You may also consider looking at the OpenID Connect website for a list of certified implementations. The list is available at `https://openid.net/developers/certified/`.

In the next section, we are going to look at how to integrate Keycloak using different technology stacks.

> **Note**
>
> The code examples provided in this chapter are not targeted at being run in production; instead, they demonstrate how to integrate Keycloak with different types of applications.

Integrating with Golang applications

Go applications can integrate with Keycloak using whatever library you prefer, as long as it complies with the OpenID Connect or OAuth2 specifications.

For the sake of simplicity and to provide a generic example of how to integrate with Keycloak, we are going to use the `https://github.com/coreos/go-oidc` package. The code examples for this section are available in the following directory:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/golang
```

In the preceding directory, you will find a `main.go` file that contains all the code you will need to follow and run the examples.

In the next section, we are going to start looking at how to enable a web application that can authenticate users using Keycloak.

Configuring a Golang client

First, you need to create a provider using a base URL that the OpenID Connect Discovery Document will be fetched from. This will set up the necessary endpoints that your application will be talking to when you're authenticating users and obtaining tokens from Keycloak. The following code creates a new provider using the discovery document available at `http://localhost:8180/realms/myrealm/.well-known/openid-configuration`:

```
func createOidcProvider(ctx context.Context) *oidc.Provider {
     provider, err := oidc.NewProvider(ctx, "http://
localhost:8180/realms/myrealm")
```

```
      if err != nil {
            log.Fatal("Failed to fetch discovery document: ",
err)
      }

      return provider
}
```

You also need to provide information about the client in Keycloak that the application will use to access Keycloak. For that, we need to set the client ID and secret like so:

```
func createConfig(provider oidc.Provider) (oidc.Config, oauth2.
Config) {
      oidcConfig := &oidc.Config{
            ClientID: "mywebapp",
      }

      config := oauth2.Config{
            ClientID:    oidcConfig.ClientID,
            ClientSecret: CLIENT_SECRET,
            Endpoint:    provider.Endpoint(),
            RedirectURL: "http://localhost:8080/auth/callback",
            Scopes:      []string{oidc.ScopeOpenID, "profile",
"email"},
      }

      return *oidcConfig, config
}
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the preceding code with the secret generated by Keycloak for the `mywebapp` client. For that, go the `mywebapp` client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

The next step is to change your application so that it redirects users to Keycloak when they try to access the application. In the following code, we have set a cookie to track the value of the state parameter and redirect users to Keycloak:

```
func redirectHandler(w http.ResponseWriter, r *http.Request) {
state := addStateCookie(w)
     http.Redirect(w, r, oauth2Config.AuthCodeURL(state), http.
StatusFound)
}
```

Note, however, that, as a developer, you are responsible for implementing how users are redirected to Keycloak using the `oauth2Config.AuthCodeuRL` function. We are also implementing the necessary logic to generate the `state` parameter, as well as how we can store it as an HTTP cookie, so that we can associate the original authorization request with a response from Keycloak once the user is authenticated. Depending on the library you are using, you do not need to perform any of these steps in your code as they might be performed transparently by the library.

Finally, let's implement the callback URL that Keycloak is going to redirect users to right after a successful authentication attempt:

```
func callbackHandler(resp http.ResponseWriter, req *http.
Request) {
     err := checkStateAndExpireCookie(req, resp)

     if err != nil {
          http.Error(resp, err.Error(), http.StatusBadRequest)
          return
     }
     tokenResponse, err := exchangeCode(req)
     if err != nil {
          http.Error(resp, "Failed to exchange code", http.
StatusBadRequest)
          return
     }
     idToken, err := validateIDToken(tokenResponse, req)
     if err != nil {
          http.Error(resp, "Failed to validate id_token", http.
StatusUnauthorized)
          return
```

```
        }
        handleSuccessfulAuthentication(tokenResponse, *idToken,
resp)
}
```

The callback handler is responsible for the following:

- Checking whether the state is the same as the one that was originally sent when you performed the authorization request to Keycloak.
- Exchanging the code that was returned from Keycloak to obtain the ID token, access token, and refresh token.
- Verifying the ID token's validity in terms of signature, audience, and expiration date.

> **Note**
>
> A proper integration would avoid reusing the same `state` value, as well as managing local sessions once the user has been authenticated for the very first time. In fact, clients should prefer using **Proof Key for Code Exchange** (**PKCE**) to prevent **Cross-Site Request Forgery** (**CSRF**) and code replay attacks. We are using the state due to the lack of PKCE support from the `go-oidc` package. To use PKCE, you will need to implement it by yourself or use some third-party package.

Let's start the application by running the following command at the root directory of your project:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/golang
$ go run main.go
```

Your application should start and be available at `http://localhost:8080`. Now, try to access that URL and log into Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing these user credentials, you should be redirected back to the application, now as an authenticated user, and a page will appear that contains the tokens that have been issued by the server.

In this section, you learned about the basics of to how integrate a GoLang application with Keycloak. The `go-oidc` package is a well-known package that provides OpenID Connect capabilities for client applications. It provides a good baseline for integrating with Keycloak and allows you to enable authentication for your application. However, it requires additional work from the developer to get it done right, as well as to maintain the code.

If you are using a framework, such as Gin, or if you know about any other package that do not require you to understand the integration internals while providing a rich set of features and configuration, you should use that instead.

There are also quite a few third-party libraries you can find that are targeted at integrating with Keycloak. Unfortunately, we cannot recommend any of them since they are not in conformance with some of the recommendations that were mentioned at the beginning of this chapter – mainly the fact they are not backed by a strong community but by individuals.

In the next section, we are going to look at more integration options that use other programming languages.

Integrating with Java applications

Frameworks, web containers, and application servers that provide support for OpenID Connect and OAuth2 as part of their offerings should make your life a lot easier, since the integration is already available to your application and there is no need to add any other dependencies.

Leveraging what is already in your technology stack is usually the best choice. But that will not always be the case.

Keycloak also provides client-side implementations that have support for some of the most common frameworks, web containers, and application servers available. Also known as Keycloak adapters, these implementations are targeted at people looking for a deeper integration with Keycloak.

In the next few sections, we will look at the different options you can choose from so that you can pick the one that works best for your applications.

Using Quarkus

Quarkus provides an OpenID Connect compliant extension called `quarkus-oidc`. It provides a simple and rich configuration model that can protect both frontend and backend applications. Quarkus has built-in support for the most common **Integrated Development Environments** (**IDEs**), such as IntelliJ and Eclipse, and you should be able to quickly create or configure an existing project in order to integrate it with Keycloak.

> **Tip**
>
> If you are new to Quarkus or just want to protect your applications using OpenID Connect and Keycloak, please look at the guides available at `https://quarkus.io/guides`. Most of these guides and code examples use Keycloak as an OpenID Provider and will help you quickly get started. Search for guides using `OpenID Connect` as a keyword to filter all the available guides related to integrating with Keycloak.

In summary, the `quarkus-oidc` extension allows you to protect two main types of applications: `web-app` and `service`.

The `web-app` type represents applications that authenticate using Keycloak through the browser, using the authorization code grant type. These are frontend applications.

On the other hand, the `service` type represents applications that rely on bearer tokens issued by a Keycloak server to authorize access to their protected resources. These are backend applications.

To use the `quarkus-oidc` extension in your project, add the following dependency to the application's `pom.xml` file:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-oidc</artifactId>
</dependency>
```

Now that we've added the `quarkus-oidc` dependency, it is time to decide on the type of application.

The code examples for this section are available in this book's GitHub repository at the following link:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/quarkus
```

In the preceding directory, you will find a `frontend` directory and a `backend` directory, both of which contain all the code you will need to follow and run the upcoming examples.

In the next section, we are going to start looking at how to enable a web application so that we can authenticate users using Keycloak.

Creating a Quarkus client

In this section, we will be looking at the code examples that are available from the following directory:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/quarkus/frontend
```

Let's start by configuring a `web-app` application by adding the following properties to the `src/main/resources/application.properties` file:

```
quarkus.oidc.auth-server-url=http://localhost:8180/auth/realms/
myrealm
quarkus.oidc.client-id=mywebapp
quarkus.oidc.client-secret=CLIENT_SECRET
quarkus.oidc.application-type=web-app
quarkus.http.auth.permission.authenticated.paths=/*
quarkus.http.auth.permission.authenticated.policy=authenticated
```

From a configuration perspective, the main configuration options are as follows:

- The `quarkus.oidc.auth-server-url` property defines the URL that the application should fetch the OpenID Connect Discovery document from.
- The `quarkus.oidc.client-id` property maps a client in Keycloak with this application. For this application, we are going to use the `mywebapp` client, which we created at the beginning of this chapter.
- The `quarkus.oidc.client-secret` property is the secret that was generated by Keycloak when the client was created.
- The `quarkus.oidc.application-type` property defines that this application is a web application.
- The `quarkus.http.auth.permission.authenticated.paths` and `quarkus.http.auth.permission.authenticated.policy` properties define that all the paths in the applications require an authenticated user.

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the `src/main/resources/application.properties` file with the secret that was generated by Keycloak for the **mywebapp** client. For that, go to the **mywebapp** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

Let's start the application by running the following command at the root directory of your project:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/quarkus/frontend
$ ./mvnw quarkus:dev
```

Your application should start and be available at `http://localhost:8080`. Try to access that URL and log into Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the necessary user credentials, you should be redirected back to the application, now as an authenticated user.

> **Note**
>
> By default, Quarkus is going to set a cookie that will expire based on the expiration time of the token issued by Keycloak. If you are experiencing the user not redirected to Keycloak to authenticate, you might want to clear your browser cookies. This behavior is something you can configure. For more details, look at the **quarkus-oidc** extension documentation.

In this section, you learned about how to configure a web application in order to authenticate users using Keycloak. At this point, you should be able to create your own application or configure an existing one to authenticate users using Keycloak.

In the next section, we will look at how to configure a backend application to authorize access to resources based on tokens issued by Keycloak.

Creating a Quarkus resource server

The code examples that will be presented in this section are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/quarkus/backend
```

For backend applications that have been protected using a OAuth2 Bearer Token, the configuration is similar to configuring frontend applications, except for changing `quarkus.oidc.application-type` to `service`, as well as `quarkus.oidc.client-id` so that it maps to a different client in Keycloak:

```
quarkus.oidc.auth-server-url=http://localhost:8180/auth/realms/
myrealm
quarkus.oidc.client-id=mybackend
quarkus.oidc.credentials.secret=CLIENT_SECRET
quarkus.oidc.application-type=service
quarkus.http.auth.permission.authenticated.paths=/*
quarkus.http.auth.permission.authenticated.policy=authenticated
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the `src/main/resources/application.properties` file with the secret that was generated by Keycloak for the **mybackend** client. For that, go the **mybackend** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

The `quarkus.oidc.application-type` property, which is now set to `service`, indicates that this application should authorize access based on bearer tokens.

Let's start the application by running the following command at the root directory of your project:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/quarkus/backend
$ ./mvnw quarkus:dev
```

Your application should start and be available at `http://localhost:8080`. To access the resources at the running application, you will need an access token. To obtain one, use the following command:

```
$ export access_token=$(\
    curl -X POST http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/token \
    --user mybackend:CLIENT_SECRET\
    -H 'content-type: application/x-www-form-urlencoded' \
    -d 'username=alice&password=alice&grant_type=password' | jq
--raw-output '.access_token' \
 )
```

Once you have run this command, an access token will be saved in an `access_token` environment variable, and you can now access the application:

```
$ curl -X GET \
  http://localhost:8080/hello \
  -H "Authorization: Bearer "$access_token
```

As a result, you should expect the following output from that command:

```
$ Hello RESTEasy
```

Now, if you try to access the application without a `Bearer` token or use an invalid one, you should get a 401 status code, indicating that your request was forbidden:

```
$ curl -v -X GET \
  http://localhost:8080/hello
```

The `quarkus-oidc` extension validates tokens based on whether they represent a **JSON Web Token** (**JWT**) or not. If the token is a JWT, the extension will try to validate the token locally by checking its signatures, audience, and expiration date. Otherwise, if the token is opaque and the format is unknown, it will invoke the token's introspection endpoint at Keycloak to validate it. For Quarkus applications, the `quarkus-oidc` extension is the best option you have. It provides an amazingly simple configuration while providing a lot of other options you can use to customize its behavior.

We only covered the main steps of setting up the `quarkus-oidc` extension here so that you can authenticate your users through Keycloak. There is lot more you can do with this extension, such as leverage capabilities for logout, obtain information about the subject into your beans, multi-tenancy, and so on. For more details, please check out the extension's documentation at `https://quarkus.io/guides/security#openid-connect`.

In the next section, we will look at how to integrate with Spring Boot applications.

Using Spring Boot

Spring Boot applications can integrate with Keycloak by leveraging Spring Security's OAuth2/OpenID libraries. Spring Boot also has built-in support for the most common IDEs, such as IntelliJ and Eclipse, so you should be able to quickly create or configure an existing project so that it can be integrated with Keycloak.

There are two main libraries, where each is targeted by a specific type of application: clients and resource servers.

The code examples in this section are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/springboot
```

In the preceding directory, you will find a `frontend` directory and a `backend` directory containing all the code you will need to follow and run the examples.

In the next section, we are going to start looking at how to enable a web application so that we can authenticate users using Keycloak.

Creating a Spring Boot client

The code examples presented in this section are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/springboot/frontend
```

First, add the following dependencies to your project to enable OAuth2/Open ID Connect support:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

You should now change the `src/main/resources/application.yaml` file so that you can configure the application, as follows:

```
spring:
  security:
    oauth2:
      client:
        registration:
          myfrontend:
            provider: keycloak
            client-id: mywebapp
            client-secret: CLIENT_SECRET
```

```
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/"
            scope: openid
        provider:
          keycloak:
            authorization-uri: http://localhost:8180/auth/
realms/myrealm/protocol/openid-connect/auth
            token-uri: http://localhost:8180/auth/realms/
myrealm/protocol/openid-connect/token
            jwk-set-uri: http://localhost:8180/auth/realms/
myrealm/protocol/openid-connect/certs
```

> **Note**
> You should change the reference to `CLIENT_SECRET` in the preceding configuration with the secret generated by Keycloak for the **mywebapp** client. For that, go the **mywebapp** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

Let's start the application by running the following command at the root directory of your project:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/springboot/frontend
$ ./mvnw spring-boot:run
```

Your application should start and be available at `http://localhost:8080`. Try to access that URL and log into Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

In this section, you learned about how to configure a web application to authenticate users using Keycloak. With that, you should be able to create your own application or configure an existing one to authenticate users using Keycloak.

In the next section, we will look at how to configure a backend application to authorize access to resources based on tokens issued by Keycloak.

Creating a Spring Boot resource server

The code examples presented in this section are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/springboot/backend
```

First, add the following dependencies to your project to enable OAuth2/Open ID Connect support:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId> spring-boot-starter-oauth2-resource-server</
artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

For backend applications protected using a OAuth2 Bearer Token, the configuration is similar to configuring frontend applications. But here, the application is going to act as a resource server that validates JWT tokens:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8180/auth/realms/myrealm
```

Let's start the application by running the following command at the root directory of your project:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/springboot/backend
$ ./mvnw spring-boot:run
```

Your application should start and be available at `http://localhost:8080`. For accessing resources at the running application, you now need an access token. To obtain one, use the following command:

```
$ export access_token=$(\
    curl -X POST http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/token \
    --user mybackend:CLIENT_SECRET\
    -H 'content-type: application/x-www-form-urlencoded' \
    -d 'username=alice&password=alice&grant_type=password' | jq
--raw-output '.access_token' \
 )
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the preceding command with the secret generated by Keycloak for the **mybackend** client. For that, go the **mybackend** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

Once you run this command, an access token will be saved in an `access_token` environment variable, and you can now access the application:

```
$ curl -X GET \
  http://localhost:8080 \
  -H "Authorization: Bearer "$access_token
```

As a result, you should expect the following output from that command:

```
$ Greetings from Spring Boot!
```

Now, if you try to access the application without a `Bearer` token or use an invalid one, you should get a 401 status code, indicating that your request was forbidden:

```
$ curl -v -X GET \
  http://localhost:8080
```

In this section, you learned about how to use Spring Security's OAuth2/OpenID libraries to integrate with Keycloak. We only covered the main steps for setting up Spring Security here so that you can authenticate your users through Keycloak. For more details, please check out the Spring Security documentation at `https://docs.spring.io/spring-security-oauth2-boot/docs/current/reference/html5/`.

In the next section, we are going to look at Keycloak adapters, which we can use as an alternative in case none of the integration options we've presented so far work for you.

Using Keycloak adapters

In addition to the Keycloak server itself, there are several client libraries under the Keycloak umbrella that provide integration with different languages, frameworks, web containers, and application servers.

Also known as Keycloak adapters, these client implementations are meant for integrating with Keycloak, so you should not expect from them to work with other OAuth2 and OpenID Connect servers.

By being specific to Keycloak, you should expect a deeper integration from Keycloak adapters with Keycloak where specific features or capabilities can't be found in any other standard compliant client implementation.

> **Tip**
>
> It may sound like these adapters are the best you can get for integrating with Keycloak, but you should always prefer using generic OpenID Connect libraries, as well as whatever comes for free from the stack you are using, as mentioned in the *Choosing an integration option* section.

We are not going to go deeper into all the Keycloak adapters. Instead, we will quickly iterate over each one and point you to their respective documentation and examples.

In general, these adapters rely on hooks provided by the underlying programming language, framework, web container, and application server they are related to, so you should expect differences when using each adapter. However, regardless of the adapter you choose, you will be using a `keycloak.json` file to configure the adapter and how your application will be interacting with Keycloak to authenticate, authorize, and log out users.

From a configuration perspective, you should expect the same configuration experience across the different adapters. However, you may experience some gaps in some of them due to limitations and constraints from the underlying runtime.

In the following sections, you are going to learn about the different types of adapters and the types of applications they are meant for. We are not going to deep dive into the details because you will find the comprehensive documentation and examples for all the supported adapters in the Keycloak Securing Applications documentation at `https://www.keycloak.org/docs/latest/securing_apps/` and in the Keycloak Quickstarts repository at `https://github.com/keycloak/keycloak-quickstarts`.

Using WildFly and the Red Hat Enterprise Application Platform (EAP)

The Keycloak WildFly and EAP adapter are targeted at applications that have been deployed to the WildFly JEE application server or to the Red Hat EAP.

You can use this adapter by following two main patterns:

- Embedded configuration
- Managed configuration

The **embedded configuration** means that the adapter's configuration is defined in a `keycloak.json` file within your application.

In the **managed configuration**, the adapter's configuration is external to your application and managed through the Keycloak adapter subsystem.

The main difference between these two approaches is whether you need to redeploy your application due to configuration changes. In the managed configuration, changes to the configuration are made via the application server management interfaces.

However, the managed configuration is usually not in sync with the latest configuration options that you can set, and that makes the embedded configuration more appealing.

For more details about this adapter, check out the Keycloak WildFly and EAP adapter documentation:

`https://www.keycloak.org/docs/latest/securing_apps/#jboss-eap-wildfly-adapter`

Using JBoss Fuse

For more details about this adapter, check out the Keycloak Fuse adapter documentation:

`https://www.keycloak.org/docs/latest/securing_apps/#_fuse7_adapter`

Using a web container

For more details about this adapter, check out the following documentation:

- Keycloak Tomcat adapter documentation: `https://www.keycloak.org/docs/latest/securing_apps/#_tomcat_adapter`
- Keycloak Jetty adapter documentation: `https://www.keycloak.org/docs/latest/securing_apps/#_jetty9_adapter`

Desktop applications

The Keycloak desktop adapter is a handy library that leverages Keycloak's capabilities to authenticate users using a Java desktop application.

It relies on the system's default browser to redirect users to Keycloak to authenticate, and also allows you to have access to tokens issued by Keycloak once the user has been successfully authenticated.

By using this adapter, you should be able to authenticate your users using Kerberos tickets, where Keycloak acts a broker to your existing Kerberos infrastructure.

For more details about this adapter, check out the following Keycloak desktop adapter documentation:

`https://www.keycloak.org/docs/latest/securing_apps/#_installed_adapter`.

In this section, you learned about the different options for integrating and protecting your Java application with Keycloak. You learned how to leverage some of the capabilities provided by two common frameworks, Quarkus and Spring Boot, and that Keycloak provides client implementations for applications running on any of the supported web containers and application servers, as well as how to integrate Keycloak with desktop applications.

In the upcoming sections, we are going to look at more integration options when it comes to using different programming languages.

Integrating with JavaScript applications

You will find different OpenID Connect client implementations for JavaScript that you can use to integrate Keycloak with your **Single-Page Applications** (**SPA**).

In this section, we are going to cover how to use the Keycloak JavaScript adapter, a client implementation provided by Keycloak that is targeted at JavaScript-based applications running in a browser, as well as for those using React.JS or React Native.

The code examples for this section are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/keycloak-js-adapter
```

In the preceding directory, you will find all the code you'll need to follow and run the upcoming examples.

The first step to configuring your application with the Keycloak JS adapter is adding the `keycloak.js` library to your page:

```
<script type="text/javascript" src="KC_URL/js/keycloak.js"></
script>
```

Here, `KC_URL` is the URL where your Keycloak server is available, such as `http://localhost:8180/auth` if you are running it locally.

> **Tip**
> By fetching the library from the server as opposed to embedding it in your application, you are guaranteed to always be using the version of the library that is compatible with the Keycloak server your application is talking to.

Now that the library is available on your page, you need to create a `keycloak` object with the client's information and initialize it when the browser window is loaded:

```
keycloak = new Keycloak({ realm: 'myrealm', clientId:
'mybrowserapp' });
keycloak.init({onLoad: 'login-required'}).success(function () {
    console.log('User is now authenticated.');
    profile();
}).error(function () {
    window.location.reload();
});
```

Similarly, to other types of adapters provided by Keycloak, the client information can also be fetched from a `keycloak.json` file at the root path of your application.

The `init` method is responsible for bootstrapping the adapter and returning a promise that we can use to perform actions, based on whether the user is authenticated or when an error occurs during this process.

When your page is loaded for the first time, the adapter is going to check whether the user is already authenticated. If they aren't authenticated yet, the adapter is going to redirect the user to Keycloak. Once the user is successfully authenticated and returns to your application, the adapter will run the function defined by the `success` callback, which, in turn, is going to show a page with information about the user.

Now, let's start the application by running the following code:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/keycloak-js-adapter
$ npm install
$ npm start
```

Your application should start and be available at `http://localhost:8080`. Try accessing that URL and logging into Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

If your application needs to access protected resources in some backend server using a bearer token, you can easily obtain the access token from the `keycloak` object and pass it over when you make HTTP requests:

```
function sendRequest() {
    var req = new XMLHttpRequest();
    req.onreadystatechange = function() {
        if (req.readyState === 4) {
            output(req.status + '\n\n' + req.responseText);
        }
    }
    req.open('GET', 'https://myservice.url', true);
    req.setRequestHeader('Authorization', 'Bearer ' + keycloak.
token);
    req.send();
}
```

The Keycloak JavaScript adapter allows you to quickly integrate with Keycloak. This library was built due to the lack of good JavaScript libraries for OpenID Connect at the time it was created, which does not hold true anymore due to the number of libraries available today. This adapter is actively maintained under the Keycloak umbrella and is well-documented, but still specific to integrating with Keycloak as opposed to being a generic and fully compliant OpenID Connect library.

> **Note**
> Using OpenID Connect and OAuth2 in browser-based applications is surrounded by security concerns due to their nature. When it comes to choosing a good library, you should follow the best practices as per the OAuth2 Security Best Practices for Browser-Based Apps, available at `https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps`.

We have only scratched the surface here and there is far more you can do with it, such as obtaining tokens issues from the server, refresh tokens or automatically doing this based on a certain period of time, and logouts.

For more details about the Keycloak JavaScript adapter, check out the documentation at `https://www.keycloak.org/docs/latest/securing_apps/#_javascript_adapter`.

In the next section, we are going to continue looking at how to integrate with Node.js applications.

Integrating with Node.js applications

For Node.js applications, Keycloak provides a specific adapter called Keycloak Node.js Adapter. Like other adapters, it is targeted at integration with Keycloak rather than a generic OpenID Connect client implementation.

The Keycloak Node.js adapter hides most of the internals from your application through a simple API that you can use to protect your application resources. The adapter is available as an `npm` package and can be installed into your project as follows:

```
$ npm install keycloak-connect
```

The code examples for this section are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/nodejs
```

In the preceding directory, you will find a `frontend` directory and a `backend` directory, which contain all the code you'll need to follow and run the following examples.

Now that you have installed the `keycloak-connect` dependency on your application, we are going to look at how to configure your application as a client and as a resource server.

Creating a Node.js client

Once you've installed the `keycloak-connect` package, you need to change your application code so that it creates a `keycloak` object:

```
var keycloak = new Keycloak({ store: memoryStore });
```

Since we are protecting a frontend application, we want to create a local session for our users so that they are not redirected to Keycloak once they are authenticated. For that, note that the Keycloak object is created with a `memoryStore`:

```
var memoryStore = new session.MemoryStore();
```

Just like other Keycloak adapters, the configuration is read from a `keycloak.json` file containing the client configuration:

```
{
  "realm": "myrealm",
  "auth-server-url": "${env.KC_URL:http://localhost:8180}",
  "resource": "mywebapp",
  "credentials" : {
      "secret" : CLIENT_SECRET
  }
}
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the `keycloak.json` file with the secret that was generated by Keycloak for the **mywebapp** client. For that, go the **mywebapp** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

The next step is to install the adapter as a middleware so that you can use it to protect the resources in your application:

```
app.use(keycloak.middleware());
```

Now that the middleware has been installed, protecting the resources in your application should be as simple as doing the following:

```
app.get('/', keycloak.protect(), function (req, res) {
    res.setHeader('content-type', 'text/plain');
    res.send('Welcome!');
});
```

The `keycloak.protect` method automatically adds the necessary capabilities to your endpoints, to check whether users are authenticated yet or not so that they can be redirected to Keycloak if not. After successful authentication, the middleware will automatically process the response from Keycloak and establish a local session for the user based on the tokens issued by the server.

Now, let's start the application:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/nodejs/frontend
$ npm install
$ npm start
```

Your application should start and be available at `http://localhost:8080`. Try to access that URL and log into Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

Creating a Node.js resource server

The code examples presented in this server are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/nodejs/backend
```

For backend applications, you can create a `keycloak` object as follows:

```
var keycloak = new Keycloak({});
```

Compared to frontend applications, we do not need to track user sessions; instead, we must rely on bearer tokens to authorize requests.

Similar to the previous example, we also need to update the `keycloak.json` file with the client configuration:

```
{
  "realm": "myrealm",
  "bearer-only": true,
  "auth-server-url": "${env.KC_URL:http://localhost:8180/
auth}",
  "resource": "mybackend"
}
```

In this configuration, we are explicitly marking this application so that it only accepts bearer tokens, forcing the adapter to check whether a request can access resources in the application by performing local validations and introspections on the token.

The next step is to install the adapter as a middleware so that you can use it to protect the resources in your application:

```
app.use(keycloak.middleware());
```

Now that the middleware has been installed, protecting the resources in your application should be as simple as doing the following:

```
app.get('/protected', keycloak.protect(), function (req, res) {
    res.setHeader('content-type', 'text/plain');
    res.send('Access granted to protected resource');
});
```

The `keycloak.protect` method automatically adds bearer token authorization to your endpoints so that requests containing an authorization header with a valid token can fetch the protected resources in your application.

Now, let's start the application:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/nodejs/backend
$ npm install
$ npm start
```

Your application should start and be available at `http://localhost:8080`. To access the resources at the running application, you will need an access token. To obtain one, use the following command:

```
$ export access_token=$(\
    curl -X POST http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/token \
    --user mybackend:CLIENT_SECRET\
    -H 'content-type: application/x-www-form-urlencoded' \
    -d 'username=alice&password=alice&grant_type=password' | jq
--raw-output '.access_token' \
 )
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the preceding command with the secret generated by Keycloak for the **mybackend** client. For that, go the **mybackend** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

Once you've run that command, an access token will be saved in an `access_token` environment variable, which means you can now access the application:

```
$ curl -v -X GET \
  http://localhost:8080/protected \
  -H "Authorization: Bearer "$access_token
```

As a result, you should expect the following output:

```
$ Access granted to protected resource
```

Now, if you try to access the application without a `Bearer` token or use an invalid one, you should get a 403 status code, indicating that your request was forbidden:

```
$ curl -v -X GET \
  http://localhost:8080/protected
```

There is much more you can do with the Keycloak Node.js adapter in terms of configuration and usage. You should be able to use `keycloak.protect` to perform role-based access controls and obtain the tokens representing the authenticated subject.

The Keycloak Node.js adapter is actively maintained under the Keycloak umbrella, but it's still specific to integrating with Keycloak as opposed to being a generic and fully compliant OpenID Connect library.

For more details about the Keycloak JavaScript adapter, check out the available documentation at `https://www.keycloak.org/docs/latest/securing_apps/#_nodejs_adapter`.

In this section, you learned how to configure your Node.js application so that you can integrate with Keycloak using the `keycloak-connect` library. Next, you will learn how to integrate Python applications with Keycloak.

Integrating with Python applications

Python applications that use Flask can easily enable OpenID Connect and OAuth2 to applications through the Flask-OIDC library. It can be used to protect client as well as resource server applications.

> **Tip**
> If you are looking for a library to enable OpenID Connect support for command-line interfaces, there are different OpenID Connect client implementations you can use at `https://openid.net/developers/certified/`.

To install Flask-OIDC, run the following command:

```
$ pip install Flask-OIDC
```

The code examples for this section are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/python
```

In the preceding directory, you will find a `frontend` directory and a `backend` directory containing all the code you will need to follow and run the following examples.

In the next section, we are going to start looking at how to enable a web application in order to authenticate users using Keycloak.

Creating a Python client

The code examples presented here are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/python/frontend
```

To enable authentication to your web application, you will need a configuration file called `oidc-config.json` at the root path of your program that contains some metadata about the endpoints and client information from Keycloak:

```
{
  "web": {
    "client_id": "mywebapp",
    "client_secret":CLIENT_SECRET,
    "auth_uri": "http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/auth",
    "token_uri": "http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/token",
    "issuer": "http://localhost:8180/auth/realms/myrealm",
    "userinfo_uri": "http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/userinfo",
    "redirect_uris": [
      "http://localhost:8080/oidc/callback"
    ]
  }
}
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the preceding configuration with the secret generated by Keycloak for the **mywebapp** client. For that, go the **mywebapp** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

Finally, create an application by creating an `app.py` file with the following content:

```
from flask import Flask
app = Flask(__name__)
app.secret_key = 'change_me'
app.config['OIDC_CLIENT_SECRETS'] = 'oidc-config.json'
app.config['OIDC_COOKIE_SECURE'] = False
from flask_oidc import OpenIDConnect
oidc = OpenIDConnect(app)

@app.route('/')
@oidc.require_login
def index():
    if oidc.user_loggedin:
        return 'Welcome %s' % oidc.user_getfield('preferred_
username')
    else:
        return 'Not logged in'
```

To start the application, run Flask and configure it to run on port `8080`:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/python/frontend
$ flask run -p 8080
```

Your application should start and be available at `http://localhost:8080`. Try to access that URL and log into Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

Creating a Python resource server

The code examples presented here are available from the following GitHub repository:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/nodejs/backend
```

To protect the resources in your application, you need to annotate your endpoints with `@oidc.accept_token()`:

```
import json
from flask import Flask, g
app = Flask(__name__)
app.secret_key = 'change_me'
app.config['OIDC_CLIENT_SECRETS'] = 'oidc-config.json'
app.config['OIDC_RESOURCE_SERVER_ONLY'] = 'true'
from flask_oidc import OpenIDConnect
oidc = OpenIDConnect(app)

@app.route('/, methods=['POST'])
@oidc.accept_token(True)
def api():
    return json.dumps({'hello': 'Welcome %s' % g.oidc_token_
info['preferred_username']})
```

Create an `oidc-config.json` file at the root path of your program with some metadata about the endpoints and client information from Keycloak:

```
{
  "web": {
    "client_id": "mybackend",
    "client_secret": CLIENT_SECRET,
    "auth_uri": "http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/auth",
    "token_uri": "http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/token",
    "issuer": "http://localhost:8180/auth/realms/myrealm",
    "token_introspection_uri": "http://localhost:8180/auth/
realms/myrealm/protocol/openid-connect/token/introspect",
    "redirect_uris": [
```

```
        "http://localhost:8080/oidc/callback"
      ]
    }
}
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the preceding configuration with the secret generated by Keycloak for the **mybackend** client. For that, go the **mybackend** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

To start the application, run `flask` and configure it to run on port `8080`:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/python/backend
$ flask run -p 8080
```

Your application should start and be available at `http://localhost:8080`. To access the resources at the running application, you will need an access token. To obtain one, use the following command:

```
$ export access_token=$(\
    curl -X POST http://localhost:8180/auth/realms/myrealm/
protocol/openid-connect/token \
    --user mybackend:CLIENT_SECRET\
    -H 'content-type: application/x-www-form-urlencoded' \
    -d 'username=alice&password=alice&grant_type=password' | jq
--raw-output '.access_token' \
 )
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the preceding command with the secret generated by Keycloak for the **mybackend** client. For that, go the **mybackend** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

Once you've run that command, an access token will be saved in an `access_token` environment variable, which means you can now access the application:

```
$ curl -v -X POST -d 'access_token='$access_token \
      -H "Content-Type: application/x-www-form-urlencoded" \
      http://localhost:8080
```

Now, if you try to access the application without a `Bearer` token or use an invalid one, you should get a 401 status code, indicating that your request was forbidden:

```
$ curl -v -X POST \
  http://localhost:8080
```

`@oidc.accept_token()` does not support bearer tokens sent via the Authorization HTTP header, as per RFC 6750 - Bearer Token Usage, but they can be sent as parameters when you're making `GET` and `POST` requests.

This is not in compliance with the standards and is not the best way to send tokens to applications, especially if you're using the `GET` HTTP method.

Flask-OIDC is one of the available options for integrating Python applications with Keycloak. It provides support for protecting applications acting as clients, as well as resource servers.

The fact that bearer tokens require HTTP requests to pass the token via parameters is probably something you want to change. This will help you follow the best practices surrounding bearer token authorization.

The library also supports methods that are specific to integrating with Keycloak, such as checking whether a token is carrying a specific client role.

In this section, you learned about the possibility to integrate Python applications with Keycloak using the Flask-OIDC library. If this library is not part of your technology stack, you can still leverage any other library or framework for the same purpose, as long it is compliant with the OpenID Connect specifications.

In the next section, we are going to look an integration option related to the proxied architectural style, which is useful if none of the options that have been presented so far are enough to address your requirements.

Using a reverse proxy

By running in front of your application, you can use reverse proxies to add additional capabilities to your application. The most common proxies provide support for OpenID Connect where enabling authentication is a matter of changing the proxy configuration.

Whether using a proxy is better than having the integration code and configuration within your application really depends on the use case and, depending on the circumstances, it might be your only option or the option that will save you precious time from implementing your own integration code, even if you have a library available for the technology stack your application is using.

Nowadays, OpenID Connect and OAuth2 support is a mandatory capability for proxies, and you find support for these protocols in most of them, regardless of whether they're open source or proprietary. As an example, two of the most popular proxies, Apache HTTP Server and Nginx, provide the necessary extensions for these protocols.

In this section, we are going to cover how to set up Apache HTTP Server in front of our application so that we can integrate it with Keycloak and authenticate users using `mod_auth_oidc`. The documentation on how to install it is available at `https://github.com/zmartzone/mod_auth_openidc`.

Once the module has been installed, we need to configure the server so that we can proxy our application and use the module to make sure users are authenticated through Keycloak:

```
LoadModule auth_openidc_module modules/mod_auth_openidc.so
ServerName localhost

<VirtualHost *:80>
    ProxyPass / http://localhost:8000/
    ProxyPassReverse / http://localhost:8000/

    OIDCCryptoPassphrase CHANGE_ME

    OIDCProviderMetadataURL http://localhost:8180/auth/realms/
myrealm/.well-known/openid-configuration

    OIDCClientID mywebapp
    OIDCClientSecret CLIENT_SECRET
    OIDCRedirectURI http://localhost/callback
```

```
    OIDCCookieDomain localhost
    OIDCCookiePath /
    OIDCCookieSameSite On

    <Location />
        AuthType openid-connect
        Require valid-user
    </Location>
</VirtualHost>
```

> **Note**
>
> You should change the reference to `CLIENT_SECRET` in the preceding configuration with the secret generated by Keycloak for the **mywebapp** client. For that, go the **mywebapp** client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Secret** field in this tab.

Now, let's start the application:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-
Applications/ch7/reverse-proxy/app/
$ npm install
$ npm start
```

Your application should start and be available at `http://localhost`. Try to access that URL and log into Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

Try not to implement your own integration

OAuth2 and OpenID Connect are simple protocols, and their simplicity is, in part, due to the effort that's been made to make the protocol easier to implement by client applications. You may feel tempted to write your own code to integrate with Keycloak, but this is usually a bad choice.

You should rely on well-known and widely used libraries, frameworks, or capabilities provided by the platform where your application is deployed. By doing that, you can focus on your business and, most importantly, delegate to people specialized and focused on these standards to keep their implementations always up to date with the latest versions of the specifications, as well as with any fixes for security vulnerabilities.

Also, remember that the more people there are using an implementation, the less likely it is that you will face bugs and security vulnerabilities that can impact not only your application, but also your organization.

Summary

In this chapter, you learned how to integrate Keycloak with different types of applications, depending on the technology stack they are using, as well as the platform they are running. You also learned about the importance of using well-known and established open standards and what that means in terms of interoperability. This means you are free to choose the OpenID Connect client implementation that better serves your needs, while still respecting compliance and keeping your applications up to date with the OAuth2 and OpenID Connect best practices and security fixes.

Finally, you learned why you should avoid implementing your own integration, as well as the things you should consider when you're looking for alternatives, if none of the other options work for you.

In the next chapter, you will learn about the different authorization strategies you can use to protect your application resources.

Questions

1. What is the best way to integrate with Keycloak?
2. Should I always consider using the Keycloak adapters if they fit into my technology stack?
3. How should you secure a native or mobile application with Keycloak?
4. What is the best integration option for cloud-native applications?