings

Enable API cache ☐

CloudWatch Settings

Enable CloudWatch Logs ☐ ⓘ

Enable Detailed CloudWatch Metrics ☐ ⓘ

Default Method Throttling

Choose the default throttling level for the methods in this stage. Each method in this stage will respect these rate and burst settings. Your current account level throttling rate is 1000 requests per second with a burst of 2000 requests. ⓘ

Enable throttling ☒ ⓘ

Rate requests per second

Burst requests

Client Certificate

Select the client certificate that API Gateway will use to call your integration endpoints in this stage.

Certificate ⌵

Save Changes

Figure 5.21 You can use the stage settings page to adjust other settings. We'll cover these in more detail in chapter 7.

The next page you see will show the Invoke URL and a number of options (figure 5.21). Copy the URL, because you'll need it for the User Profile button.

5.3.4 Invoking Lambda via API Gateway

The final two steps are to update the Show Profile click handler and `config.js` to invoke the Show Profile Lambda function via the API Gateway. Open `user-controller.js` in the `js` folder of the 24-Hour Video website, and add the code shown in the next listing (right after the logout click-handler definition).

Listing 5.10 The Show Profile click event handler

```
this.uiElements.profileButton.click(function (e) {
  var url = that.data.config.apiUrl + '/user-profile';
  $.get(url, function (data, status) {
```

```
        alert(JSON.stringify(data));
    })
  });
```

← The retrieved response from the API Gateway must be stringified to be displayed in an alert.

Finally, update the contents of `config.js` to match the next listing. Once you’ve done that, you can test the entire system.

Listing 5.11 Updated `config.js`

```
var configConstants = {
  auth0: {
    domain: 'AUTH0-DOMAIN',
    clientId: 'AUTH0-CLIENTID'
  },
  apiBaseUrl: 'https://API-GATEWAY-URL/dev'
};
```

Update the domain and client ID to match your Auth0 settings (figure 5.7).

← Update the `apiBaseUrl` to match the URL given in the API Gateway.

Check that the 24-Hour Video website is running. If it isn’t, run `npm start` from the terminal (make sure you’re in the website’s directory) and sign in via Auth0. Click the User Profile button. You should see an alert with the contents of the user’s profile in Auth0.

5.3.5 Custom authorizer

API Gateway supports custom request authorizers. These are Lambda functions that the API Gateway can use to authorize requests. A custom authorizer runs at the method request stage—that is, before the request reaches the target back end. A custom authorizer can validate a bearer token and return a valid IAM policy, which authorizes the request. If the returned policy is invalid, the request is not allowed to continue. To prevent constant invocations of custom authorizers, policies along with the incoming token are cached for an hour.

The benefit of using a custom authorizer is that you can write a dedicated Lambda function to validate the JWT (instead of doing it in every function you want to invoke). Figure 5.22 shows what a modified request flow looks like when a custom authorizer is introduced.

You’re going to implement a custom authorizer now to see how it works. There are three steps:

- 1 Create a new Lambda function in AWS.
- 2 Write a custom authorizer function and deploy it.
- 3 Change the method request settings in the API Gateway to use a custom authorizer.

The first step is to create a regular Lambda function just as you did before:

- 1 Create a function in Lambda’s console.
- 2 Name this function `custom-authorizer`.

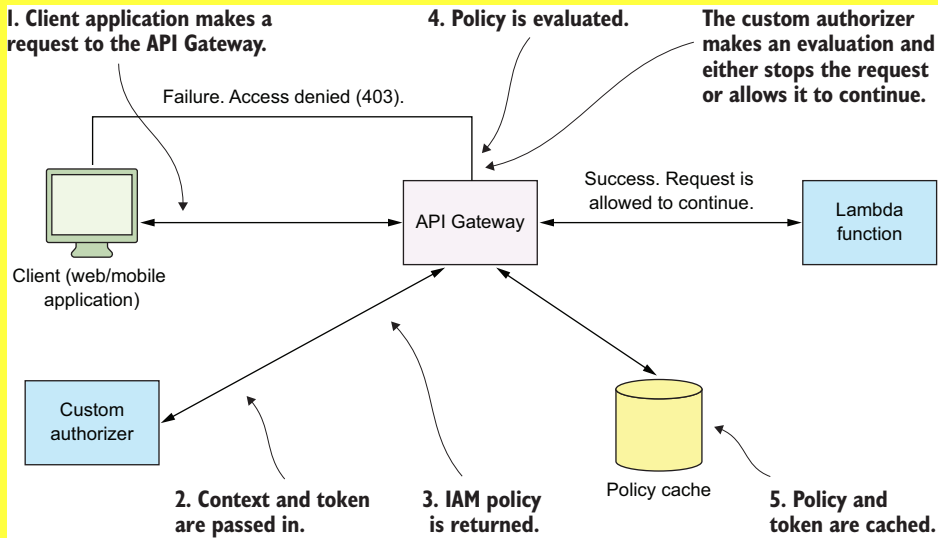


Figure 5.22 The custom authorizer is useful as a means of validating JWT for all Lambda functions that are supposed to be secured.

- 3 Assign the `api-gateway-lambda-exec-role` to it and save.
- 4 Make a copy of the `user-profile` Lambda function on your computer and rename it to `custom-authorizer`.
- 5 Update the function name or the ARN in the deploy script in `package.json`.
- 6 Open `index.js` and replace it with the code in listing 5.12 (this function is referenced from Amazon's documentation at <http://amzn.to/24Dli80>). As you can see, the code for this Lambda function is similar to that of the `user-profile` function. The main difference is a new function called `generatePolicy`, which returns an IAM policy that allows execution to continue.

Listing 5.12 Custom authorizer

```
'use strict';

var jwt = require('jsonwebtoken');

var generatePolicy = function(principalId, effect, resource) {
  var authResponse = {};
  authResponse.principalId = principalId;
  if (effect && resource) {
    var policyDocument = {};
    policyDocument.Version = '2012-10-17';
    policyDocument.Statement = [];
    var statementOne = {};
    statementOne.Action = 'execute-api:Invoke';
    statementOne.Effect = effect;
    statementOne.Resource = resource;
    policyDocument.Statement[0] = statementOne;
  }
  return authResponse;
};
```

The policy stipulates that the API Gateway is allowed to invoke the required resource.

```

        authResponse.policyDocument = policyDocument;
    }
    return authResponse;
}

exports.handler = function(event, context, callback){
    if (!event.authorizationToken) {
        callback('Could not find authToken');
        return;
    }

    var token = event.authorizationToken.split(' ')[1];

    var secretBuffer = new Buffer(process.env.AUTH0_SECRET);
    jwt.verify(token, secretBuffer, function(err, decoded){
        if(err){
            console.log('Failed jwt verification: ', err,
                ➡ 'auth: ', event.authorizationToken);

            callback('Authorization Failed');
        } else {
            callback(null,
                ➡ generatePolicy('user', 'allow', event.methodArn));
        }
    })
};

```

The auth0 secret is accessed through an environment variable that you can set in Lambda's console.

If the token is validated, the function returns a user policy that allows the invocation of the API.

Deploy the function to AWS once you've implemented it. You also need to add the AUTH0_SECRET as an environment variable to the function:

- 1 In the AWS console, choose Lambda and then choose the custom-authorizer function.
- 2 In the Code tab, find the Environment Variable section.
- 3 Add AUTH0_SECRET as the key and your Auth0 secret as the value.
- 4 Click Save at the top of the page to persist your settings.

The final steps are to create a custom authorizer in the API Gateway and connect it to the GET method you created previously:

- 1 In the API Gateway, choose the 24-Hour Video API.
- 2 Select Authorizers from the menu on the left.
- 3 You should see a New Custom Authorizer form on the right side. If you don't see it, click the Create drop-down and choose Custom authorizer.
- 4 Fill out the custom authorizer form (figure 5.23).
 - Select your Lambda region (us-east-1).
 - Set the Lambda function name, which is custom-authorizer.
 - Set a name for your authorizer. It can be anything you want, like custom-authorizer or authorization-check.
 - Make sure that the Identity token source is set to `method.request.header.Authorization`.

New Custom Authorizer

Provide a name, Lambda function, and identity token source for your authorizer.

Lambda region*	us-east-1	
Lambda function*	custom-authorizer	<small>?</small>
Authorizer name*	custom-authorizer	
Execution role	arn:aws:iam::myAccount:role/myRole	<small>?</small>
Identity token source*	method.request.header.Authorization	<small>?</small>
Token validation expression		<small>?</small>
Result TTL in seconds*	300	<small>?</small>

* Required

[Cancel](#) [Create](#)

The role that the API Gateway can use to make requests to the custom authorizer.

API Gateway can attempt to validate the token using a regular expression before the Lambda function is invoked.

Figure 5.23 You can use the custom authorizer to implement various authorization strategies. You can create multiple authorizers and connect them to methods in the API Gateway.

- 5 Click Create to create the custom authorizer.
- 6 Confirm that you want to allow API Gateway to invoke the custom-authorizer function.

Now you can set your custom authorizer to invoke automatically whenever a GET request to /user-profile is issued:

- 1 In the API Gateway, click Resources under 24-hour-video (the sidebar on the left).
- 2 Click GET under /user-profile.
- 3 Click Method Request
- 4 Click the pencil button next to Authorization.
- 5 From the drop-down select your custom authorizer and save (figure 5.24).
- 6 Deploy the API again:
 - Click Actions.
 - Choose Deploy API.
 - Select dev as the Deployment Stage.
 - Choose Deploy.

To test the custom authorizer, make the User Profile button display when the user isn't logged in. To do that, open main.css and remove the style for #user-profile from it. Also, delete your JWT from local storage and refresh the site. Click the User Profile button. Your custom authorizer should reject the request. You can use this custom authorizer for all Lambda functions down the road.

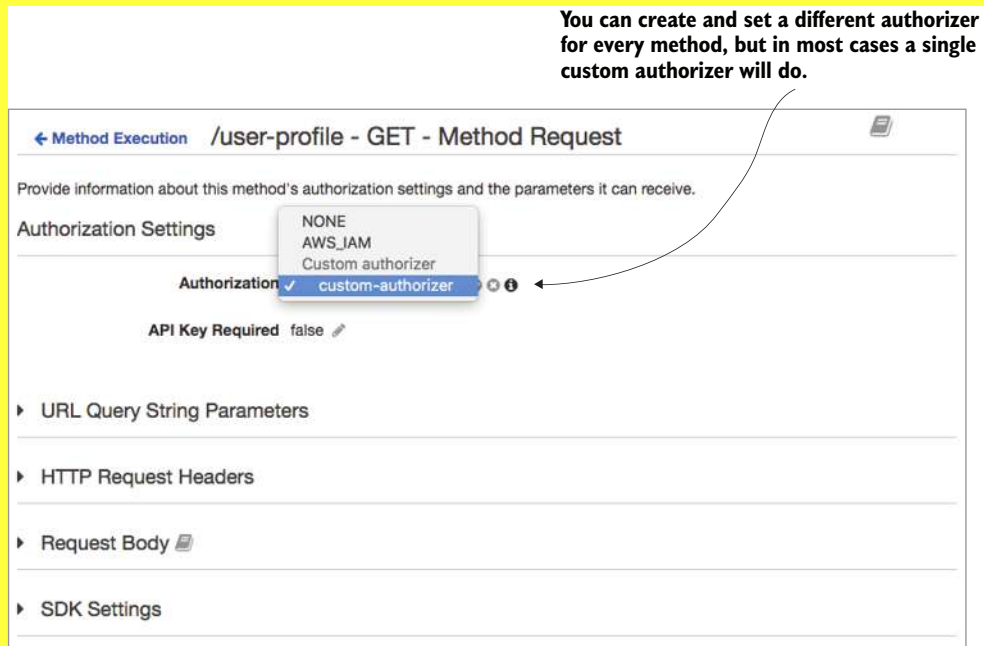


Figure 5.24 Custom authorizers are a good way to authorize requests coming via the API Gateway.

401 Unauthorized

If you've successfully signed in to the 24-Hour Video website and then refreshed after a long period of time, you might see an error message that says, "There was an error getting the profile: 401: Unauthorized." This could be because the JWT cached in your browser has expired. Sign on to your website again and everything should work again (the message will no longer appear). The default JWT expiration is 36,000 seconds (10 hours), but you can override it in Auth0 or you can choose to implement refresh tokens if you're up for a challenge (<http://bit.ly/2jxbjPg>).

5.4 Delegation tokens

Delegation tokens are designed to make integration between services easier. So far, you've taken a JSON Web Token supplied by Auth0 and sent it across to AWS where it was verified and decoded by a Lambda function. You had to write a little bit of code to do that. Delegation tokens are created for specific services that know how to decode these tokens and extract claims or information. In effect, delegation tokens are tokens created by one service to call another service or API.

5.4.1 *Real-world examples*

Firebase is a real-time streaming database that we'll look at in chapter 9. It supports delegation tokens. If a request from a client comes with a delegation token, Firebase knows how to verify it without you having to do anything (or write any code).

To add support for a Firebase delegation token, you need to generate a secret key in Firebase and add it to Auth0. Then your website can request a delegation token from Auth0, which is signed by the secret key from Firebase. Any subsequent request made to Firebase can be sent with the delegation token, which Firebase knows how to decrypt (because it provided the secret key in the first place). In chapter 9, we'll show you how to provision a delegation token for Firebase in more detail. Similarly, you can set up Auth0 to enable delegated authentication with AWS by setting up a SAML provider and configuring one or more roles.

5.4.2 *Provisioning delegation tokens*

When it comes to Auth0, to get a delegation token you need to configure an add-on for the service you wish to use and then request the token via the `/delegation` endpoint. If you wish to integrate with a service such as Firebase or use a delegation token with AWS, you'll need to enable the appropriate add-on in Auth0 (figure 5.25).

Every add-on has different configuration requirements, so you'll need to consult relevant Auth0 documentation to find out what's needed. To set up delegated authentication between Auth0 and AWS, refer to <https://auth0.com/docs/aws-api-setup>. Another great example is described in <https://auth0.com/docs/integrations/aws-api-gateway>.

5.5 *Exercises*

Try to do the following exercises to confirm your understanding of concepts presented in this chapter:

- 1 Create a Lambda function (`user-profile-update`) for updating a user's personal profile. Assume that you can access the first name, last name, email address, and `userId` on the event object. Because you don't have a database yet, this function doesn't need to persist this information, but you can log it to CloudWatch.
- 2 Create a POST method for the `/user-profile` resource in the API Gateway. This method should invoke the `user-profile-update` function and pass in the user's information. It should use the custom authorizer developed in section 5.3.5.
- 3 Create a page in the 24-Hour Video website to allow signed-in users to update their first name, last name, and email address. This information should be submitted to the `user-profile-update` function via the API Gateway.
- 4 In listing 5.3, you set the token to be included in every request using `$.ajaxSetup`. If your website makes a request to an external party, your token might be stolen. Think of a way to make the system more secure by including the token only when the website issues requests against the API Gateway.

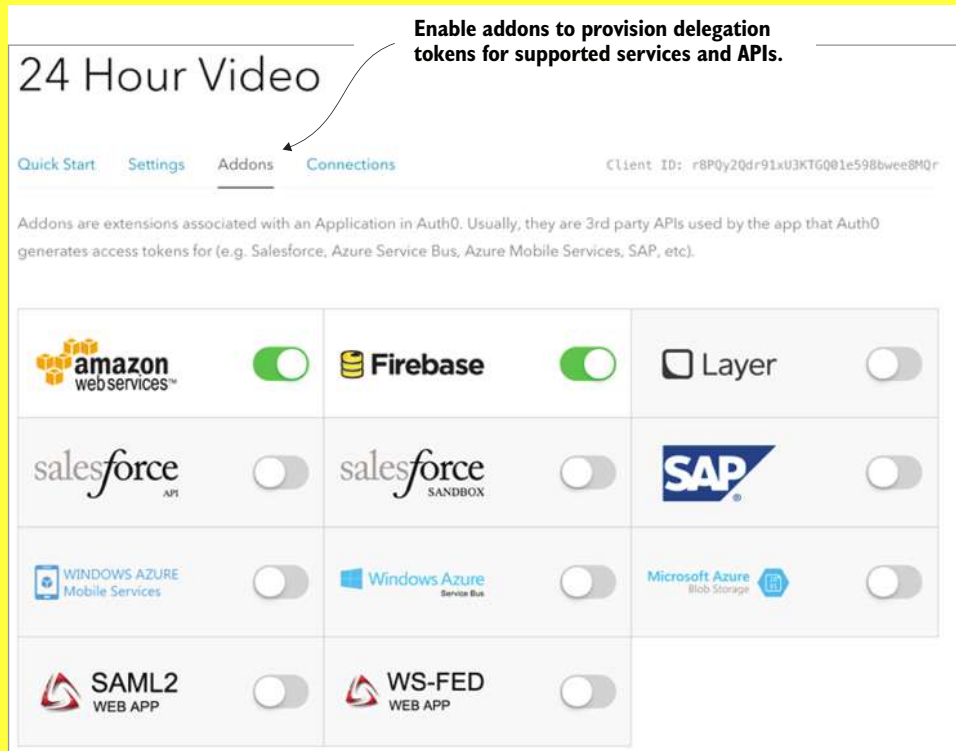


Figure 5.25 Use delegation tokens to reduce the need for more Lambda functions.

- 5 Modify the user-profile Lambda function to no longer validate the JSON Web Token. This validation isn't needed because of the custom authorizer. The function should still request user information from the Auth0 tokeninfo endpoint.
- 6 Add an additional social identity provider to your Auth0 app such as Yahoo, LinkedIn, or Windows Live.
- 7 The Auth0 JWT token stored in the browser's local storage will expire after a certain time. This may result in an error message shown to the user when the website is refreshed. Figure out a way to suppress the error message and automatically delete expired tokens.

5.6 Summary

In this chapter, we looked at how to enable authentication and authorization in a serverless application. We looked at how services can communicate directly with the client and checked out JSON Web Tokens. We also introduced Auth0, a service that takes care of many authentication and authorization concerns, and we discussed how delegation tokens can be used across different services. Finally, we stepped through an example where you did the following:

- Developed a website for 24-Hour Video
- Created an Auth0 app and added sign-in/out functionality to the website
- Developed a Lambda function to return user profile information
- Implemented an API Gateway and created a custom authorizer that decodes JWT

In the next chapter, we'll look at Lambda functions in much more detail. We'll consider advanced use cases, see how to use patterns to help implement concise functions without a large number of callbacks, and discuss ways to improve the performance of Lambda-based systems.