Adolfo Eloy Nascimento

OAuth 2.0 Cookbook

Protect your web applications using Spring Security

Packt

OAuth 2.0 Cookbook

Protect your web applications using Spring Security

Adolfo Eloy Nascimento

BIRMINGHAM - MUMBAI

OAuth 2.0 Cookbook

First published: October 2017

Production reference: 1161017

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78829-596-3

www.packtpub.com

Credits

Author Adolfo Eloy Nascimento	**Copy Editors** Juliana Nair Stuti Srivastava
Reviewer Rafael Monteiro e Pereira	**Project Coordinator** Judie Jose
Commissioning Editor Vijin Boricha	**Proofreader** Safis Editing
Acquisition Editor Rahul Nair	**Indexer** Francy Puthiry
Content Development Editor Nikita Pawar	**Graphics** Kirk D'Penha
Technical Editor Prachi Sawant	**Production Coordinator** Nilesh Mohite

About the Author

Adolfo Eloy Nascimento is a software engineer at Elo7, he has a Bachelors degree in Computer Science, and has been working with software development since 1999. In around 2003, he started working with web development implementing applications using ASP, PHP4/5, JavaScript, and Java (sometimes he still does some maintenance for a Ruby on Rails application). He started using OAuth 2.0 two years ago, when designing applications using microservice architectures, as well as modeling and interacting with public APIs.

As a tech enthusiast, Adolfo also likes to read and learn about programming languages and new technologies. He also believes that besides creating new applications, it is also important to share the knowledge he has acquired, which is what he does by writing for his personal blog, writing articles for Java Magazine in Brazil, and also writing tech books.

I would like to thank Janine for all her support and understanding during the writing of this book. Without her support it wouldn't have been possible. In addition, I would like to thank my parents who gave me great support in education. I would also like to thank Rafael Monteiro for accepting the reviewing of this book and improving the quality of each chapter. Finally, I would like to thank the Packt team, especially Rahul and Nikita, for helping me during the whole process of writing this book.

About the Reviewer

Rafael Monteiro e Pereira is a graduate in Computer Science from Mackenzie University in São Paulo, Brazil. He has always liked developing mission-critical and high-performance software. He also likes software security, especially playing with Kali Linux and its amazing set of tools. There is always a new vulnerability out there waiting to be discovered; this is what he believes.

He worked for the startup Elo7 as a big data/search software engineer and for companies in the finance/banking sector, such as BM&F Bovespa, developing their trading platform, and for Itaú-Unibanco as a lead software engineer on their new big data platform.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Table of Contents

Preface

OAuth 2.0 is a standard protocol for authorization and it focuses on client-developer simplicity, while providing specific authorization flows for web applications, desktop applications, mobile phones, and so on. Given the documentation available for OAuth specification, you may think that it is complex; however, this book promises to help you start using OAuth 2.0 through examples in simple recipes. It focuses on providing specific authorization flows for various applications through interesting recipes. It also provides useful recipes for solving real-life problems using Spring Security and creating Android applications.

What this book covers

Chapter 1, OAuth 2.0 Foundations, contains recipes that will cover the basics of OAuth 2.0 through simple recipes that allow the reader to interact with public OAuth 2.0-protected APIs such as Facebook, LinkedIn, and Google.

Chapter 2, Implement Your Own OAuth 2.0 Provider, describes the way you can implement your own OAuth 2.0 Provider, presenting recipes that help with Authorization Server and Resource Server configurations considering different OAuth 2.0 grant types. It also presents how to effectively work with refresh tokens, using different databases to store access tokens.

Chapter 3, Using OAuth 2.0 Protected APIs, presents recipes that helps to create OAuth 2.0 client applications that are able to interact with all grant types described in the OAuth 2.0 specification. It also presents how to manage refresh tokens on the client side.

Chapter 4, OAuth 2.0 Profiles, explains some OAuth 2.0 profiles and how to implement them using Spring Security OAuth2. These profiles are specified to help with specific scenarios that aren't covered by OAuth 2.0 specifications, such as token revocation and token introspection to allow remote validation. This recipe also provides some recommendations, such as how and when to use cache when using remote validation.

Chapter 5, Self Contained Tokens with JWT, focuses on the usage of JWT as OAuth 2.0 access tokens and how to implement the main extensions for JWT, such as JWS and JWE, providing signature and encryption to protect the content conveyed by a JWT access token. This chapter also presents a nice approach to increase the security of your application by using proof-of-possession semantics on OAuth 2.0.

Chapter 6, OpenID Connect for Authentication, explains the difference between authorization and authentication, and how OAuth 2.0 can help to build an authentication protocol. To illustrate the usage of OpenID Connect, all the recipes presented in this chapter are aimed at client applications instead of building an OpenID Connect Provider.

Chapter 7, Implementing Mobile Clients, covers how to implement OAuth 2.0 native mobile clients using Android as the platform chosen for the recipes. This chapter presents some guidelines specified by the recently published specification named OAuth 2.0 for native apps.

Chapter 8 , Avoiding Common Vulnerabilities, covers ways to better protect the main components considered within an OAuth 2.0 ecosystem.

What you need for this book

To run the recipes presented in this book, you will basically need JDK 8, Maven, MySQL, and Redis. JDK 8 can be downloaded at http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html . You can download and read the installation instructions for Maven at https://maven.apache.org/download.cgi. To install MySQL, download the community version for your Operational System (OS) at https://dev.mysql.com/downloads/. Some recipes rely on Redis, which can be downloaded here: https://redis.io/download. To interact with the applications that will be created during the recipes, you also need a tool to send HTTP requests to the APIs presented. The recommended tools are CURL, which can be downloaded at https://curl.haxx.se/download.html and PostMan which can be downloaded at https://www.getpostman.com/.

In addition, so that you can write the code presented throughout the recipes, you will also need a Java IDE and Android Studio for native mobile Client recipes.

Who this book is for

This book targets software engineers and security experts who are looking at developing their skills in API security and OAuth 2.0. It is also aimed to help developers who want to pragmatically add OAuth 2.0 support for Spring Boot applications as well as Android mobile applications. Prior programming knowledge and basic understanding of web development is necessary. As this book presents the most recipes using Spring Security OAuth2, it would help to have prior experience with Spring Framework.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it…, How it works…, There's more…, and See also). To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Let's create the first web page as `index.html` inside the `src/main/resources/templates` directory" A block of code is set as follows:

```
public class Entry {
    private String value;
    public Entry(String value)
     { this.value = value; }
    public String getValue()
     { return value; }
}
```

Any command-line input or output is written as follows:

```
curl -X POST --user clientapp:123456 http://localhost:8080/oauth/token
 -H "content-type: application/x-www-form-urlencoded"
 -d "code=5sPk8A&grant_type=authorization_code&redirect_uri=http%3A%2F%2Floca
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "click on Authorize so you get redirected back to the redirect URI callback."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors .

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the Search box. Please note that you need to be logged in to your Packt account. Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/OAuth2.0Cookbook_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata is verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

OAuth 2.0 Foundations

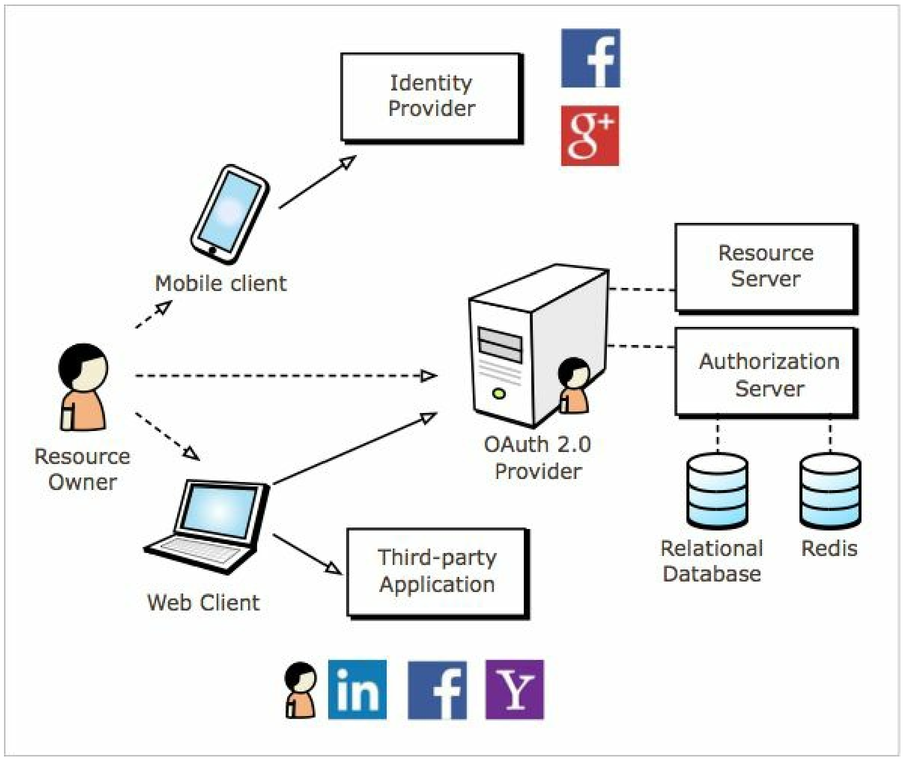
This chapter covers the following recipes:

- Preparing the environment
- Reading the user's contacts from Facebook on the client side
- Reading the user's contacts from Facebook on the server side
- Accessing OAuth 2.0 LinkedIn protected resources
- Accessing OAuth 2.0 Google protected resources bound to the user's session

Introduction

The main purpose of this chapter is to help you integrate with popular web applications and social media, although at the same time allow you to get familiarized with the foundational principles of OAuth 2.0 specification.

Before diving into the recipes for several use cases, let's look at the big picture of the most scenarios which will be covered. This will give you the opportunity to review some important concepts about OAuth 2.0 specification so we can stay on the same page with the terminologies used throughout the book.



The preceding diagram shows the four main components of the OAuth 2.0 specification:

- Resource Owner
- Authorization Server
- Resource Server

- Client

Just to review the purpose of these components, remember that the **Resource Owner** is the user which delegates authority for third-party applications to use resources on its behalf. The third-party application mentioned is represented by the client which I depicted as **Mobile client** and **Web Client**. The user's resources are usually maintained and protected by the **Resource Server** which might be implemented together with the **Authorization Server** as a single component, for example. The composition of the **Authorization Server** and **Resource Server** are referred to as the **OAuth 2.0 Provider** to simplify the terminology given to the application which is protected by OAuth 2.0.

Preparing the environment

As most examples are written in Java, we will also need an **Integrated Development Environment** (**IDE**) and a good framework to help us write simple web applications (as the OAuth 2.0 protocol was designed for HTTP usage), which will be Spring. To simplify the usage of Spring related technologies, this recipe will help you prepare an application using Spring Boot, providing an example endpoint and how to run this project using Maven.

Getting ready

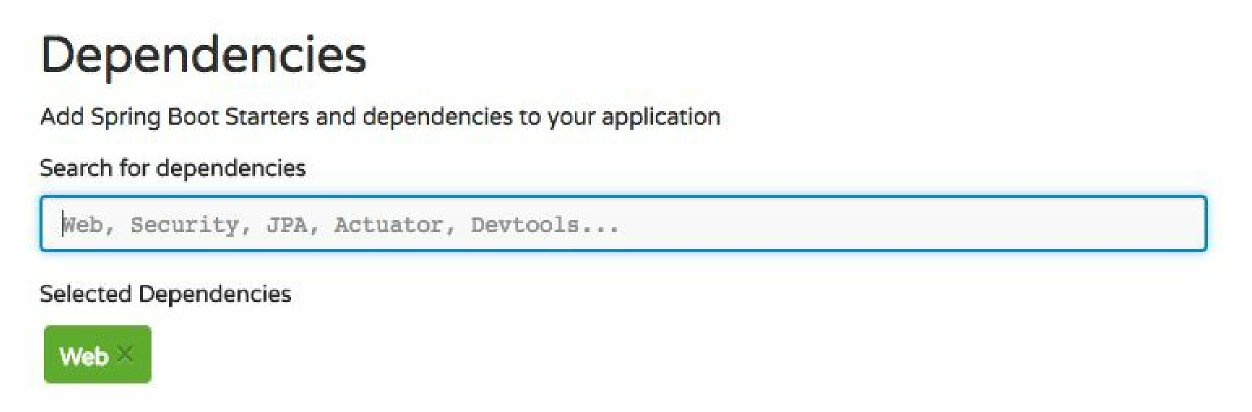
As I previously mentioned, we will run most of the recipes using the Spring Boot Framework which eases the development of applications based on the Spring Framework. So to run this recipe, you just need an environment where you can download some files from the internet, Java 8 properly configured on your machine, and the `CURL` tool.

`CURL` is a tool which allows you to run HTTP requests through the command line. It is available by default in Linux and Mac OS environments, so if you are running the recipes on Windows you should install it first. This tool can be downloaded from https://curl.haxx.se/download.html and to install it, you just have to unpack it and add the path for binaries to the PATH environment variable of Windows.

How to do it...

The following steps describe how to prepare the environment and show how to generate a simple project from the **Spring Initializr** website which will be executed using the appropriate Maven commands:

1. Generate a project using Spring Initializr service by visiting https://start.spring.io/. Spring Initializr provides lots of options to start setting up your project, such as if you want to use Maven or Gradle to manage your project dependencies, which version of Spring Boot to use, which dependencies to use, and even changing the language from Java to Groovy or Kotlin.
2. For this simple test, just use the default values for the project manager, Maven Project, with Java language and version 1.5.7 of the Spring Boot.
3. At Project Metadata, change the value of the field Group to `com.packt.example`.
4. Still on Project Metadata, change the name of the Artifact to `simplemvc`.
5. In the Dependencies section, type `web` and select Full-stack web development with Tomcat and Spring MVC. After selecting the right choice, you will see the tag `Web` underneath Selected Dependencies as follows:


Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

6. After setting up all the requirements for this simple example, click on the Generate Project button and your browser will start downloading the ZIP file into your `Downloads` folder.

7. After downloading this file, you can unzip it and import it to your IDE just to explore the structure of the created project. For Eclipse users, just import the project as a Maven project.

8. Open the class `SimplemvcApplication` and you would see the following code in your IDE:

```
@SpringBootApplication
public class SimplemvcApplication {
   public static void main(String[] args) {
         SpringApplication.run(SimplemvcApplication.class, args);
   }
}
```

9. Let's turn the class `SimplemvcApplication` into a controller by adding the annotation `@Controller` as presented in the following code:

```
@Controller @SpringBootApplication
public class SimplemvcApplication {
   public static void main(String[] args) {
         SpringApplication.run(SimplemvcApplication.class, args);
   }
}
```

10. Now that our class is declared as a controller, we can define an endpoint so we can see if the project is running properly. Add the method `getMessage` as follows, within the class `SimplemvcApplication`:

```
@GetMapping("/message")
public ResponseEntity<String> getMessage() {
   return ResponseEntity.ok("Hello!");
}
```

11. If you want to run your project inside the Eclipse IDE, you should just run the class `SimplemvcApplication` as a Java application by right-clicking at the class and selecting the menu option Run As | Java Application.
12. After the application is started you should see something like the following message at the end of the output presented in your console:

```
Started SimplemvcApplication in 13.558 seconds (JVM running for 14.011
```

13. Execute the following command to know if your application works properly (just check if the output prints `Hello`):

```
 curl "http://localhost:8080/message"
```

14. If you would like to use the command line you can also start your application by running the following Maven command (to run the application with Maven through the command line, you must install Maven, as explained in the next sections):

```
mvn spring-boot:run
```

15. If you don't have Maven installed on your machine, the first thing to do is to start downloading the latest version from https://maven.apache.org/download.cgi which at the time of this writing was `apache-maven-3.5.0-bin.tar.gz`.
16. After the file has downloaded, just unpack it into any folder you want and start running Maven commands.
17. Copy the full path of the Maven directory, which was created when you unpacked the downloaded file from the Maven website. If you are running macOS or Linux, run `pwd` at the command line to discover the full path.
18. After that, you must add the path for Maven's directory to the `PATH` environment variable. If you are using Linux or macOS, create the variable `MVN_HOME` within the `.bash_profile` file and append the content of `MVN_HOME` to the end of the `PATH` environment variable, as presented in the following code:

```
MVN_HOME=/Users/{your_user_name}/maven-3.5.0
export PATH=$PATH:$MVN_HOME/bin
```

The file `.bash_profile` should be found at the user's directory. So, to edit this file, you should open the file `/Users/{your_user_name}/.bash_profile`, or in a shorter way, by using `~/.bash_profile`. If you are using Windows, all the environment variables can be edited through the visual interface.

19. After editing this file, run the command `source ~/.bash_profile` to reload all the contents.
20. To check if Maven is perfectly running on your environment, run the following command:

```
mvn --version.
```

See also

- The OAuth 2.0 specification is available as RFC 6749 at https://tools.ietf.org/html/rfc6749
- You can read more about Spring Boot at https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

How it works...

Because of the usage of Spring Boot we can take advantage of projects like Spring MVC and Spring Security. These Spring projects help us to write web applications, REST APIs, and help us to secure our applications. By using the Spring Security OAuth2 project, for example, we can configure our own OAuth 2.0 Providers in addition, to act like clients. This is important because someone trying to write his own OAuth Provider will have to deal with too many details which could easily lead to an insecure OAuth Provider. Spring Security OAuth2 already addresses the main concerns any developer would have to think about.

In addition, Spring Boot eases the initial steps for the bootstrap of the application. When creating a Spring project without Spring Boot we need to deal with dependencies manually by taking care of possible library conflicts. To solve this problem, Spring Boot has some pre-configured modules provided by starters. As an example of a useful starter, let's consider an application with Spring Data JPA. Instead of declaring all the dependencies for `hibernate`, `entity-manager`, and `transaction-api`, just by declaring `spring-boot-starter-data-jpa` all the dependencies will be imported automatically.

While starting using Spring Boot, things can still become easier by using the Spring Initializr service provided by Pivotal (the Spring maintainer now).

There's more...

All the examples presented in Java can be imported and executed on any Java IDE, but we will use Eclipse just because it is a large, accepted tool among developers around the world. Although this book presents recipes using Eclipse, you can also stick with your preferred tool if you want.

Nowadays, many projects have been designed using Gradle, but many developers are still used to creating their projects using Maven to manage dependencies and the project itself. So, to avoid trick bugs with IDE plugins or any other kind of issue, the recipes using Spring Boot will be managed by Maven. In addition, Eclipse IDE already comes with a Maven plugin which at the time of writing this book was not true for Gradle. To run projects with Gradle in Eclipse, you must install a specific plugin.

See also

Spring Boot provides a lot of starters to help you develop applications using a plethora of tools and libraries. If you want to search for more just go to http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/reference/htmlsingle/#using-boot-starter.

Reading the user's contacts from Facebook on the client side

This recipe will present you with how you can integrate with Facebook using the Implicit grant type which is the better choice for public clients and runs directly on the user's web browser.

Grant types as you may already know, defines different methods for an application to retrieve access tokens from an Authorization Server. A grant type may apply for a given scenario regarding the client type being developed. Just as a reminder, OAuth 2.0 specification defines two types of client types: ***public*** *and* ***confidential****.*

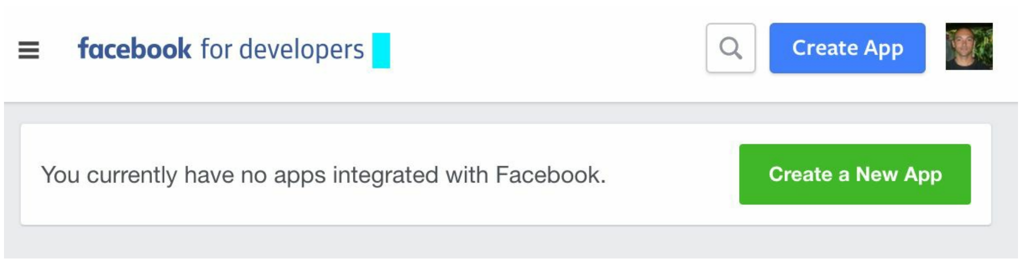
Getting ready

To run this recipe, you must create a web application using Spring Boot, which will help the development of the application. In addition, we also need to register our application on Facebook. That's one important step when using OAuth 2.0, because as an OAuth Provider, Facebook needs to know which clients are asking for access token and, of course, the Resource Owner (the user) would want to know who is to be granted access to her profile.

How to do it...

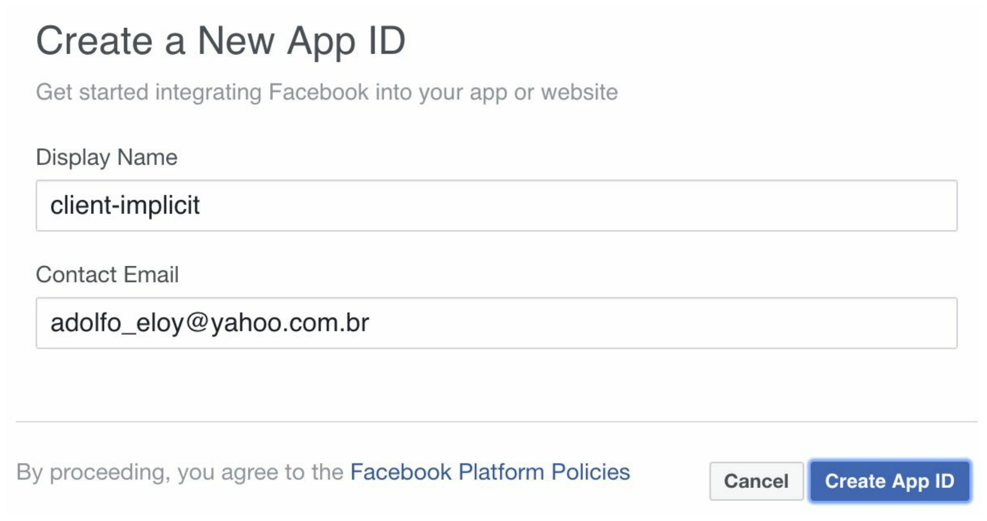
Follow these steps to create a client application to integrate with Facebook using client-side flow from OAuth 2.0:

1. First of all, remember that you must have a Facebook account, and have to register a new application. Go to https://developers.facebook.com/apps/ and you should see something like this:



You currently have no apps integrated with Facebook.

2. Click on Create a New App to start registering your application, and you should see the following interface which allows you to define the name of the application:



Create a New App ID

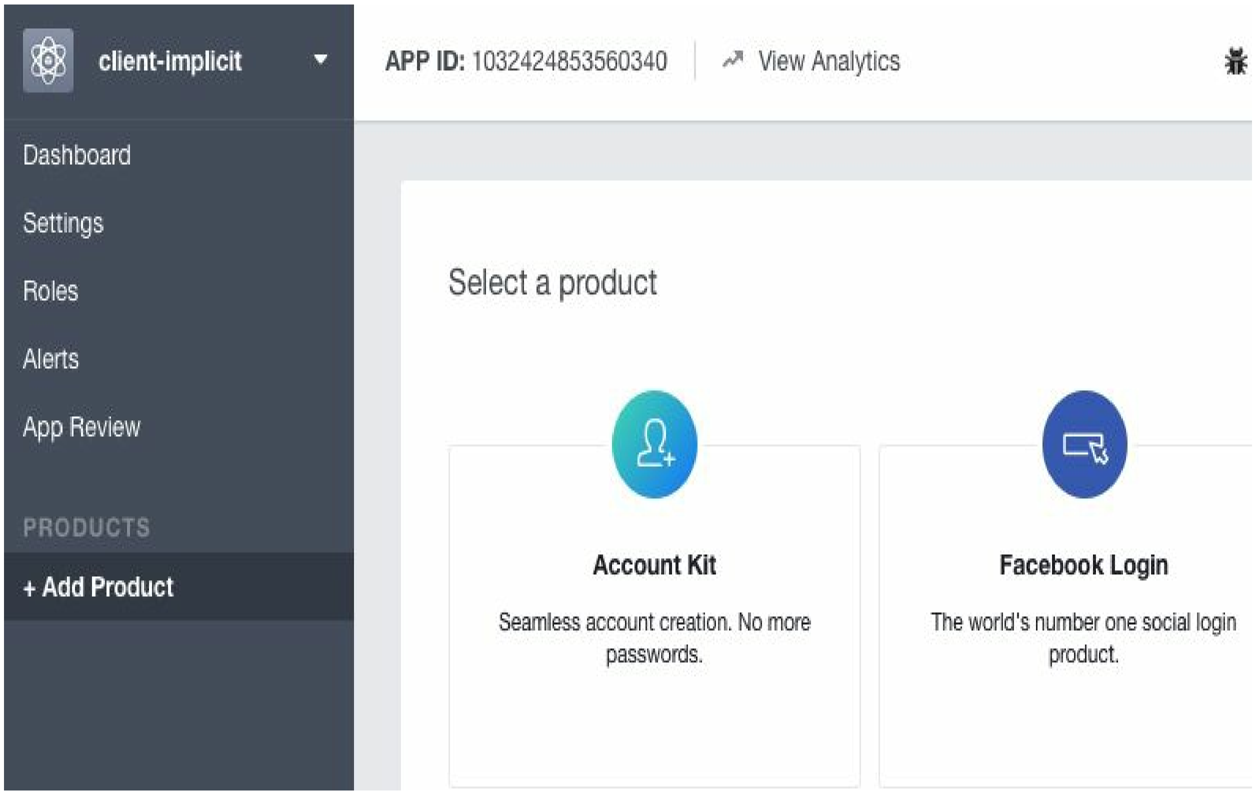
Get started integrating Facebook into your app or website

Display Name: client-implicit

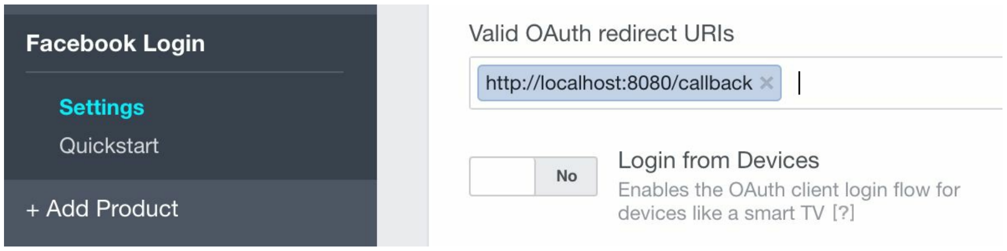
Contact Email: adolfo_eloy@yahoo.com.br

By proceeding, you agree to the Facebook Platform Policies

3. Click on Create App ID and you will be redirected to the newly created application's dashboard as follows:



4. To start using the Facebook's Graph API and retrieve the user's contacts, we first need to select one product from several provided by Facebook. For what we need now, you must click on the Set Up button from the Facebook Login box.
5. After clicking on Set Up, you must choose one platform, which must be Web for this recipe.
6. After choosing the Web platform, enter the Site URL for your application. I am using a fictitious URL named `http://clientimplicit.test`.
7. After saving the URL of your site, just click on Continue.
8. Now you are ready to set up the redirection URI for the application by clicking on Settings in the left panel, as follows. As this application isn't running for production, I have set up the redirect URI as `http://localhost:8080/callback`. Don't forget to save the changes:



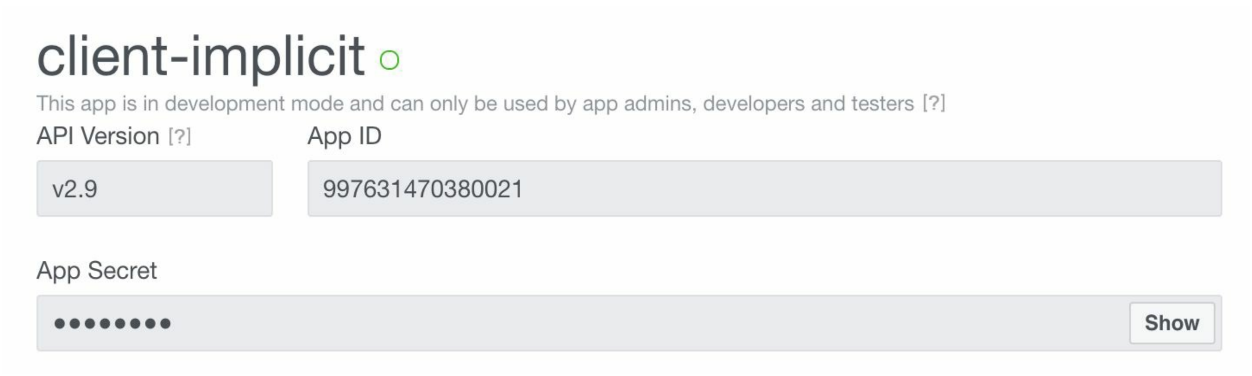
Valid OAuth redirect URIs

http://localhost:8080/callback

No | Login from Devices

Enables the OAuth client login flow for devices like a smart TV [?]

9. Now you can click on Dashboard at the left side of the panel so you can grab the App ID and App Secret, which maps to the `client_id` and `client_secret` from OAuth 2.0 specifications respectively.
10. Copy the App ID and App Secret from the dashboard as represented by the following screenshot as follows, so we can use in the code which comes in the next steps:



client-implicit

This app is in development mode and can only be used by app admins, developers and testers [?]

API Version [?]: v2.9

App ID: 997631470380021

App Secret: ●●●●●●●●

11. Once we have our client registered on Facebook, we are ready to start writing code to retrieve the OAuth 2.0 access token using the **Implicit grant type** (requesting the access token from the client side).
12. Create a new web application using Spring Initializr at https://start.spring.io/ and define the following data:
    - Set up the group as `com.packt.example`
    - Define the artifact as `client-implicit`
    - Add `Web` and `Thymeleaf` as dependencies for this project

13. Create the file `client.html` within the folder `templates` which resides inside the `src/main/resources` project's directory.

14. Add the following content within the file `client.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head><title>Facebook - client side integration</title></head>
```

```
<body>
   Press the following button to start the implicit flow.
   <button id="authorize" type="button">Authorize</button>
   <div id="box"></div>
</body>
</html>
```

15. The button we have added to the `client.html` page does not have any behavior attached. So, to allow the user to start the Implicit grant type flow, add the following JavaScript code after the body tag:

```
<script
   src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.
<script type="text/javascript" th:inline="javascript">
/*<![CDATA[*/
$(document).ready(function() {
   $("#authorize").click(makeRequest);
});

function makeRequest() {
    var auth_endpoint = "https://www.facebook.com/v2.9/dialog/oauth",
         response_type = "token",
         client_id = "1948923582021549",
         redirect_uri = "http://localhost:8080/callback",
         scope = "public_profile user_friends";

   var request_endpoint = auth_endpoint + "?" +
         "response_type=" + response_type + "&" +
         "client_id=" + client_id + "&" +
         "redirect_uri=" + encodeURI(redirect_uri) + "&" +
         "scope=" + encodeURI(scope);

   window.location.href = request_endpoint;
}
/*]]>*/
</script>
```

16. Before starting the application, we need to map a URL pattern so the HTML code we wrote before can be rendered. To do so, open the class `ClientImplicitApplication.java` and assure your code looks like the following:

```
@Controller @SpringBootApplication
public class ClientImplicitApplication {
    public static void main(String[] args) {
         SpringApplication.run(ClientImplicitApplication.class, args);
   }

   @GetMapping("/")
   public String client() { return "client"; }
}
```

17. In the previous code, we've mapped the root of our application to the `client.html` web page. But all of this is now serving the purpose of sending to the user the Authorization Server (in this case Facebook) so she could grant our application access to protected resources (which are her friends). Try to start the application and go to `http://localhost:8080/`.
18. Click on the ;Authorize button that will be provided by `client.html` to start the Implicit grant flow, log in with your Facebook account, and accept the permissions requested by the `client-implicit` application (make sure that jquery is properly declared inside `client.html` file).
19. If you grant all the permissions at the consent user page you shall be redirected to `http://localhost:8080/callback` URL that was specified at the client registration phase on Facebook. Click on Continue and pay attention to the content received and the URL fragment in the browser's address bar. It should be something like the following:

```
http://localhost:8080/callback#access_token=EAAbsiSHMZC60BANUwKBDCYeyS
```

20. Now we need to extract both the `access_token` and the `expires_in` parameters which comes after `#character`, and start using the Facebook Graph API to retrieve the user's friends.

21. The first thing we can do is to create another URL mapping through our default controller, which is `ClientImplicitApplication`. Open this class and add the following method so we can deal with Facebook's redirection:

```
@GetMapping("/callback")
public String callback() { return "callback_page"; }
```

22. As we can see, the method `callback` is returning the `callback_page` string, which will automatically be mapped to the file `callback_page.html`. So, let's create this file inside the templates folder which resides in the `src/main/resources` project directory. At first just add the following HTML content to the `callback_page.html` file:

```
<!DOCTYPE html>
<html>
<head><title>Insert title here</title></head>
<body>
Friends who has also granted client-implicit
<div id="friends">
   <ul></ul>
</div>
```

```
</body>
</html>
```

23. After receiving the `acces_token` as a URL fragment, this file will use JavaScript code to interact with Facebook's graph API to retrieve the user's friends, and will populate the tag `<ul>` with each friend received within the respective `<li>` tag. Let's start writing our JavaScript code by adding the following content after the body tag:

```
<script
   src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.
<script type="text/javascript">
/*<![CDATA[*/
$(document).ready(function() {
   var fragment = window.location.hash;
 });
/*]]>*/
</script>
```

24. As we have the fragment content in the fragment variable, add the following function at the end of the JavaScript code:

```
function getResponse(fragment) {
   var attributes = fragment.slice(1).split('&');
   var response = {};

   $(attributes).each(function(idx, attr) {
         var keyValue = attr.split('=');
         response[keyValue[0]] = keyValue[1];
   });

   response.hasError = function() {
         return !response['access_token'];
   };

   return response;
}
```

25. The code presented before creates an object named `response` which might contain an `access_token` or error description. Unfortunately, Facebook returns all the error data as URL query parameters instead of using the fragment as per OAuth 2.0's specifications. At least the object returned by the `getResponse` function can tell if the response has an error.
26. Now let's update the main JavaScript code to the following. The following code extracts the response from the URL fragment, clears the fragment of the URL as a security measure, and in case of an error, just presents a message to the user through a `<div>` HTML tag:

```
$(document).ready(function() {
   var fragment = window.location.hash;
   var res = getResponse(fragment);
   window.location.hash = '_#';
    if (res.hasError()) {
        $("<div>Error trying to obtain user's authorization!</div>").i
        return;
   }
});
```

27. Now let's create the most expected function, which is responsible for using the `access_token` to interact with the Facebook Graph API. Add the following function at the end of the JavaScript code:

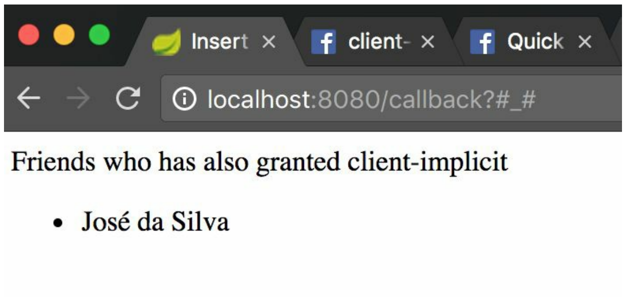
```
function getFriends(accessToken, callback) {
   var baseUrl = 'https://graph.facebook.com/v2.9/';
   var endpoint = 'me/friends';
   var url = baseUrl + endpoint;

   $.ajax({
        url: url,
        beforeSend: function(xhr) {
            xhr.setRequestHeader("Authorization", "Bearer " + accessTo
        },
        success: function(result){
               var friends = result.data;
               callback(friends);
        },
        error: function(jqXHR, textStatus, errorThrown)   {
            console.log(textStatus);
        }
    });
}
```

28. And to finish, just update the main JavaScript code which is using all the declared functions as follows:

```
$(document).ready(function() {
   var fragment = window.location.hash;
   var res = getResponse(fragment);
   window.location.hash = '_#';
   if (res.hasError()) {
       $("<div>Error trying to obtain user's authorization!</div>").ir
       return;
   }
   getFriends(res['access_token'], function(friends) {
       $(friends).each(function(index, friend) {
           $('#friends').find('ul').append('<li>' + friend.name + '</l
       });
   });
});
```

29. Now it's time to run the `client-implicit` application to see the usage of OAuth 2.0 and the Facebook Graph API in practice.

30. Start the application.
31. Go to `http://localhost:8080/` and click on the Authorize button.
32. Grant the requested permissions.
33. When you are redirected back to `client-implicit`, you should see something like the following in your web browser:



Friends who has also granted client-implicit

- José da Silva

34. As you might notice, your application might not retrieve any users yet. That's because Facebook just allows you to present friends who also authorized your application. In our case, another user should be the `client-implicit` user and you have to register her as a tester for your application.

When running on Firefox which version is over 42, you must need to disable Tracking Protection that is a feature provided by Firefox to block content loaded from domains that track users across sites).

How it works...

To start using OAuth 2.0's protected resources, before requesting the `access_token` through the user's grant, we registered the application `client-implicit` through the OAuth 2.0 Provider (Facebook). The responsibility of maintaining the client's data belongs to the Authorization Server. In using Facebook the boundaries between the Authorization Server and the Resource Server is not so clear. The most important thing to understand here is that Facebook is acting as an OAuth 2.0 Provider.

As per the specifications, we have performed the three important steps in client registration process, which was to choose the client type, register the redirection URI, and enter the application's information.

By registering the application, we've received the `client_id` and `client_secret`, but as we are using the Implicit grant flow the `client_secret` won't be needed. That's because this first recipe presents an application which runs directly on the web browser. So, there is no way to safely protect the `client_secret`. When not using the `client_secret` we must try not to expose the received `access_token` that comes with the URL fragment after the user grants permission to access her resources. Another measure to help the application to not expose the access token was to clear the URL fragment.

Another measure that might be applied is to not use any external JavaScript code as such that was used to send usage metrics (in such a way that the `access_token` could be sent to the external service).

After the registration process we got into the code to effectively interact with the Facebook Graph API, which at the time of this writing was version 2.9. Facebook offers two ways to log a user with a valid account:

- By using the Facebook SDK
- By manually building a login flow

To make the OAuth 2.0 usage explicit, this recipe was written by manually building a login flow. So to start the process of user's authentication and authorization, the `client-implicit` application builds the URL for the Authorization Server manually as follows:

```
var request_endpoint = auth_endpoint + "?" +
   "response_type=" + response_type + "&" +
   "client_id=" + client_id + "&" +
   "redirect_uri=" + encodeURI(redirect_uri) + "&" +
   "scope=" + encodeURI(scope);

window.location.href = request_endpoint;
```

After simply redirecting the user to the Authorization Server's endpoint, the flow is transferred to Facebook, which authenticates the user if needed, and the user authorizes whether or not the client application can make use of its resources. Once the user authorizes the client, she is redirected back to the registered redirection URI, which in our case was `http://localhost:8080/callback`.

When receiving the `access_token` all we need to do is extract the token from the URL fragment and start using the Facebook Graph API.

There's more...

As an exercise, you might try to use Facebook SDK, which should be simpler to use for abstracting what we did into the SDK's API. Besides, using the SDK or not, one important thing that should be added to our code is the usage of the state parameter to avoid **Cross Site Request Forgery** (**CSRF**) attacks.

A CSRF attack allows a malicious user to execute operations in the name of another user (a victim). Regarding web applications, a valid approach to avoid CSRF is to make the client send a variable to the server with some random string which might be checked after receiving it back from the server's response, so the first value and the second (received) must be the same.

Regarding security issues, one other valuable suggestion is to send the `access_token` to the server side so you don't have to request a new access token on every web page of your application (but take care with the expiration time).

See also

- Preparing the environment
- Reading the user's contacts from Facebook on the server side

Reading the user's contacts from Facebook on the server side

Now you are perfectly familiarized with the Facebook login process and Graph API usage. But to allow for a safer approach to get user authorization to retrieve contacts (or friends) from Facebook, this chapter presents how to use the server side approach which maps directly to the Authorization Code grant type from the OAuth 2.0 specifications.

Getting ready

For this recipe, we need to create a simple web application in the same way we did for `client-implicit`. As we will develop an application which interacts with Facebook at the server side, we are supposed to write a lot of code. But instead of writing too much code, let's use the Spring Social Facebook project.

There is an important step to perform, similar to what we did for `client-implicit`; as the application is a Facebook client we need to register a new application.

How to do it...

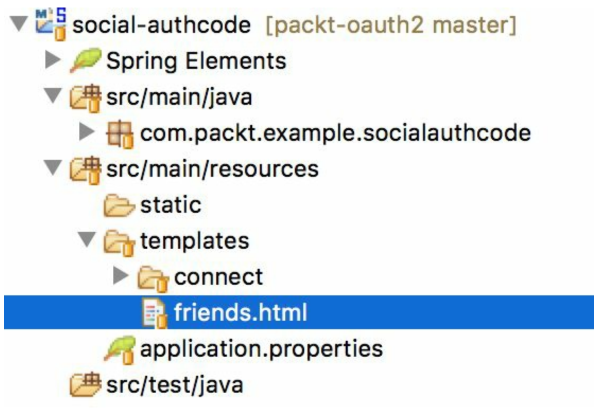
Follow the steps below to create a client application to integrate with Facebook using the server-side flow from OAuth 2.0:

1. Go to https://developers.facebook.com/apps/ and add a new application by clicking on Add a New App.
2. Register a new client application on Facebook with the Display Name `social-authcode`.
3. You will be guided to select one Facebook product. So, choose Facebook Login by clicking on Set Up and then choose Web as a platform.
4. You will be asked to enter the site URL, which might be `http://socialauthcode.test/`.
5. After creating the application on Facebook, click on Facebook Login on the left panel to configure a valid redirect URI, which should be `http://localhost:8080/connect/facebook`.
6. Click on Dashboard on the left panel so you can retrieve the App ID and App Secret which map to `client_id` and `client_secret`, as you may already know, and grab the credentials to use later when implementing the client application.
7. Now let's create the initial project using Spring Initializr, as we did for other recipes in this book. Go to https://start.spring.io/ and define the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `social-authcode`
   - Add `Web` and `Thymeleaf` as the dependencies for this project
8. Import the project to your IDE. When using Eclipse, just import it as a Maven project.
9. Now add the following dependencies into the `pom.xml` file to add support for Spring Social Facebook:

```
<dependency>
```

```
        <groupId>org.springframework.social</groupId>
        <artifactId>spring-social-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.social</groupId>
        <artifactId>spring-social-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.social</groupId>
        <artifactId>spring-social-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.social</groupId>
        <artifactId>spring-social-facebook</artifactId>
    </dependency>
```

10. Create an HTML file named `friends.html` inside the templates directory located within `src/main/resources`, as follows:



11. Open the file `friends.html` and add the following content:

```
    <!DOCTYPE html>
    <html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Friends</title>
    </head>
    <body>
        <h3>Hello, <span th:text="${facebookProfile.name}">User</span>!</h3
        <h4>Your friends which also allowed social-authcode:</h4>
        <div th:each="friend:${friends}">
                <b th:text="${friend.id}">[id]</b> - <b th:text="${friend.nam
                <hr/>
        </div>
    </body>
    </html>
```

12. Now, we need URL mapping so the previous HTML file can be rendered. To do this, create a new Java class named

FriendsController.java inside the package com.packt.example.socialauthcode, with the following content:

```
@Controller @RequestMapping("/")
public class FriendsController {
    @GetMapping
    public String friends(Model model) { return "friends"; }
}
```

13. As you may realize, the template friends.html relies on an object named facebookProfile and another named friends. The object facebookProfile must have the name attribute and the friends object must be a list of objects which have id and name properties. The good news is that we don't have to declare classes for these objects because **Spring Social** already provides them. We just need to have a valid user connection to start using these objects, so add the following attributes inside the FriendsController class:

```
@Autowired
private Facebook facebook;
@Autowired
private ConnectionRepository connectionRepository;
```

14. With the class ConnectionRepository we can save or retrieve user connections with any provider (only Facebook matters now). Let's take advantage of this class to know if there is any user connected with Facebook and if not, we must redirect the user so she can authorize the social-authcode of the application to retrieve protected resources (her friends). Replace the code from friends method of the FriendsController class with the code presented in the following code:

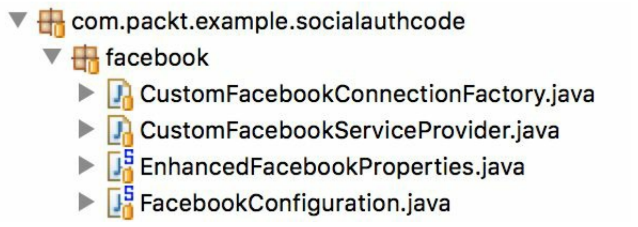
```
@GetMapping
public String friends(Model model) {
    if (connectionRepository.findPrimaryConnection(Facebook.class) ==
        return "redirect:/connect/facebook";
    }
    return "friends";
}
```

15. Now, add the following source code after the if block, that checks for a connection. This new block of code will be executed when there is a user connected to Facebook (when importing User and Reference classes, make sure to import from org.springframework.social.facebook.api

package):

```
String [] fields = { "id", "email", "name" };
User userProfile = facebook.fetchObject("me", User.class, fields);

model.addAttribute("facebookProfile", userProfile);
PagedList<Reference> friends = facebook.friendOperations().getFriends(
model.addAttribute("friends", friends);
```

16. Although this short method executes all that's needed to retrieve the user's profile and contacts using the Authorization Code grant type, you must create some configuration classes. To better group the configuration classes, create a new package called `facebook` inside `com.packt.example.socialauthcode`, which will accommodate the following classes:



```
com.packt.example.socialauthcode
  facebook
    CustomFacebookConnectionFactory.java
    CustomFacebookServiceProvider.java
    EnhancedFacebookProperties.java
    FacebookConfiguration.java
```

17. Create the class `EnhancedFacebookProperties`, as presented in the following code, inside the inner package `facebook`, so we can configure the application properties as `client_id` and `client_secret` (don't forget to create the respective getters and setters for each attribute):

```
@Component
@ConfigurationProperties(prefix = "facebook")
public class EnhancedFacebookProperties {
   private String appId;
   private String appSecret;
   private String apiVersion;
   // getters and setters omitted for brevity
}
```

18. Before continuing creating the other classes, you must configure the `appSecret` and `apiVersion` values so the application `social-authcode` is able to request an `access_token`. As you may realize, the class `EnhancedFacebookProperties` is annotated with `@ConfigurationProperties` which allows for defining the properties inside the `application.properties` file, as follows:

```
facebook.app-id=1948923582021549
facebook.app-secret=1b4b0f882b185094a903e76a661c7c7c
facebook.api-version=2.9
```

19. Now create the class `CustomFacebookServiceProvider`, as follows. This class is responsible for creating a custom instance of `OAuth2Template` allowing us to effectively configure the Facebook API version which at the time of this writing was 2.9:

```
public class CustomFacebookServiceProvider extends
        AbstractOAuth2ServiceProvider<Facebook> {

   private String appNamespace;
   private String apiVersion;

   public CustomFacebookServiceProvider(
         String appId, String appSecret, String apiVersion) {
         super(getOAuth2Template(appId, appSecret, apiVersion));
         this.apiVersion = apiVersion;
   }

   private static OAuth2Template getOAuth2Template(
         String appId, String appSecret, String apiVersion) {
         String graphApiURL =
               "https://graph.facebook.com/v" + apiVersion + "/";

         OAuth2Template template = new OAuth2Template(
               appId, appSecret, "https://www.facebook.com/v" + apiVer

         template.setUseParametersForClientAuthentication(true);
         return template;
   }

   @Override
   public Facebook getApi(String accessToken) {
         FacebookTemplate template = new FacebookTemplate(
               accessToken, appNamespace);
         template.setApiVersion(apiVersion);
         return template;
   }

}
```

20. So that the `CustomFacebookServiceProvider` can be properly created, create the class `CustomFacebookConnectionFactory` as presented in the following code:

```
public class CustomFacebookConnectionFactory extends
   OAuth2ConnectionFactory<Facebook> {
   public CustomFacebookConnectionFactory(String appId, String appSecr
      super("facebook",
           new CustomFacebookServiceProvider(appId, appSecret, apiVers
           new FacebookAdapter());
   }
}
```

21. And finally create the class `FacebookConfiguration` with the following content:

```
@Configuration @EnableSocial
@EnableConfigurationProperties(FacebookProperties.class)
public class FacebookConfiguration extends SocialAutoConfigurerAdapter
   @Autowired
   private EnhancedFacebookProperties properties;

   @Override
   protected ConnectionFactory<?> createConnectionFactory() {
         return new CustomFacebookConnectionFactory(this.properties.ge
             this.properties.getAppSecret(), this.properties.getApiVer
   }
}
```

22. If you look at the content of `FriendsController`, you should see that this class is using an instance of `Facebook` which provides the API to interact with Facebook Graph API. The instance of `Facebook` must be created through a Spring bean declared as follows inside the `FacebookConfiguration` (When importing the `Connection` class, make sure you import from `org.springframework.social.connect` package):

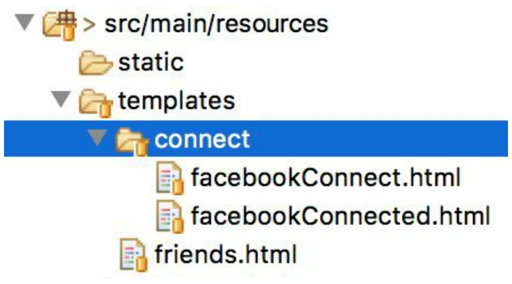
```
@Bean
@ConditionalOnMissingBean(Facebook.class)
@Scope(value = "request", proxyMode = ScopedProxyMode.INTERFACES)
public Facebook facebook(ConnectionRepository repository) {
   Connection<Facebook> connection = repository
         .findPrimaryConnection(Facebook.class);
   return connection != null ? connection.getApi() : null;
}
```

23. As we are using Spring Social, most redirection will be handled by the `ConnectController` class which is declared by Spring Social. But how does Spring Social know to build the redirect URI? We have not provided the application's domain. By default, Spring Social uses request data to build the redirect URL automatically. But as the application might be deployed behind a proxy, the provider won't be capable of redirecting the user back to the callback URL defined inside `ConnectController`. To overcome this issue, declare the following method inside the `FacebookConfiguration` class:

```
@Bean
```

```
        public ConnectController connectController(
                ConnectionFactoryLocator factoryLocator,
                ConnectionRepository repository) {
            ConnectController controller = new ConnectController(factoryLocator
            controller.setApplicationUrl("http://localhost:8080");
            return controller;
        }
```

24. This controller provides all we need to handle OAuth 2.0's authorization flow. It also allows the rendering of two views which by default are named `{provider}Connect` and `{provider}Connected` where the provider in this case is `facebook`. To satisfy both views, create the following HTML files inside the folder `templates/connect` within the `src/main/resources` project's directory, as follows:



25. Now add the following content to `facebookConnect.html`:

```
        <html>
        <head>
            <title>Social Authcode</title>
        </head>
        <body>
            <h2>Connect to Facebook to see your contacts</h2>

            <form action="/connect/facebook" method="POST">
                  <input type="hidden" name="scope"
                        value="public_profile user_friends" />
                  <input type="hidden" name="response_type"
                        value="code" />
                   <div class="formInfo">
                        Click the button to share your contacts
                        with <b>social-authcode</b>
                  </div>
                  <p><button type="submit">Connect to Facebook</button></p>
            </form>

        </body>
        </html>
```

26. And now add the following content to `facebookConnected.html`:

```
<html>
   <head><title>Social Authcode</title></head>
   <body>
         <h2>Connected to Facebook</h2>
         <p>Click <a href="/">here</a> to see your friends.</p>
   </body>
</html>
```

27. That's it. Now you can start the application by running the class `SocialAuthcodeApplication`.

How it works...

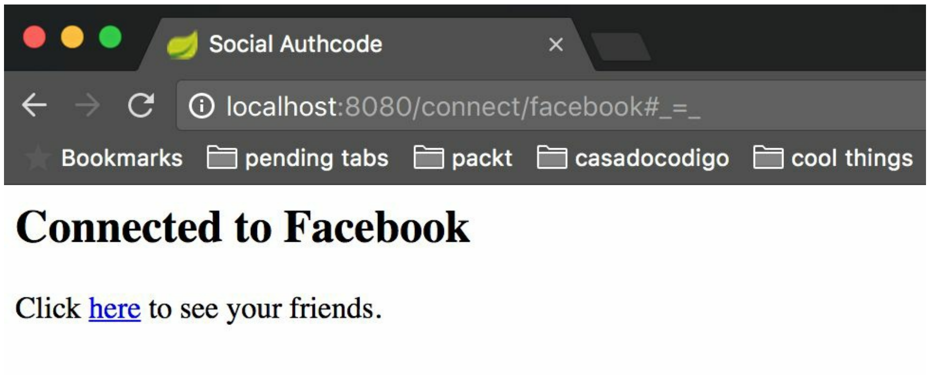
This chapter presented you with how to register your application and how to connect with Facebook through the use of the Authorization Code grant type. Because it's a server side flow, it is supposed to be more secure than using the client-side approach (that is, to use the Implicit grant type). But instead of writing the code to handle all the conversations between `social-authcode` and Facebook (the OAuth 2.0 dance) we are using Spring Social, which provides the `ConnectController` class which has the capability of starting the authorization flow as well as receiving all callbacks that must be mapped when registering the application.

To better understand how this application works, run the class `SocialAuthcodeApplication` as Java code and go to `http://localhost:8080/` to see the page that will present you with the possibility of connecting to Facebook. Click on Connect to Facebook and you will be redirected to the Facebook authentication page (as per OAuth 2.0's specifications).

After authenticating the user, Facebook presents the user consent page presenting the scope the client application is asking for. Click on continue to grant the requested permission.

After granting permission for `public_profile` and `friend_list` scopes, the user must be redirected back to `localhost:8080/connect` with the authorization code embedded (which will be extracted and validated by `ConnectController` automatically).

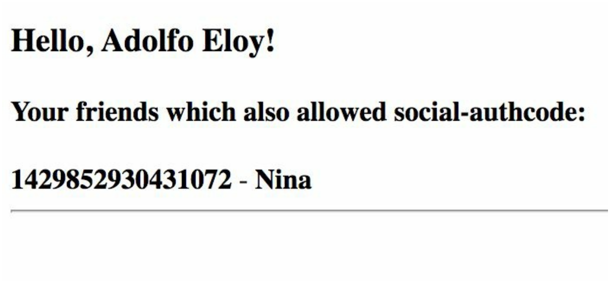
Note that `ConnectController` will render the `facebookConnected` view by presenting the following page:



Connected to Facebook

Click here to see your friends.

Click on the link here so the application can retrieve the friends which have also authorized `social-authcode`. You are supposed to see the following page with different content:



Hello, Adolfo Eloy!

Your friends which also allowed social-authcode:

1429852930431072 - Nina

There's more...

When registering the application on Facebook, we also configured the redirect URI to be `http://localhost:8080/connect`. Why not use `http://localhost:8080/callback`? By using the `/connect` endpoint, we take advantage of the endpoints defined by `ConnectController`. If you do not want to use Spring Social, you are supposed to validate the authorization code through the use of state parameters by yourself. When using Spring Social, we also take advantage of callbacks which are particular to the Facebook provider as **De-authorize Callback URL's** which might be set up in the settings from the Facebook Login product.

Even though we are using Spring Social Facebook we are still creating some classes that are also provided by Spring Social Facebook. As you could realize, the name of some classes begin with `Custom`. That's because we can customize how to create an instance of `OAuth2Template` as well as the `FacebookTemplate` class. It's important because the version supported at the time of this writing was 2.5, which was to be deprecated soon, and that's the version defined inside the Facebook provider for Spring Social.

There is an important thing to be mentioned about the interactions between the client and the OAuth 2.0 Provider, which in this case is Facebook. As you may realize, we are registering the redirect URI without using TLS/SSL. The URI we've registered is HTTP instead of HTTPS. All the recipes in this book are using such an approach just to ease the creation of the examples. Be sure to use HTTPS in production to protect the integrity and confidentiality of data transferred between your application and any other provider.

Another valuable improvement which might be done is to use a **Relational Database Management System** (**RDBMS**) to persist connections with providers. As the application does not explicitly define the strategy for connection persistence, Spring Social provides the in-memory version, so whenever you restart your server the user's connections will be lost. If you want to try using a database, you might declare a bean of type `JdbcUsersConnectionRepository`, and create the following table within the database of your choice:

```
create table UserConnection (userId varchar(255) not null,
    providerId varchar(255) not null,
    providerUserId varchar(255),
    rank int not null,
    displayName varchar(255),
    profileUrl varchar(512),
    imageUrl varchar(512),
    accessToken varchar(512) not null,
    secret varchar(512),
    refreshToken varchar(512),
    expireTime bigint,
    primary key (userId, providerId, providerUserId));
create unique index UserConnectionRank on UserConnection(userId, providerId,
```

To get more details about this, look at Spring Social's official documentation at http://docs.spring.io/spring-social/docs/1.1.4.RELEASE/reference/htmlsingle/#section_establishingConnections.

See also

- Preparing the environment
- Reading the user's contacts from Facebook on the client side
- Accessing OAuth 2.0 Google protected resources bound to user's session

Accessing OAuth 2.0 LinkedIn protected resources

This recipe presents you with how to retrieve a LinkedIn user's profile through the user's authorization using OAuth 2.0 and Spring Social to abstract all Authorization Code grant type from the OAuth 2.0 protocol.

Getting ready

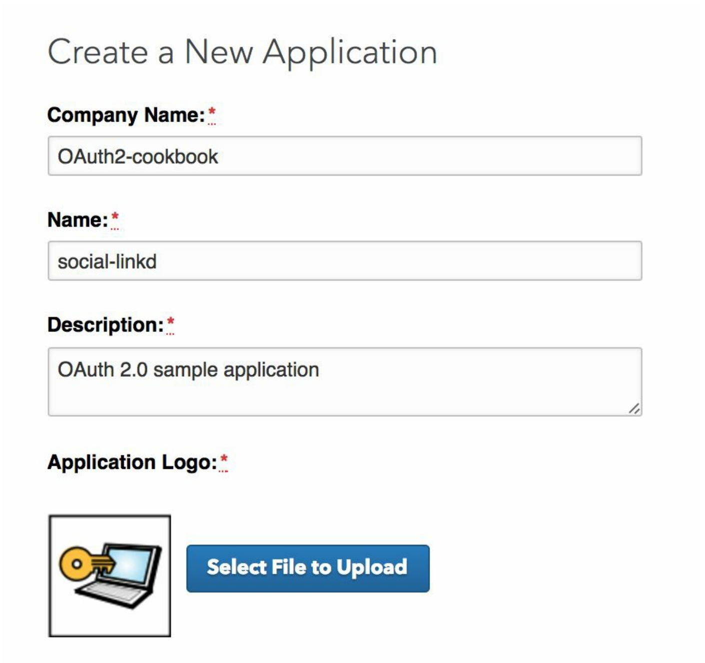
To run this recipe, create a simple web application using Spring Boot which will help the development of the application. To abstract and ease the OAuth 2.0 grant type implementation, and to help in using the LinkedIn API, this recipe also relies on the `spring-social-linkedin` project.

How to do it...

As described by OAuth 2.0's protocol, the client application must be registered at the Authorization Server which in this case is LinkedIn:

1. So to satisfy this condition, the first thing to do is to register the application on LinkedIn by accessing https://www.linkedin.com/developer/apps/.

2. When accessing the previous URL, click on Create Application and you will be redirected to the following page which will ask you for basic information about the application being created:



Create a New Application

Company Name:* OAuth2-cookbook

Name:* social-linkd

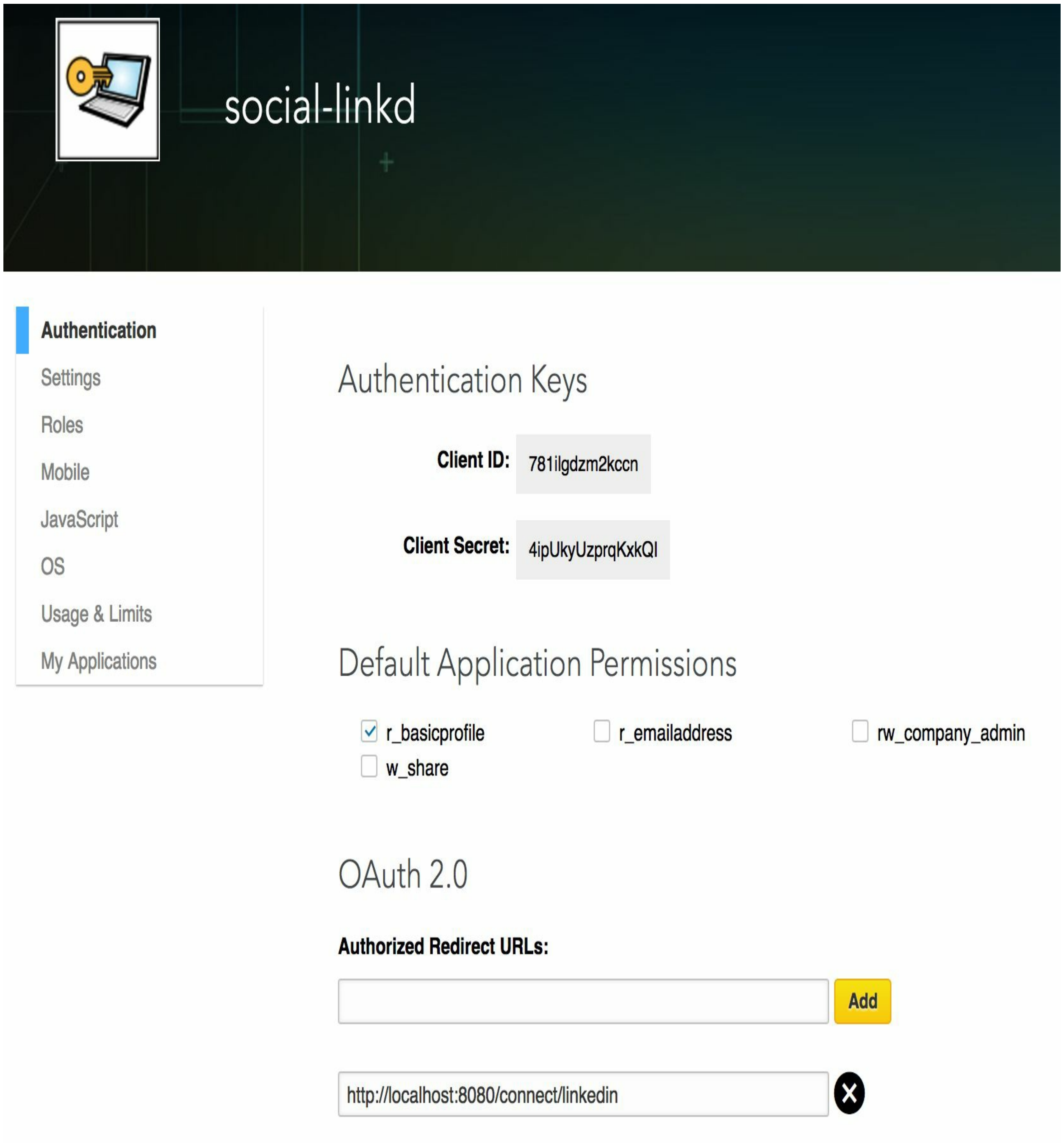
Description:* OAuth 2.0 sample application

Application Logo:*

3. Unlike many other OAuth 2.0 Providers, LinkedIn requires an application logo as you might have seen in the previous image. LinkedIn also asks for more business data such as the website URL, business email, and business phone.

4. Fill out the form and click on the Submit button. You will be redirected to the application's dashboard as shown in the following screenshot

which presents you with the Authentication Keys and the field to define the redirection URL:



Authentication Keys

Client ID: 781ilgdzm2kccn

Client Secret: 4ipUkyUzprqKxkQl

Default Application Permissions

[✓] r_basicprofile [ ] r_emailaddress [ ] rw_company_admin [ ] w_share

OAuth 2.0

Authorized Redirect URLs:

http://localhost:8080/connect/linkedin

5. As we are using Spring Social, let's add a Redirect URL which follows the pattern regarding the endpoint which was defined as `connect/linkedin`. After entering the Redirect URL, click on the Add and then click on Update button.

6. Now, make sure to grab the Authorization Keys (that is, `client_id` and `client_secret`) to use in the application that we will create in the next step.

7. Create the initial project using Spring Initializr as we did for the other recipes in this book. Go to https://start.spring.io/ and define the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `social-linkd` (you can use different names if you prefer, but do not forget to change all the references for `linkd` that were used throughout this recipe)
   - Add `Web` and `Thymeleaf` as the dependencies for this project
8. Import the project to your IDE (if using Eclipse, import as a Maven Project).
9. Add the following dependency to the `pom.xml` file:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-social-linkedin</artifactId>
</dependency>
```

10. For this recipe, the Spring Social provider implementation already provides a well-defined auto configuration support for Spring Boot. So, it's easier to create the application and just having to worry about the client credential settings. Open the `application.properties` file and add the following content (using the credentials generated for your application):

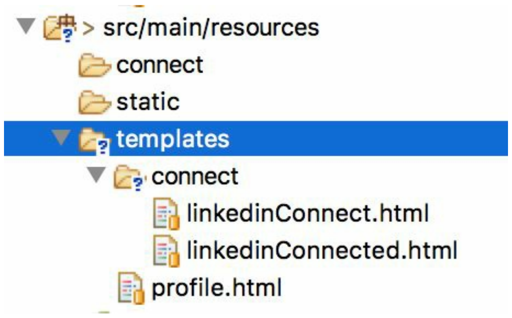
```
spring.social.linkedin.app-id=77a1bnosz2wdm8
spring.social.linkedin.app-secret=STHgwbfPSg0Hy8bO
```

11. Now create the controller class `ProfileController` which has the responsibility of retrieving the user's profile through the use of the LinkedIn API. This class should be created within the package `com.packt.linkedin.example.sociallinkd`.
12. Make sure the class `ProfileController` looks like the following:

```
@Controller
public class ProfileController {
   @Autowired
   private LinkedIn linkedin;
   @Autowired
   private ConnectionRepository connectionRepository;
```

```
        @GetMapping
        public String profile(Model model) {
            if (connectionRepository.findPrimaryConnection(LinkedIn.class)
                  return "redirect:/connect/linkedin";
            }
            String firstName = linkedin.profileOperations()
                .getUserProfile().getFirstName();
            model.addAttribute("name", firstName);
            return "profile";
        }
    }
```

13. As you might expect, the application will be able to retrieve the user's profile only when the user has made the connection between LinkedIn and the `social-linkd` client application.
14. So, if there is no connection available, the user will be redirected to `/connect/linkedin` which is mapped by the `ConnectController` class from Spring Social. Such an endpoint will redirect the user to the view, defined by the name `linkedinConnect` which maps directly to the `linkedinConnect.html` file that might be created under `templates/connect` directory, located within the `src/main/resources` project directory as follows:



15. Looking at the previous screenshot, you can see that there is also `linkedinConnected.html`, which will be presented when a user's connection is available for the `social-linkd` application.
16. All the logic to decide when to present `linkedinConnect.html` or `linkedinConnected.html` is defined inside the method `connectionStatus` from the `ConnectController` class. The main logic is defined as presented in the following code:

```
    if (connections.isEmpty()) {
       return connectView(providerId);
    } else {
       model.addAttribute("connections", connections);
```

```
        return connectedView(providerId);
    }
```

17. Add the following HTML content to `linkedinConnect.html`:

```
<html>
<head><title>Social LinkedIn</title></head>
<body>
   <h2>Connect to LinkedIn to see your profile</h2>
    <form action="/connect/linkedin" method="POST">
     <input type="hidden" name="scope" value="r_basicprofile" />
      <div class="formInfo">
       Click the button to share your profile with
       <b>social-linkedin</b>
     </div>
     <p><button type="submit">Connect to LinkedIn</button></p>
   </form>
</body>
</html>
```

18. Now add the following HTML content to `linkedinConnected.html`:

```
<html>
   <head>
     <title>Social LinkedIn</title>
   </head>
   <body>
     <h2>Connected to LinkedIn</h2>
     <p>Click <a href="/">here</a> to see your profile.</p>
   </body>
</html>
```

19. To present the user's profile, create the file `profile.html` inside the templates directory with the following content:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
   <title>LinkedIn integration</title>
</head>
<body>
   <h3>Hello, <span th:text="${name}">User</span>!</h3>
   <br/>
</body>
</html>
```

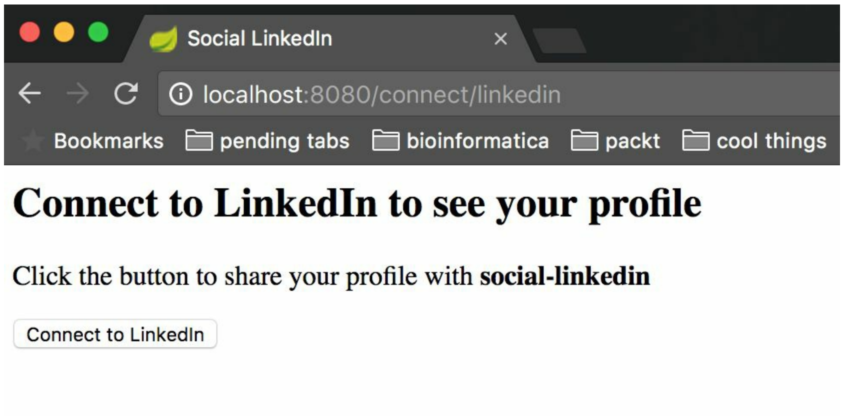
20. Now that everything is perfectly configured, start the application and go to `http://localhost:8080` to start running the authorization flow.

How it works...

This recipe presented you with how you can create an application that interacts with LinkedIn to retrieve the user's profile using OAuth 2.0 protocol. This recipe relies on Spring Social Provider for LinkedIn, which saves us from having to create a controller to deal with OAuth 2.0 callbacks as well as building URLs for authorization and token requests. This recipe differs from other recipes using Spring Social because it presents one provider implementation which support Spring Boot's auto-configuration feature, so we don't need to create any configuration classes.

Besides the fact that a lot of OAuth 2.0's details are abstracted behind Spring Social, all the steps happen when we run the application and start the authorization flow. In fact, as we are using the Authorization Code grant type, the application retrieves the access token through two steps, which are authorization and token request.

To start the authorization flow you must go to `http://localhost:8080/` which, in case of being not connected the user's LinkedIn account with the `social-linkd` application, should be redirected to `/connect/linkedin`:



Connect to LinkedIn to see your profile

Click the button to share your profile with **social-linkedin**

The redirection is performed by the method `profile` from the `ProfileController` class. As the controller and this method do not define any paths for a request, it will be defined as `/` by default. As you may notice in the following code, the first thing the method repositories do is to check if the current user has connected her account with the application, which is `social-linkedin`:

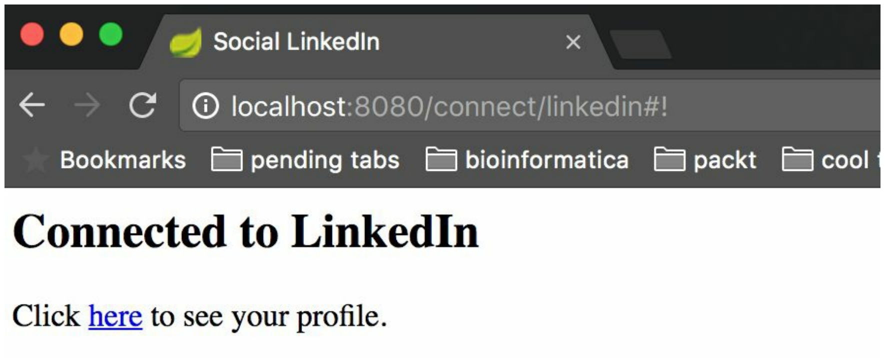
```
if (connectionRepository.findPrimaryConnection(LinkedIn.class) == null) {
    return "redirect:/connect/linkedin";
}
```

The endpoint `/connect/linkedin` maps directly to the method `connectionStatus` from the `ConnectController` class of Spring Social. If there is no connection, this method calls the private method `connectView` which builds the name `{providerId}Connect`, which in LinkedIn's case is `linkedinConnect`. This is exactly the name of the view we created as `linkedinConnect.html`.

Open the file `linkedinConnect.html` to see which scope the application is asking for LinkedIn, and you must realize that it is `r_basicprofile`. All the available scopes defined by LinkedIn should be retrieved by accessing the application dashboard which is present in the section Default Application Permissions.

Back to the page generated by `linkedinConnect` view, if you click on the Connect to LinkedIn button, you will be redirected to LinkedIn, which will ask you for your credentials and for your consent.

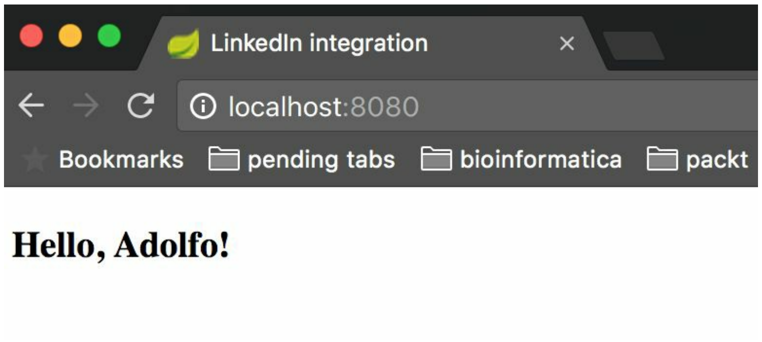
Notice that LinkedIn, unlike many other OAuth 2.0 Providers, asks for permission at the same time it authenticates the user. If you click on Allow Access and send your credentials at the authentication form, `social-linkd` will receive the authorization code and will use it to retrieve an access token and create the connection for the current user within the application. Then, if there is a connection, the private method `connectedView` from `ConnectController` will be called, which will render the following HTML page defined by `linkedinConnected.html`:



Connected to LinkedIn

Click here to see your profile.

Clicking on the link here, you will then be redirected to the main page, where your profile name will be presented as follows:



Hello, Adolfo!

There's more...

For this recipe, we didn't create any configuration class. But if you want to define the base URL for callback redirection that happens when using the OAuth 2.0 protocol, you need to create a configuration class to define a custom `ConnectController` bean, as presented in the following code:

```
@Configuration
public class LinkedInConfiguration {
    @Bean
    public ConnectController connectController(ConnectionFactoryLocator locat
        ConnectionRepository repository) {
        ConnectController controller = new ConnectController(locator, reposit
        controller.setApplicationUrl("http://localhost:8080");
        return controller;
    }
}
```

By doing such configurations, you avoid issues when the application runs behind a proxy. The redirect URI will be automatically generated using the request info which will be based on the application which might be running behind a proxy. This way, the OAuth 2.0 Provider won't be able to redirect to the right callback URL because it will be hidden by the proxy. The configuration presented previously allows you to define the proxy's URL.

Do not forget to define the same redirect URL at the OAuth 2.0 Provider and make all the communication through TLS/SSL.

See also

- Preparing the environment
- Reading the user's contacts from Facebook on the server side
- Accessing OAuth 2.0 Google protected resources bound to the user's session

Accessing OAuth 2.0 Google protected resources bound to the user's session

This recipe presents you with how to retrieve a user's profile from Google Plus, through the user's Google account. This recipe relies on Spring Social to abstract the authorization and the usage of the Google Plus API and tightens the user's Google connection with Spring Security to allow managing connections per logged user.

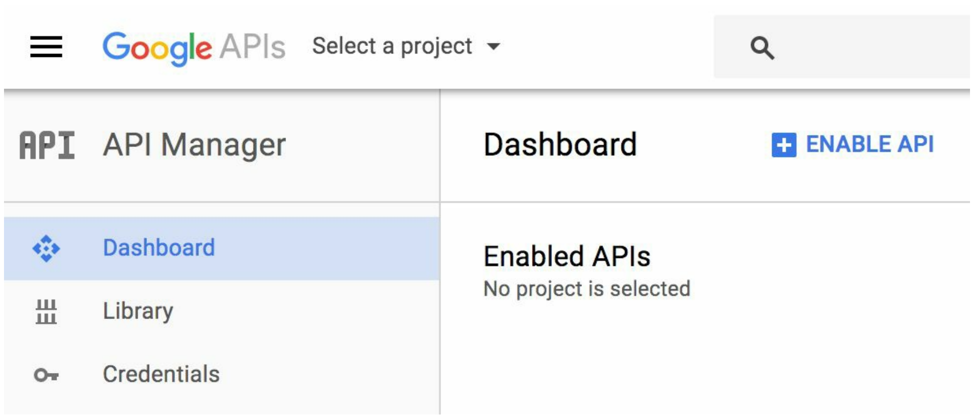
Getting ready

To run this recipe, create a web application using Spring Boot which will help with the development of the application. To abstract and ease the OAuth 2.0 grant type implementation and help with using Google Plus's API, this recipe also relies on the `spring-social-google` project and `spring-security` project.

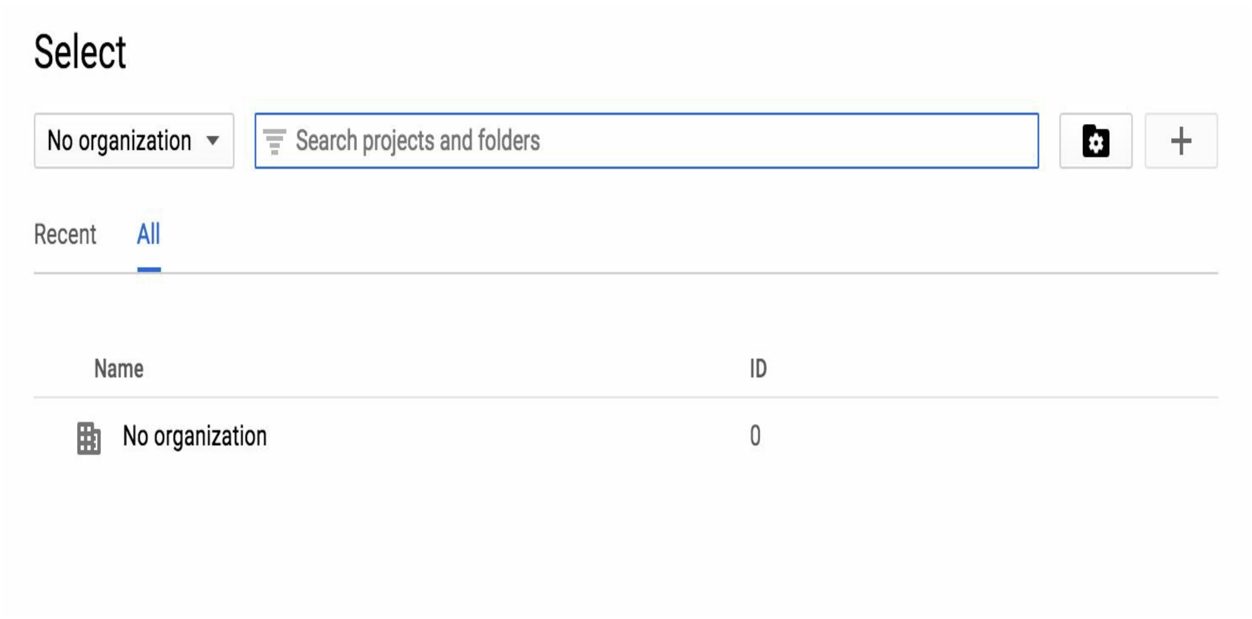
How to do it...

Follow the steps that basically present you with how to register your application with Google and how to interact with the Google Plus API through the use of the Spring Social Google provider with Spring Security.

1. Go to the Google Developers Console located at https://console.developers.google.com to start registering the application.
2. You should see the following screen if you still do not have any project created:

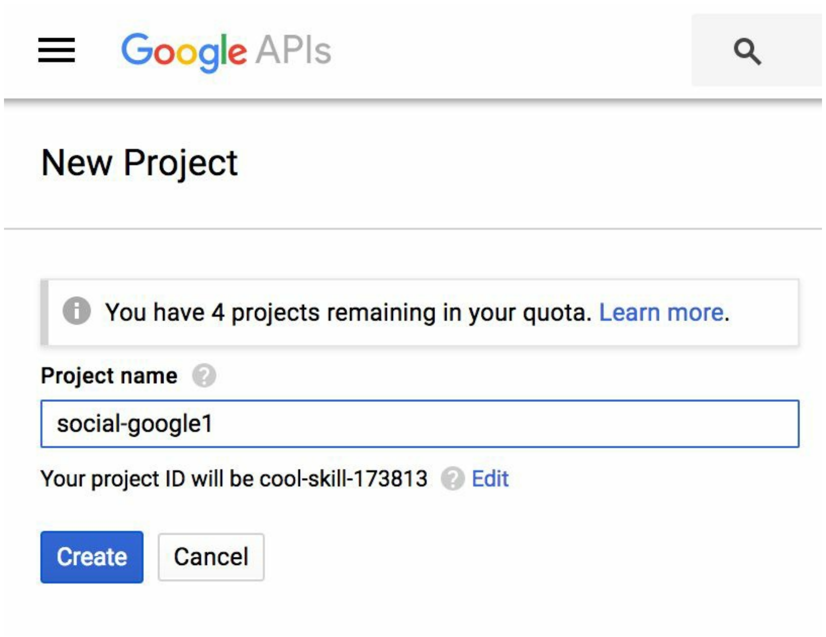


3. To create a new application, click on Select a project and you will see the following screen:



Name	ID
No organization	0

4. Therefore, click on the + button to start registering your application and you will see the following interface:

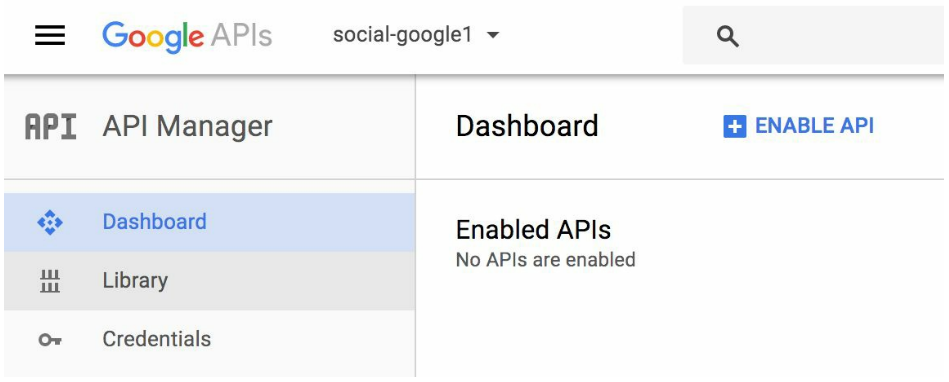


You have 4 projects remaining in your quota. Learn more.

Project name: social-google1

Your project ID will be cool-skill-173813 Edit

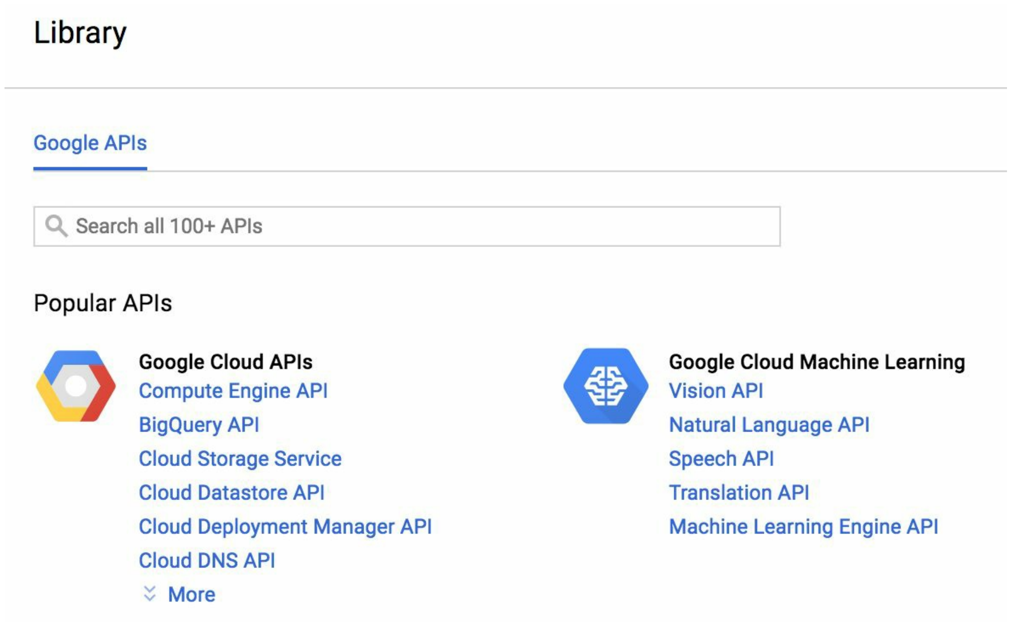
5. Define the name of your project. I have defined the name `social-google1` but you can use any of your preference (but don't forget to change all references to this name throughout the recipe). Then, after setting the name, just click on Create and you will be redirected to the dashboard of your new application as presented in the following screenshot:



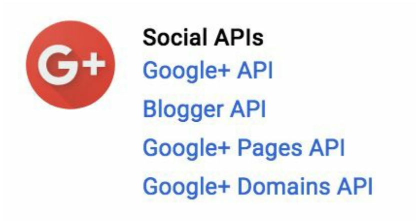
Enabled APIs

No APIs are enabled

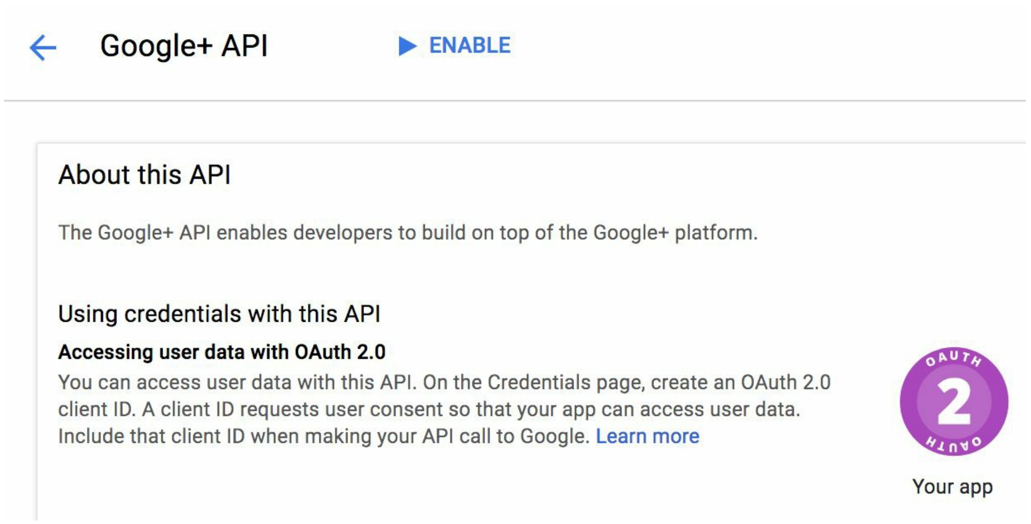
6. Now, to be able to retrieve a user's profile, you must enable an API which in the case of this recipe will be the Google Plus API. Click on the ENABLE API link or on Library at the left side of the dashboard presented in the previous screenshot to see the following Google API portfolio:



7. Then select Google+ API by searching through the Search all input text, or by clicking on the link presented in the Libraries page as follows:



8. Click on the ENABLE link at the top of the page as presented in the following screenshot:



Google+ API

ENABLE

About this API

The Google+ API enables developers to build on top of the Google+ platform.

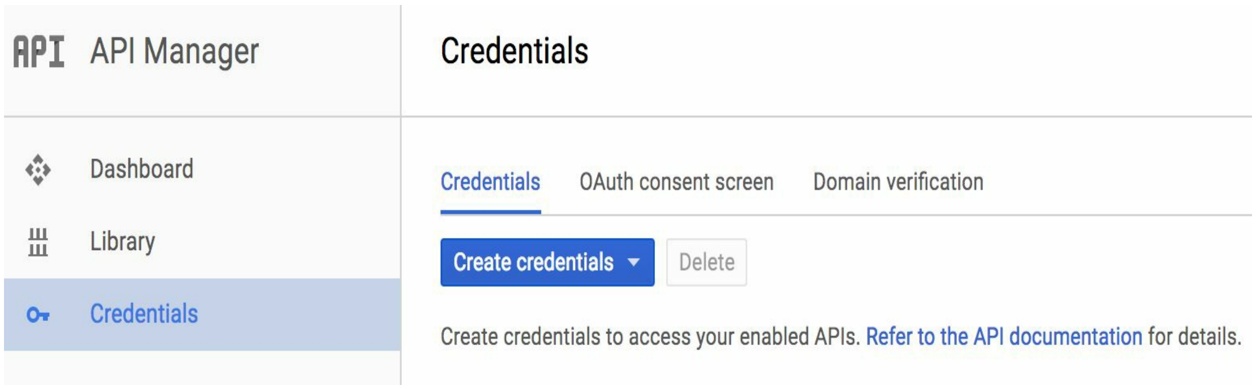
Using credentials with this API

Accessing user data with OAuth 2.0

You can access user data with this API. On the Credentials page, create an OAuth 2.0 client ID. A client ID requests user consent so that your app can access user data. Include that client ID when making your API call to Google. Learn more

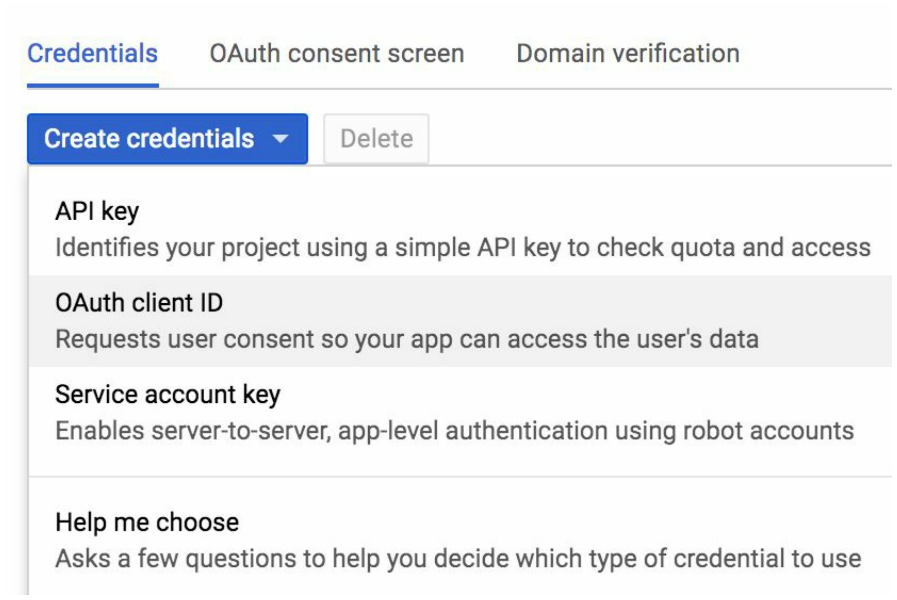
Your app

9. After enabling the API, you need to create OAuth 2.0 credentials to be able to interact with the Google Plus API, so you must click on Credentials at the left side of the panel:



Create credentials to access your enabled APIs. Refer to the API documentation for details.

10. Click on Create credentials and select the OAuth client ID as follows:



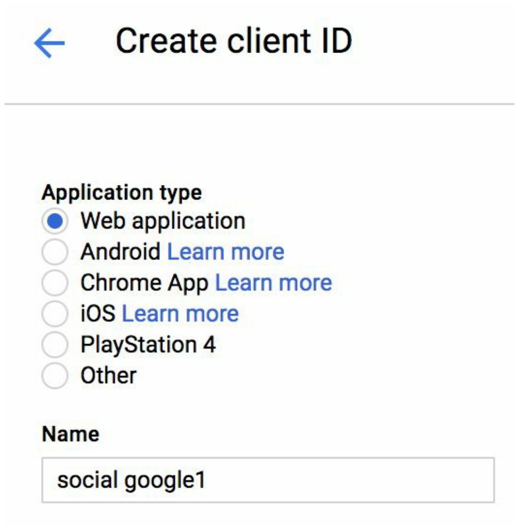
API key
Identifies your project using a simple API key to check quota and access

OAuth client ID
Requests user consent so your app can access the user's data

Service account key
Enables server-to-server, app-level authentication using robot accounts

Help me choose
Asks a few questions to help you decide which type of credential to use

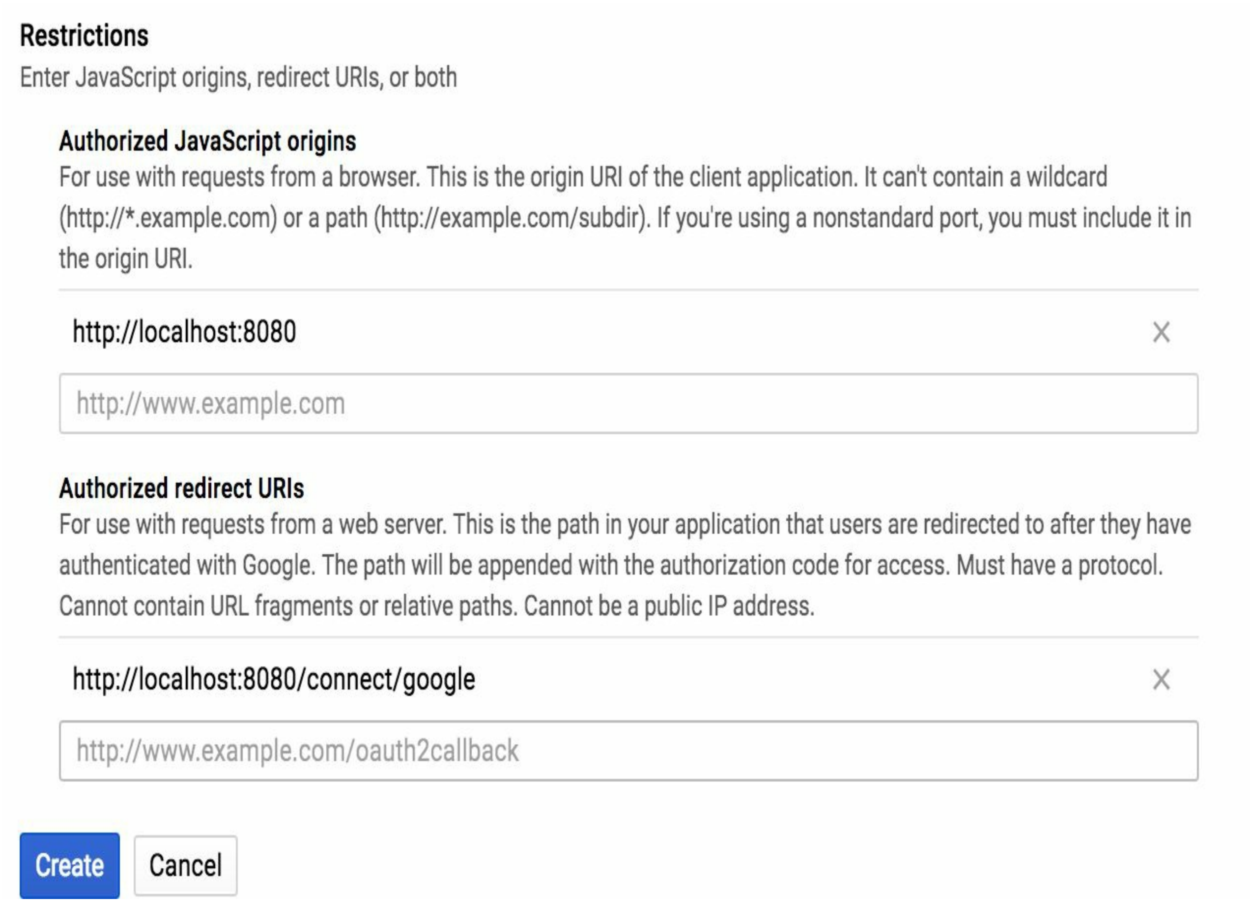
11. Select the Application Type which must be Web application for this recipe:



Application type
● Web application
○ Android Learn more
○ Chrome App Learn more
○ iOS Learn more
○ PlayStation 4
○ Other

Name
social google1

12. Then enter the URL settings for JavaScript origins and Authorized redirect URIs, as presented in the following screenshot:



Restrictions

Enter JavaScript origins, redirect URIs, or both

Authorized JavaScript origins

For use with requests from a browser. This is the origin URI of the client application. It can't contain a wildcard (http://*.example.com) or a path (http://example.com/subdir). If you're using a nonstandard port, you must include it in the origin URI.

http://localhost:8080

http://www.example.com

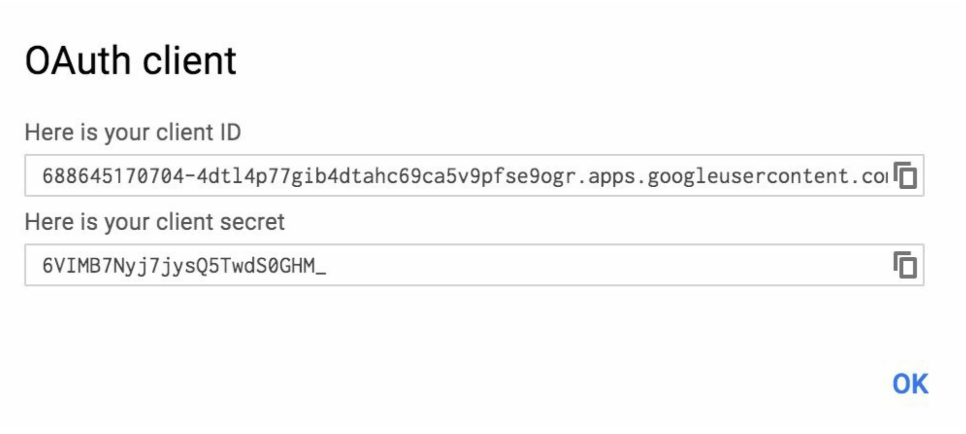
Authorized redirect URIs

For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorization code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

http://localhost:8080/connect/google

http://www.example.com/oauth2callback

13. Click on Create and grab the client ID and client secret that will be prompted by Google as follows:



OAuth client

Here is your client ID

688645170704-4dtl4p77gib4dtahc69ca5v9pfse9ogr.apps.googleusercontent.co

Here is your client secret

6VIMB7Nyj7jysQ5TwdS0GHM_

14. Now, you have all that's needed to create a `social-google1` project.

15. Create the initial project using Spring Initializr as we did for other recipes in this book. Go to https://start.spring.io/ and define the following

data:
- Set up the group as `com.packt.example`.
- Define the Artifact as `social-google1` (you can use different names if you prefer, but do not forgot to change all the references to `social-google1` used throughout this recipe).
- Add `Web`, `Thymeleaf` and `Security` as the dependencies for this project.

16. Import the project to your IDE (if using Eclipse, import as a Maven Project).
17. Open the `pom.xml` file and add the following dependencies:

```
<dependency>
   <groupId>org.springframework.social</groupId>
   <artifactId>spring-social-google</artifactId>
   <version>1.0.0.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.springframework.social</groupId>
   <artifactId>spring-social-security</artifactId>
</dependency>
```

18. Create the class `GoogleProperties` within the package `com.packt.example.socialgoogle1.config` and add the following content:

```
@ConfigurationProperties("spring.social.google")
public class GoogleProperties {
   private String appId;
   private String appSecret;

   public String getAppId() { return this.appId; }
   public void setAppId(String appId) { this.appId = appId; }
   public String getAppSecret() { return this.appSecret; }
   public void setAppSecret(String appSecret) { this.appSecret = appSe
}
```

19. Now add the respective attributes within the file `application.properties` which maps to the attributes defined in the `GoogleProperties` class (change the credentials to those received when registering the application):

```
spring.social.google.appId=688645170704
spring.social.google.appSecret=OIRTUxhs
```

20. Create the class `GoogleConfigurerAdapter` inside `com.packt.example.socialgoogle1.config` with the following content:

```
@Configuration @EnableSocial
@EnableConfigurationProperties(GoogleProperties.class)
```

```
public class GoogleConfigurerAdapter extends SocialConfigurerAdapter {
   @Autowired
   private GoogleProperties properties;
}
```

21. Now add the following method to configure the `ConnectionFactory` for the Google provider (this method must be declared inside `GoogleConfigurerAdapter`):

```
@Override
public void addConnectionFactories(ConnectionFactoryConfigurer configu
   Environment environment) {
   GoogleConnectionFactory factory =  new GoogleConnectionFactory(
      this.properties.getAppId(), this.properties.getAppSecret());
   configurer.addConnectionFactory(factory);
}
```

22. Add the following method to declare the bean responsible for providing the (DSL) *Domain Specific Language* for Google API (when importing the `Scope` class, import it from `org.springframework.context.annotation` package and the `Connection` class, you must import from `org.springframework.social.connect` package):

```
@Bean
@Scope(value = "request", proxyMode = ScopedProxyMode.INTERFACES)
public Google google(final ConnectionRepository repository) {
   final Connection<Google> connection = repository.findPrimaryConnect
   return connection != null ? connection.getApi() : null;
}
```

23. As per the documentation of Spring Social, we need to configure the `ConnectionRepository` bean using a Session scope, or in other words, a `ConnectionRepository` must be created on a per user basis. To do so, add the following two method declarations inside `GoogleConfigurerAdapter`:

```
@Override
public UsersConnectionRepository getUsersConnectionRepository(
         ConnectionFactoryLocator connectionFactoryLocator) {
   return new InMemoryUsersConnectionRepository(connectionFactoryLocat
}

@Override
public UserIdSource getUserIdSource() {
   return new AuthenticationNameUserIdSource();
}
```

24. Now, as the class `AuthenticationNameUserIdSource` retrieves the logged user

from the Spring Security context, you need to configure Spring Security, defining how to protect the application as well as declaring how to authenticate users. Create the class `SecurityConfiguration` within the package `com.packt.example.socialgoogle1.security`, containing the following initial code:

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapte
}
```

25. Then add the following method inside `SecurityConfiguration` to define what should be protected, what does not need to be protected, and how the authentication must be performed (which is defined to use form basis authentication):

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http.authorizeRequests()
         .antMatchers("/connect/google?*").permitAll()
         .anyRequest().authenticated().and()
         .formLogin().and()
         .logout().permitAll().and()
         .csrf().disable();
}
```

26. And to declare some predefined users, add the following method to `SecurityConfiguration`.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exc
   auth.inMemoryAuthentication()
         .withUser("adolfo").password("123").authorities("USER")
         .and()
         .withUser("jujuba").password("123").authorities("USER");
}
```

27. Therefore, create the main controller that decides when to redirect a user to the provider's authentication and authorization page and which retrieves the user's profile by interacting with the Google Plus API. Create the class `GooglePlusController` inside the`com.packt.example.socialgoogle1` package as presented in the following code:

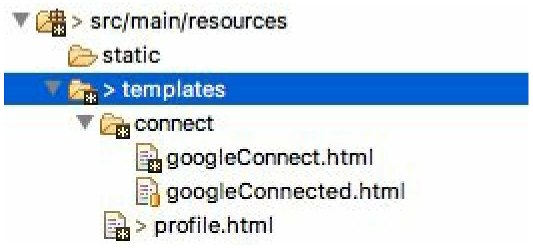
```
@Controller
public class GooglePlusController {
   @Autowired
```

```
        private Google google;
        @Autowired
        private ConnectionRepository connectionRepository;

        @GetMapping
        public String profile(Model model) {
              if (connectionRepository
                    .findPrimaryConnection(Google.class) == null) {
                    return "redirect:/connect/google";
              }

              String name = google.plusOperations()
                    .getGoogleProfile()
                    .getDisplayName();
              model.addAttribute("name", name);

              return "profile";
        }
  }
```

28. Now let's start creating all the views, primarily defining `profile.html` inside the `templates` directory, which resides within the `src/main/resources` project directory. Add the following content to `profile.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<title>LinkedIn integration</title>
</head>
<body>
   <h3>
         Hello, <span th:text="${name}">User</span>!
   </h3>
   <br />
</body>
</html>
```

29. Create the `googleConnect.html` and `googleConnected.html` files inside `templates/connect` as presented in the following screenshot:



30. Add the following content to `googleConnect.html`:

```
<html>
```

```
<head>
<title>Social Google+</title>
</head>
<body>
   <h2>Connect to Google+ to see your profile</h2>

   <form action="/connect/google" method="POST">
         <input type="hidden" name="scope"
           value="https://www.googleapis.com/auth/plus.me" />
         <div class="formInfo">
           Click the button to share your profile with
           <b>google plus</b>
         </div>
         <p>
           <button type="submit">Connect to Google</button>
         </p>
   </form>
</body>
</html>
```

31. Add the following content to `googleConnected.html`:

```
<html>
   <head><title>Social Google Plus</title></head>
   <body>
         <h2>Connected to Google</h2>
         <p>Click <a href="/">here</a> to see your profile.</p>
   </body>
</html>
```

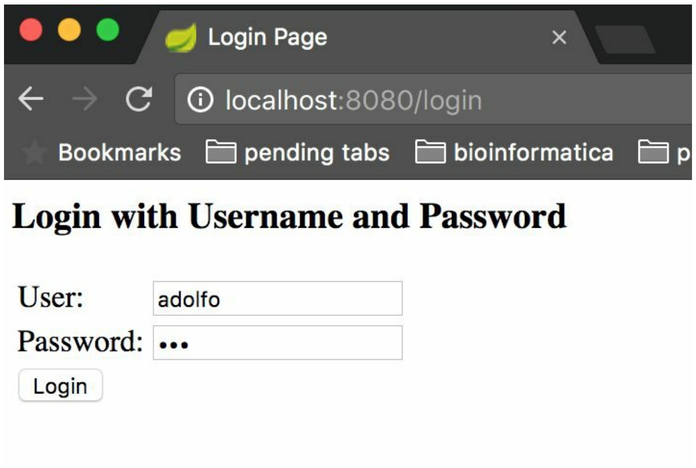
32. Now the application is ready to be executed. Start the application and go to `http://localhost:8080` to start the authorization process to interact with the Google Plus API.

How it works...

An important thing that we did for this recipe was to bind the application users to their respective connection with the OAuth 2.0 Provider (Google in this case). It's important because by doing so, we have a connection per user, unlike the other recipes using Spring Social. But instead of allowing users to register themselves to the `social-google1` application, we are using an in-memory model using pre-defined user credentials, as presented in the following code:

```
auth.inMemoryAuthentication()
    .withUser("adolfo").password("123").authorities("USER")
    .and()
    .withUser("jujuba").password("123").authorities("USER");
```

So, when running the application and pointing your browser to `http://localhost:8080`, you must be prompted by an authentication form, as follows.



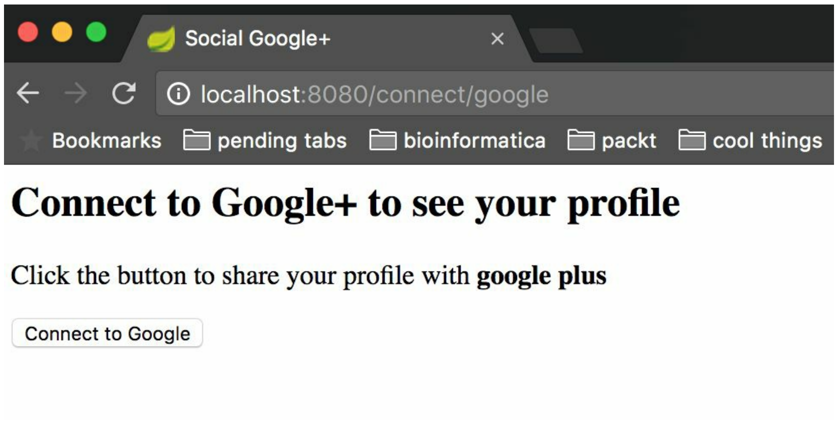
Login with Username and Password

User: adolfo

Password: •••

Login

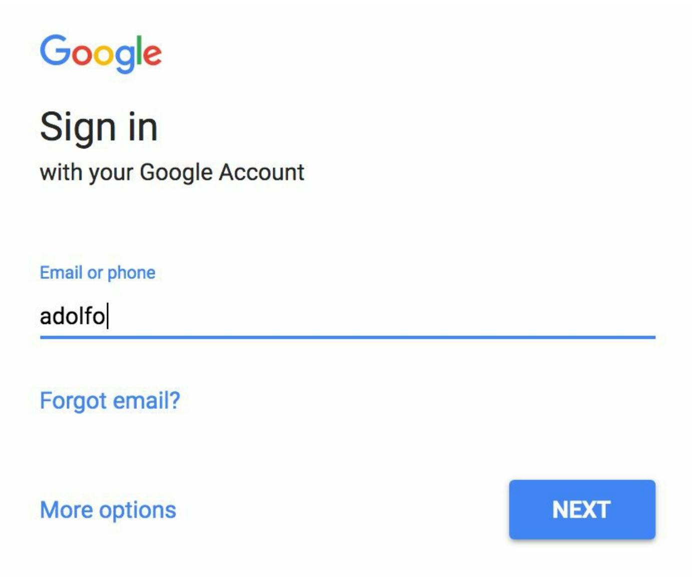
Enter one of the credentials we declared within the `SecurityConfiguration` class and click on the Login button, which will lead you to the following page:



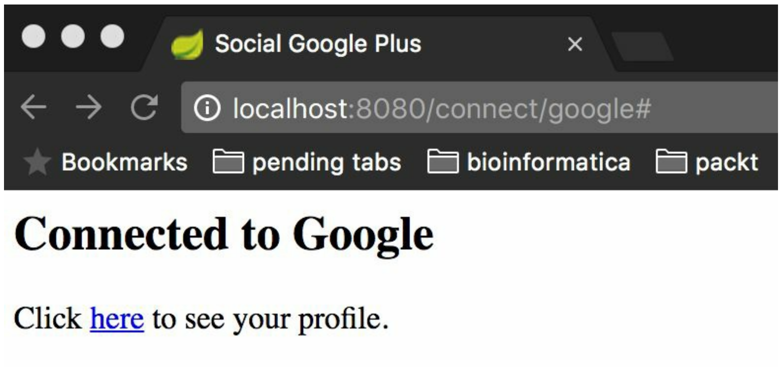
Connect to Google+ to see your profile

Click the button to share your profile with **google plus**

This is the page where you might choose to connect with Google by clicking on Connect to Google, which will redirect you to Google's authentication and authorization form as presented in the following screenshot:



Authenticate yourself and grant all the requested permissions and you will be redirected back to the connected page:



Connected to Google

Click here to see your profile.

Click on the link here and you will be redirected to the profile's HTML view which will retrieve your name from the Google Plus API. Now, if you go to `http://localhost:8080/logout`, you will be logged out, as you might expect, and if you try to log in with another user you will have to start a new connection flow proving that you have a connection per logged user.

There's more...

For this recipe, we did not configure the base URL, which must be done to avoid issues when running your application behind a proxy. To do so, you might add the following bean declaration inside `GoogleConfigurerAdapter`:

```
@Bean
public ConnectController connectController(
      ConnectionFactoryLocator locator,
      ConnectionRepository repository) {

      ConnectController controller =
            new ConnectController(locator, repository);
      controller.setApplicationUrl("http://localhost:8080");
      return controller;
}
```

Do not forget to define the same redirect URL for the OAuth 2.0 Provider and to make all the communications through TLS/SSL.

To improve security configurations, you might use a database to store all users and respective credentials being held in a cryptographically manner (these features are provided by Spring Security and can be read about in the official documents at https://projects.spring.io/spring-security/).

See also

- Preparing the environment
- Reading the user's contacts from Facebook on the server side
- Accessing OAuth 2.0 LinkedIn protected resources

Implementing Your Own OAuth 2.0 Provider

In this chapter, we will cover the following recipes:

- Protecting resources using the Authorization Code grant type
- Supporting the Implicit grant type
- Using the Resource Owner Password Credentials grant type as an approach for OAuth 2.0 migration
- Configuring Client Credentials grant type
- Adding support for refresh tokens
- Using a relational database to store tokens and client details
- Using Redis as a token store
- Implementing client registration
- Breaking the OAuth 2.0 Provider in the middle
- Using Gatling to load test the token validation process using shared databases

Introduction

Nowadays we have scenarios which demand that applications interact with a large number of services and also provide services by themselves distributed as APIs throughout the network. Despite this, it's common to allow users of our applications to grant permissions to third-party applications, where OAuth 2.0 has proven to be a good option.

In this chapter, you will learn how to create, configure, and distribute an OAuth 2.0 Provider covering distinct scenarios using all the grant types described by the OAuth 2.0 specification, as well as how to use different access token management strategies through relational databases and Redis (a NoSQL database). All the recipes in this chapter will be implemented using **Spring Security OAuth2**, which at the time of writing this book, was at the 2.2.0.RELEASE version (check the official documentation for Spring Security OAuth2 at http://projects.spring.io/spring-security-oauth/docs/oauth2.html). It's important learning how to configure your own OAuth 2.0 Provider because of the large number of integrations being done among applications nowadays. Additionally, by reading this chapter, you will be able to apply all the OAuth 2.0 details from specifications practically through the usage of Spring Security OAuth2.

Bear in mind to use TLS/SSL in production to always protect all transferred data between clients and the OAuth 2.0 Provider. This must be considered to all recipes on this book, so when running production OAuth 2.0 applications, make sure to use TLS/SSL.

Protecting resources using the Authorization Code grant type

This recipe shows you how to configure the most well-known OAuth 2.0 grant type, which is the Authorization Code grant type. After configuring an OAuth 2.0 Provider comprised of an Authorization Server and a Resource Server, the application built through this recipe will provide all the necessary Resource Owner's authorizations for resources usage (resources available through APIs protected by the Resource Server).

Getting ready

To run this recipe, you can use your preferred IDE and must have Java 8 and Maven installed. To run the examples, I recommend you to use the command line tool CURL, or install the application Postman which allows to create HTTP requests in an intuitively manner. If you want to use Postman, the installation file can be download from https://www.getpostman.com/. This recipe will use Spring Security OAuth2 Framework and to keep it as possible, we will not add any database support at moment; that is, we will use **in-memory** configuration to store client details as well as access tokens. The source code for this recipe can be downloaded from https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/auth-code-server.

How to do it...

The following steps will guide you to configure an Authorization Server and a Resource Server using Spring Security OAuth2, which prevents you from having to write an OAuth 2.0 Provider from scratch (which would be very unproductive and prone to security failures):

1. Create the initial project using Spring Initializr as we did for the other recipes in this book. Go to https://start.spring.io/ and define the following data:
    - Set up the Group as `com.packt.example`
    - Define the Artifact as `auth-code-server`
    - Add `Web` and `Security` as dependencies for this project
2. After creating the `auth-code-server` project, import it to your IDE. If you are using Eclipse, import it as a Maven project.
3. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project (to use an up-to-date Spring Security OAuth2 version, we have to override the version provided by Spring Boot):

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version>
</dependency>
```

4. Open the `application.properties` file and add the following content to configure the user of the `auth-code-server` application (you can use a different user of course, but remember to change it whenever appropriate):

```
security.user.name=adolfo
security.user.password=123
```

5. As we want to protect the user's resources through OAuth 2.0, we need to create something to be protected. To do so, create the `UserController.java` and `UserProfile.java` classes within the `com.packt.example.authcodeserver.api` package.

6. Open the `UserProfile.java` class and make sure to add the following attributes (do not forget to create appropriate getters and setters for each attribute):

```
public class UserProfile {
    private String name;
    private String email;
    // getters and setters hidden for brevity
 }
```

7. Open the `UserController.java` class and add the `@Controller` annotation at the head of the class declaration as follows:

As you might notice, Spring provides us some annotations such as `@Controller`, `@Service`, and `@Component`. Some annotations such as `@Service` and `@Component` just defines a declared class as a Spring managed bean (to be managed by Spring which allows for dependency injection mechanism). The `@Controller` annotation is a specialization of `@Component` annotation adding semantics for a web controller that can map endpoints to Java source code.

```
@Controller
public class UserController {
}
```

8. Now, let's add the respective method that will provide the endpoint which will be protected by OAuth 2.0, as presented in the following code (import the `User` class from package `org.springframework.security.core.userdetails`):

```
@RequestMapping("/api/profile")
public ResponseEntity<UserProfile> profile() {
    User user = (User) SecurityContextHolder.getContext()
         .getAuthentication().getPrincipal();
    String email = user.getUsername() + "@mailinator.com";

    UserProfile profile = new UserProfile();
    profile.setName(user.getUsername());
    profile.setEmail(email);

    return ResponseEntity.ok(profile);
}
```

9. Once we have the endpoint to be OAuth 2.0 protected, let's create the

OAuth 2.0 Authorization Server configuration by creating the `OAuth2AuthorizationServer` class within the `com.packt.example.authcodeserver.config` package.

10. Add the following annotations to `OAuth2AuthorizationServer` class and extend the `AuthorizationServerConfigurerAdapter` class which comes from the Spring Security OAuth2 project:

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends
       AuthorizationServerConfigurerAdapter {
}
```

11. To configure all the client details data, override the `configure` method which allows you to customize the `ClientDetailsServiceConfigurer` instance:

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws E
    clients.inMemory()
        .withClient("clientapp").secret("123456")
        .redirectUris("http://localhost:9000/callback")
        .authorizedGrantTypes("authorization_code")
        .scopes("read_profile", "read_contacts");
}
```

12. At the moment, the application is ready to start issuing access tokens, given the user grants permission. But to be allowed to access the user's resources (the Resource Owner profile for this recipe), we need to create the Resource Server's configuration by declaring the `OAuth2ResourceServer` class within the same package as `OAuth2AuthorizationServer`.
13. Then add the following annotations at the class level for `OAuth2ResourceServer` as follows:

```
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer
     extends ResourceServerConfigurerAdapter {
}
```

14. And to start protecting the user's profile endpoint, add the following configuration method within the `OAuth2ResourceServer` class:

```
@Override
```

```
        public void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests().anyRequest().authenticated().and()
                .requestMatchers().antMatchers("/api/**");
        }
```

15. The application is ready for access token issuing as well as access token validation through the API's usage.

How it works...

Through this recipe we will be able to access the user's resources, which must be available at `http://localhost:8080/api/profile` if the application is running correctly. Supposing you have followed all the steps in this recipe and the application is running, after retrieving an access token, you would be able to send a request to the profile endpoint to obtain the following **JavaScript Object Notation** (**JSON**) result:

```
{
 "name": "adolfo",
 "email": "adolfo@mailinator.com"
}
```

But to access this endpoint, you have to present a valid access token retrieved after the user's approval to share her profile. The access token in this case will be validated because of the Resource Server configuration. By adding the `@EnableResourceServer` annotation as we did for the `OAuth2ResourceServer` class, we are importing some configurations which will add the filter `OAuth2AuthenticationProcessingFilter` within the Spring Security's `FilterChain` configuration. That's the filter that will be in charge of starting the access token validation process for any endpoint that matches the `/api/**` pattern.

In addition to the Resource Server configuration, we added the Authorization Server configuration through the `OAuth2AuthorizationServer` class. By looking at the `OAuth2AuthorizationServer` class, you might realize that it does not contain a lot of code. It's really easy to configure an Authorization Server with Spring Security OAuth2 Framework, but there are a lot of things happening behind the scenes.

By adding the `@EnableAuthorizationServer` annotation, we are importing some important configuration classes which are `AuthorizationServerEndpointsConfiguration` and `AuthorizationServerSecurityConfiguration`. The `AuthorizationServerEndpointsConfiguration` class, has an important dependency for an OAuth 2.0 Authorization Server to work. That's the `AuthorizationServerEndpointsConfigurer` class, which declares the following beans:

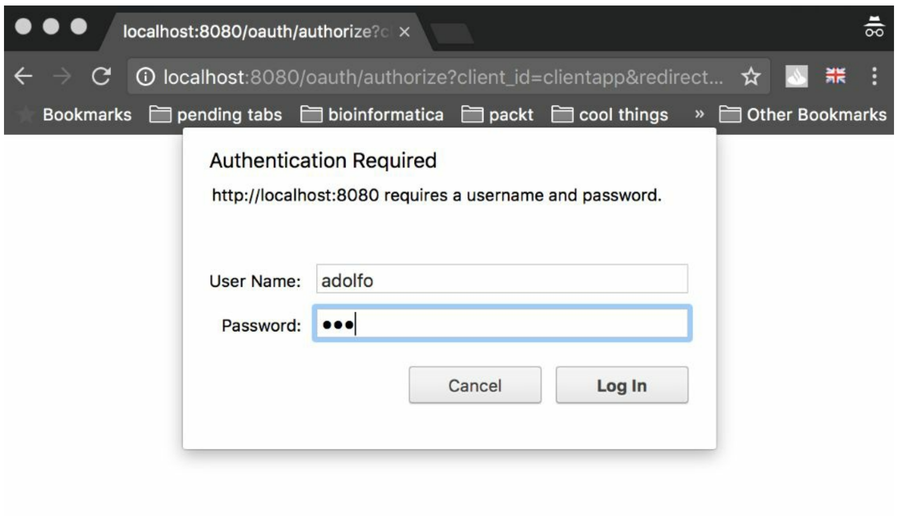
- AuthorizationEndpoint
- TokenEndpoint
- CheckTokenEndpoint

Those beans declare OAuth 2.0 related endpoints as endpoints for authorization flow and for access token and refresh token requests. If you want to deeply understand how each endpoint works, I encourage you to read the source code of Spring Security OAuth2 available at GitHub. Running the application, we can interact with some of the OAuth 2.0 declared endpoints. To see how the process works, start the application through your IDE or by running the `mvn spring-boot:run` command. After the application is running, point to the following URL through your browser:

```
http://localhost:8080/oauth/authorize?client_id=clientapp&redirect_uri=http:/
```

As we are using the Authorization Code grant type, we need to redirect the Resource Owner to the authorization page which is declared by the `/oauth/authorize` endpoint. Notice that we are sending the `client_id`, `redirect_uri`, `response_type`, which must be `code` and the scope as `read_profile`. If you want to better understand each of these parameters you can read about the Authorization Request section of the OAuth 2.0 specification at https://tools.ietf.org/html/rfc6749#section-4.1.1. Note that we are not sending the important `state` parameter to avoid CSRF attacks (we will see more of this in Chapter 8, *Avoiding Common Vulnerabilities*).

After going to the authorization endpoint, the Authorization Server must authenticate the Resource Owner as per the OAuth 2.0 specification:


localhost:8080/oauth/authorize?c
localhost:8080/oauth/authorize?client_id=clientapp&redirect...
Bookmarks
pending tabs
bioinformatica
packt
cool things
Other Bookmarks
Cancel
Log In

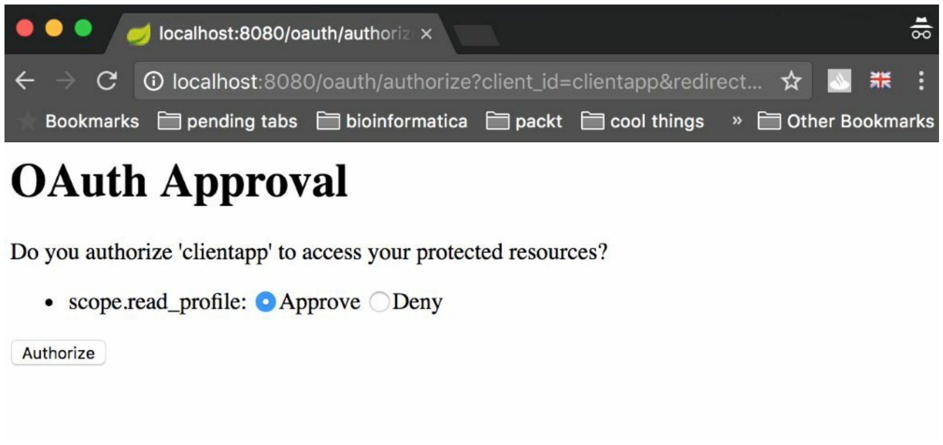
Authentication Required

http://localhost:8080 requires a username and password.

User Name: adolfo

Password: •••

Enter the User Name and Password declared in the `application.yml` configuration file, which in my case is `adolfo` and `123` respectively. After clicking on Log In, you will be redirected to the user's consent page where the Resource Owner can choose which scopes to grant permission to for third-party application (client). The following page is generated automatically by Spring Security OAuth2 but can easily be replaced by a custom one:


localhost:8080/oauth/authoriz
localhost:8080/oauth/authorize?client_id=clientapp&redirect...
Bookmarks
pending tabs
bioinformatica
packt
cool things
Other Bookmarks
Authorize

OAuth Approval

Do you authorize 'clientapp' to access your protected resources?

- scope.read_profile: [✓ Approve] Deny

The next step is to click on Authorize so that the user can be redirected back to the client application, which must be specified by the `redirect_uri` attribute. When redirected back to the redirection URI, the Authorization Server must issue an Authorization Code which depicts the user's approval granting permission for the specified client on the specified Resource Server. The Authorization Code comes as a query string parameter defined as `code` as follows:

```
http://localhost:9000/callback?code=5sPk8A
```

The previous URL has the code randomly generated by the Authorization Server which must be used by the client to request for an access token at the server side. To request an access token using the previous received Authorization Code, run the following CURL command through any terminal (pay attention to the code parameter which must be different for you):

```
curl -X POST --user clientapp:123456 http://localhost:8080/oauth/token -H "cc
```

Note that we need to use the same redirect URI we have sent before on the authorization request. As per the OAuth 2.0 specification, the redirect URI must be sent if you have sent the redirect URI on the authorization request. After running the previous CURL command, you should receive something like this:

```
{
   "access_token": "43c6f525-041f-43c8-b970-  82ba435d3c2c",
   "token_type": "bearer",
   "expires_in": 43199,
   "scope": "read_profile"
}
```

Let's try to access the user's profile by presenting the retrieved access token to the `/api/profile` endpoint, as follows:

```
curl -X GET http://localhost:8080/api/profile -H "authorization: Bearer 43c6f
```

If everything is running as expected, you should see the following content in the output:

```
{
   "name": "adolfo",
   "email": "adolfo@mailinator.com"
}
```

There's more...

This recipe was written to be as straightforward as possible, but when configuring an OAuth 2.0 Provider, consider using a database instead of declaring all the client details using an in-memory database as we did.

We are also supposed to use TLS/SSL to protect any interaction between the client and OAuth 2.0 Provider components.

Supporting the Implicit grant type

This recipe shows you how to configure the Implicit grant type, which is appropriate for web applications which run directly on web browsers (such as JavaScript applications). This recipe will provide everything needed to allow the client to use resources in the name of the Resource Owner by accessing an OAuth 2.0 protected API.

Getting ready

To run this recipe, you can use your preferred IDE and must have Java 8 and Maven installed. As this grant type run totally on the web browser, you don't need to run commands to retrieve access tokens as if they were being performed on the server side. Here the access token will be retrieved implicitly when the user grants permissions for third-party applications. This recipe will use Spring Security OAuth2 Framework and, to keep it as straight as possible, we will not add any database support. The source code for this recipe can be downloaded from https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/implicit-server.

How to do it...

The following steps will guide you to configure an Authorization Server and a Resource Server using Spring Security OAuth2:

1. Create the initial project using Spring Initializr, as we did for other recipes in this book. Go to https://start.spring.io/ and define the following data:
    - Set up the Group as `com.packt.example`
    - Define the Artifact as `implicit-server`
    - Add `Web` and `Security` as dependencies for this project

2. After creating the `implicit-server` project, import it to your IDE. If you are using Eclipse, import it as a Maven project.
3. Open the `pom.xml` file and add the following dependency as we will use the Spring Security OAuth2 project (I recommend you to use the latest version of this project, particularly if you are using JWT, which is not the case for this recipe):

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
 <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. Open the `application.properties` file and add the same configuration as we did for the first recipe to set up the user's credentials (which were username `adolfo` and password `123`).
5. Create the `UserProfile.java` and `UserController` classes within the `com.packt.example.implicitserver.api` package. The content for both classes must be the as same provided for the first recipe (you can also download the source code from GitHub if you want).
6. Now create the `OAuth2ResourceServer` class with the following content to declare how to protect endpoints which matches the `/api/**` pattern. This class should be created within the `com.packt.example.implicitserver.confi` package:

```
@Configuration
```

```
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
    public void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests().anyRequest().authenticated()
        .and()
            .requestMatchers().antMatchers("/api/**");
    }
}
```

7. Note that the Resource Server is being configured the same way we did when adding support for the Authorization Code grant type.

8. Create the `OAuth2AuthorizationServer` class as presented in the following code, to configure the Implicit grant type:

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConf
    public void configure(ClientDetailsServiceConfigurer clients) thro
        clients.inMemory()
            .withClient("clientapp").secret("123456")
            .redirectUris("http://localhost:9000/callback")
            .authorizedGrantTypes("implicit")
            .accessTokenValiditySeconds(120)
            .scopes("read_profile", "read_contacts");
    }
}
```

9. The preceding class looks similar to the one created for the Authorization Code grant type. The difference now is that we are defining `authorizedGrantTypes` as Implicit and we are also defining a short validity time for the access token.
10. Run the application using your IDE actions or by running the Maven command, `mvn spring-boot:run`.

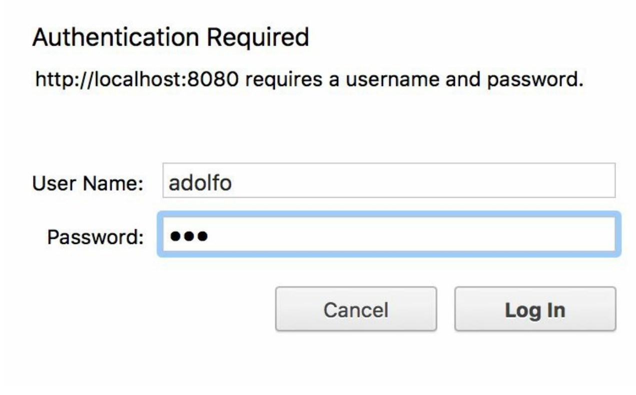
How it works...

Everything we did when using the `@EnableAuthorizationServer` and `@EnableResourceServer` annotations configures our current application, `implicit-server` to support both OAuth 2.0 token validation as well as the OAuth 2.0 endpoints for authorization and token requests. The main difference now is that we have changed the authorized grant type to use, so the authorization flow will be a little bit different. In addition, we have the fact that we don't need to send a request to `/oauth/token` for this grant type because the access token is retrieved implicitly, as defined by the name of the grant type by itself.

With the application running, go to the following authorization URL which is sending the `response_type` parameter as `token` instead of `code`:

```
http://localhost:8080/oauth/authorize?client_id=clientapp&redirect_uri=http:/
```

Note that we are not encoding the parameters using URL encoding; that's because of didactical purposes. When implementing a real application, consider encoding all the parameters. After going to the authorization endpoint, the Authorization Server will authenticate the Resource Owner by asking for the User Name and Password, as shown in the following screenshot:

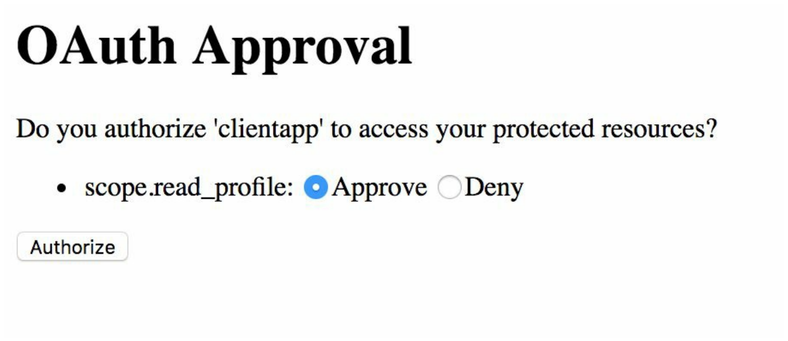

Authentication Required

http://localhost:8080 requires a username and password.

User Name: adolfo

Password: ●●●

Enter the User Name and Password, which in my case was `adolfo` and `123` respectively. After being authenticated, the user will be sent to the user's consent page, as follows:


OAuth Approval

Do you authorize 'clientapp' to access your protected resources?

- scope.read_profile: [✓ Approve] Deny

Select the required scope, which is `read_profile`, and click on Authorize so you get redirected back to the redirect URI callback, as defined by the `redirect_uri` parameter. As you might see in the following code, the `access_token` is retrieved as a URL fragment in addition to the state parameter that we sent before when redirecting the user to the Authorization Server's authorization endpoint:

```
http://localhost:9000/callback#access_token=a68fce80-522f-43ee-85d4-6705c34e5
```

Along with the `access_token` and the state parameter, we also receive back the `token_type` and `expires_in` parameter. These parameters were also present when requesting for an access token when using the Authorization Code grant type. But when using the Implicit grant type, it's important to bear in mind that this grant type is not allowed to issue an access token as per the OAuth 2.0 specification. This behavior makes sense because as the user must be present when using an application that runs within the browser, then the user can always authorize the third-party application again if needed.

Moreover, the Authorization Server has plenty of conditions to recognize the user's session and avoid asking the Resource Owner to authenticate and authorize the client again. Another reason for not issuing a refresh token for the Implicit grant type, is because this grant type is aimed at public applications that are not able to protect confidential data, as is the case with refresh tokens.

Now, try to retrieve the user's profile through the `/api/profile` endpoint using the previously issued access token and the user's profile should be returned as JSON content.

There's more...

When using the Implicit grant type, always be aware of requiring the redirect URI registration by the third-party application. This will ensure that the access token won't be delivered to an undesired registered client. Any malicious user might be capable of registering an application to try to impersonate another regular client application to receive an access token in its name, and the results might be disastrous if not requiring the registering of the redirection URI.

Another important issue that will be enforced on each recipe is to bear in mind the need to use TLS/SSL in production to always protect all the transferred data between clients and the OAuth 2.0 Provider.

See also

- Protecting resources using the Authorization Code grant type
- Using the Resource Owner Password Credentials grant type as an approach for OAuth 2.0 migration
- Configuring Client Credentials grant type

Using the Resource Owner Password Credentials grant type as an approach for OAuth 2.0 migration

This recipe will show you how to configure the Resource Owner Password Credentials, or Password Credentials for short. Although this grant type should be avoided at any cost, because by using it we are asking for the user's credentials (and that's what OAuth 2.0 wants to solve by the user's access delegation), it's important to mention this recipe as a strategy when migrating from a user's credential sharing approach to the OAuth 2.0 approach. In addition, it might be used safely when both the client and the OAuth 2.0 Provider belong to the same solution.

Getting ready

To run this recipe, you can use your preferred IDE and must have Java 8 and Maven installed. It's also recommended to have CURL or Postman installed, because we will interact with the Authorization Server and Resource Server automatically without actually using any client. By running this recipe, your OAuth 2.0 Provider will be able to issue an access token for any registered client by using the Resource Owner's credentials. This recipe still does not use any kind of persistent database to store client details and tokens. The source code for this recipe can be downloaded at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/password-server.

How to do it...

The following steps will guide you to configure an Authorization Server and a Resource Server using Spring Security OAuth2:

1. Create the initial project using Spring Initializr as we did for other recipes in this book. Go to https://start.spring.io/ and define the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `password-server`
   - Add `Web` and `Security` as dependencies for this project
2. After creating the `password-server` project, import it to your IDE. If using Eclipse, import it as a Maven project.
3. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
 <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. Open the `application.properties` file and add the same configurations that we did for the first recipe to set up the user's credentials, which were `adolfo` for `security.user.name` and `123` for `security.user.password`.
5. Create the `UserProfile.java` and `UserController` classes within the `com.packt.example.passwordserver.api` package. The source code for both classes, must be the same that was provided for the first recipe on this chapter (you can also download the source code from GitHub if you want).
6. Now create the `OAuth2ResourceServer` class with the following content to declare how to protect endpoints, which matches the `/api/*` pattern. This class should be created within the `com.packt.example.passwordserver.config` package:

```
@Configuration
@EnableResourceServer
```

```
public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests().anyRequest().authenticated()
        .and()
            .requestMatchers().antMatchers("/api/**");
    }
}
```

7. Create the `OAuth2AuthorizationServer` class within the same package as `OAuth2ResourceServer`, as follows, to configure the Password grant type:

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends
        AuthorizationServerConfigurerAdapter {
    @Override
    public void configure(ClientDetailsServiceConfigurer clients)
            throws Exception {
        clients.inMemory()
            .withClient("clientapp")
            .secret("123456")
            .redirectUris("http://localhost:9000/callback")
            .authorizedGrantTypes("password")
            .scopes("read_profile", "read_contacts");
    }
}
```

8. The only difference between the `OAuth2AuthorizationServer` created for this recipe and the others created for any grant type presented in this chapter is the declared `authorizedGrantType`, which in this case is `password`. The main difference appears when interacting with our OAuth 2.0 Provider.

9. If you run the application as it is, you will face the following error when trying to request an access token. Such an error occurs because the Password grant type requires you to declare an `AuthenticationManager` inside the `OAuth2AuthorizationServer` configuration class:

```
{
 "error": "unsupported_grant_type",
 "error_description": "Unsupported grant type: password"
}
```

10. Inject the following attribute within the `OAuth2AuthorizationServer` class:

```
@Autowired
private AuthenticationManager authenticationManager;
```

11. Then set up the `authenticationManager` for `AuthorizationServerEndpointsConfigurer` by overriding the following method from `AuthorizationServerConfigurerAdapter` , within `OAuth2AuthorizationServer` as follows:

```
    public void configure(AuthorizationServerEndpointsConfigurer endpoints
        endpoints.authenticationManager(authenticationManager);
    }
```

12. Now run the application through your IDE actions or by running the Maven `mvn spring-boot:run` command.

How it works...

As we are using the same configuration annotations for the `OAuth2AuthorzationServer` and `OAuth2ResourceServer` classes, all the OAuth 2.0 support is being added the same way as that provided by Spring Security OAuth2. The important difference in configurations here is denoted by the usage of an `authenticationManager` instance. The `authenticationManager` has to be present because the Authorization Server has to authenticate the Resource Owner's credential when trying to respond for a required access token by the third-party application.

In addition to the configuration differences, we also have an important difference in the authorization flow by itself. When using this grant type, the user (or the Resource Owner) has to send her credentials, which will be in the power of the client application. There must be a trust relation between the Resource Owner and the client application in most cases when dealing with the client and server that are comprised of the same solution (for example, if you as a Resource Owner are interacting with the official Facebook's client application, which in turn interacts with Facebook's server-side applications).

To see how this grant type works for this recipe, start the application and try sending the following request, which is sending the Resource Owner's credentials as the form parameter's username and password respectively (look for the client authentication which must always be performed by the Authorization Server when requesting an access token through the `/oauth/token` endpoint):

```
curl -X POST --user clientapp:123456 http://localhost:8080/oauth/token -H "ac
```

After running the previous command, you should see the following JSON response (obviously with a different access token):

```
{
   "access_token": "28405009-b53d-4e52-bfc3-c8889a477675",
   "token_type": "bearer",
   "expires_in": 43199,
   "scope": "read_profile"
}
```

Note that as we are not defining any access token expiration time; Spring Security OAuth2 defines 43200 seconds by default. Depending on the grant type used, you should consider using a short time access token (for example, when using the Implicit grant type).

Now that you have a valid access token, try to retrieve the user's profile through the following command:

```
curl -X GET http://localhost:8080/api/profile -H "authorization: Bearer 28405
```

After running the previous command, you should see the user's name and email.

There's more...

Besides the fact that we should avoid this grant type, it's not a problem if you use it when interacting with one server that belongs to the same domain of the client application. That is to say, that both client and OAuth 2.0 Provider belong to the same solution as well. As it comprises of the same application divided between the client and server, the users can trust sharing the credentials because it belongs to the same application. The only important thing to mention is that, as the client application, it must throw away the client's username and password required to obtain an access token.

Once again, do not forget to use TLS/SSL when running such solutions described by this recipe in production.

See also

- Protecting resources using the Authorization Code grant type
- Supporting the Implicit grant type
- Configuring the Client Credentials grant type

Configuring the Client Credentials grant type

This recipe shows you how to configure the Client Credentials grant type, which is appropriate when an application needs to access resources for its own benefit instead of accessing the Resource Owner's resources.

Getting ready

To run this recipe, you can use your preferred IDE and must have Java 8 and Maven installed. For this recipe, we will be executing some HTTP requests through the usage of CURL commands as we did before. Make sure you have CURL or Postman tools installed before continuing. This recipe will use Spring Security OAuth2 Framework and we will not add any database support. The source code for this recipe can be downloaded from https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/client-credentials-server.

How to do it...

The following steps will guide you to configure an Authorization Server and a Resource Server using Spring Security OAuth2:

1. Create the initial project using Spring Initializr, as we did for other recipes in this book. Go to https://start.spring.io/ and define the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `client-credentials-server`
   - Add `Web` and `Security` as dependencies for this project
2. After creating the `client-credentials-server` project, import it to your IDE. If using Eclipse, import it as a Maven project.
3. Open the `pom.xml` file and add the following dependency as we will use the Spring Security OAuth2 project:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
 <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. Open `application.properties` and add the same content that we have added for the first recipe to configure the user's credentials.
5. Although this recipe isn't focused on the user's experience, we still have to create the API to retrieve user profile. To keep users being able to access the application in a safe manner to access their own profile, create the `UserProfile` and `UserController` classes within the `api` sub-package. The content for both classes must be the same as that was provided in the first recipe (generate a constructor using fields for `UserProfile` class).
6. Open `UserController` class and replace the value for `@RequestMapping` annotation with `"/user"` instead of `"/api/profile"`.
7. As we are using the Client Credentials grant type, we won't allow any client to access the user's profile. Instead of the user's profile, we will create an API where the client application is able to retrieve all the users registered on the application server that we are protecting with OAuth

2.0. Perhaps, this business rule does not make sense to you, but let's stay focused on how to use the Client Credentials grant type with Spring Security OAuth2 instead of focusing on the business product itself. So, create the `AdminController` class as presented in the following code within the package `com.packt.example.clientcredentialsserver.api`:

```
@Controller
@RequestMapping("/api")
public class AdminController {

    @RequestMapping("/users")
    public ResponseEntity<List<UserProfile>> getAllUsers() {
        return ResponseEntity.ok(getUsers());
    }

    private List<UserProfile> getUsers() {
        List<UserProfile> users = new ArrayList<>();
        users.add(new UserProfile("adolfo", "adolfo@mailinator.com"));
        users.add(new UserProfile("demigreite", "demigreite@mailinator
        users.add(new UserProfile("jujuba", "jujuba@mailinator.com"));
        return users;
    }
}
```

8. Now to protect this API, create the following Resource Server configuration class declared as `OAuth2ResourceServer` inside the `com.packt.example.clientcredentialsserver.config` package:

```
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest()
            .authenticated()
        .and()
            .requestMatchers()
            .antMatchers("/api/**");
    }
}
```

9. And to issue access tokens, create the `OAuth2AuthorizationServer` within the same package as the `OAuth2ResourceServer` class (now configure a different client ID and client secret, as well as the `authorizedGrantTypes`):

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends
        AuthorizationServerConfigurerAdapter {
```

```
        @Override
        public void configure(ClientDetailsServiceConfigurer clients)
                throws Exception {
            clients.inMemory()
                .withClient("clientadmin")
                .secret("123")
                .authorizedGrantTypes("client_credentials")
                .scopes("admin");
        }
    }
```

10. Now to protect the `/users` API from users that are not registered, let's create the following Spring Security configuration class within the config package as we did for the OAuth 2.0 Provider configuration classes:

```
    @Configuration
    @EnableWebSecurity
    public class WebSecurityConfiguration extends WebSecurityConfigurerAda
        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                    .anyRequest().authenticated().and()
                .antMatcher("/user/**")
                    .httpBasic()
                    .and()
                .csrf().disable();
        }
    }
```

11. Now run the application through your IDE actions or by running the Maven `mvn spring-boot:run` command.

How it works...

This recipe presents a particular case of OAuth 2.0 grant types which allows for any registered client to access resources on its own behalf. All the previous recipes, described how to use grant types which allow a third-party application to access resources on behalf of the Resource Owner. Because of the purpose of this grant type, we have created a fictitious API that provides one endpoint so the client could retrieve all registered users on the OAuth 2.0 Provider.

Almost all the configurations for the Authorization Server and Resource Server were similar to the other grant types. The main difference now is that we don't need any redirection URI. In addition, we also have to set up the correct grant type to use, as presented in the following code:

```
clients.inMemory()
  .withClient("clientadmin")
  .secret("123")
  .authorizedGrantTypes("client_credentials")
  .scopes("admin");
```

Let's start the application to understand how to interact with the main pieces of this OAuth 2.0 Provider. When using the Client Credentials grant type, we do not have any user interaction so we don't have the authorization flow, but we just have to request for an access token for sending the Client's Credentials and the grant type, which is `client_credentials`, as presented in the following CURL command line:

```
curl -X POST "http://localhost:8080/oauth/token" --user clientadmin:123 -d "g
```

After running the previous command, you should see something similar to the following:

```
{
    "access_token":"f6f81a52-7920-4f95-a83b-72fbfb5188c5",
    "token_type":"bearer",
    "expires_in":43157,
    "scope":"admin"
}
```

It's important to note that when using the Client Credentials grant type, as per the OAuth 2.0 specification, the Authorization Server should not issue a refreshed token because there is no reason to worry about the user experience in this case. When using Client Credentials, the client by itself is able to request a new access token in case of an expiration.

To use the access token, the process is the same because, as you might already know, it doesn't matter how the access token was obtained. So, to make a request for an OAuth 2.0 protected resource, you just need to send the HTTP authorization header with the retrieved access token as follows:

```
curl "http://localhost:8080/api/users" -H "Authorization: Bearer f6f81a52-792
```

After running the previous command, you should see the following result:

```
[
  { "name": "adolfo", "email": "adolfo@mailinator.com"},
  { "name": "demigreite", "email": "demigreite@mailinator.com"}
]
```

This recipe shows that you can still protect the other resources so that they can be accessed safely using another authentication mechanism, such as HTTP Basic Authentication or any other. To exemplify it, we have declared the configuration class `WebSecurityConfiguration` which extends from `WebSecurityConfigurerAdapter`. As you can see in the following code, all the endpoints that match with `/user/**` pattern would be protected using HTTP Basic Authentication:

```
.antMatcher("/user/**").httpBasic()
```

Note that you are not able to make a request to the `/user` endpoint using the previously issued access token because this endpoint is not protected by OAuth 2.0, but by HTTP Basic Authentication. The right way to retrieve the user's profile now is to send the following request:

```
curl "http://localhost:8080/user" --user adolfo:123
```

There's more...

This grant type is supposed to be used by applications that do not access resources on behalf of any Resource Owner, but for their own purposes. It has been adopted for the current ecosystem of microservices, because it's easy to rotate access tokens by using refresh tokens and it's also easy to use the dynamic registration process, automating all the process when integrating services. It does not need to be only interactions between microservices, but also include with third-party applications despite the fact of issuing an access token manually for any given client (it depends on the scenario of course, because if you have too much clients it would be impossible to manage client registration manually).

Shortly the word microservices, according to Sam Newman, in the book Building Microservices, it means small autonomous services that work together with well-defined responsibilities.

Regarding the fact of many small services talking to each other in such a way that we have many access token validations, it's important to consider the strategy used for token validation because such an approach might be faced by performance issues. Take a look at the final recipe in this chapter to learn how to create a load testing application, using Gatling to measure the response times when validating access tokens.

See also

- Protecting resources using the Authorization Code grant type
- Supporting the Implicit grant type
- Using the Resource Owner Password Credentials grant type as an approach for OAuth 2.0 migration

Adding support for refresh tokens

This recipe covers an important feature described by the OAuth 2.0 specification and implemented by Spring Security OAuth2 as well. That's the refresh token grant type, which allows for a better user experience because the Resource Owner does not have to go through all the steps of authentication and authorization against the Authorization Server every time an access token expires.

Getting ready

To run this recipe, you can use your preferred IDE and must have Java 8 and Maven installed. As we will add support to refresh tokens for the Authorization Code and Password grant types, now we will interact with the Authorization Server and Resource Server using the already known tools that are CURL and your web browser. By running this recipe, your OAuth 2.0 Provider will be able to issue an access token and a refresh token for any registered client. This recipe still does not use any kind of persistent database to store client details and tokens and the source code can be downloaded at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/refresh-server.

How to do it...

The following steps will guide you to configure an Authorization Server and a Resource Server using Spring Security OAuth2:

1. Create the initial project using Spring Initializr, as we did for the other recipes in this book. Go to https://start.spring.io/ and define the following data:
    - Set up the Group as `com.packt.example`
    - Define the Artifact as `refresh-server`
    - Add `Web` and `Security` as dependencies for this project
2. After creating the `refresh-server` project, import it to your IDE. If using Eclipse, import it as a Maven project.
3. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
 <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. Open the `application.properties` file and add the same configuration that we did for the first recipe to set up the user's credentials, which were `adolfo` for `security.user.name` and `123` for `security.user.password`.
5. To have an API to explore and to protect it using OAuth 2.0, you must create the `UserController` and `UserProfile` classes, within the `com.packt.example.refreshserver.api` package. The content for both classes must be the same as that provided for the first recipe (remember that you can download the source code from GitHub if you want).
6. Now let's create the classes which will be present within the `com.packt.example.refreshserver.config` package beginning by creating the Resource Server configuration, as described by the following source code:

```
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
    @Override
```

```
        public void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests()
                .anyRequest()
                .authenticated()
            .and()
                .requestMatchers()
                .antMatchers("/api/**");
        }
    }
```

7. And for the Authorization Server configuration, create the `OAuth2AuthorizationServer` class as follows:

```
    @Configuration
    @EnableAuthorizationServer
    public class OAuth2AuthorizationServer extends
      AuthorizationServerConfigurerAdapter {

     @Override
     public void configure(ClientDetailsServiceConfigurer clients)
       throws Exception {
      clients.inMemory()
       .withClient("clientapp")
       .secret("123456")
       .authorizedGrantTypes(
           "authorization_code", "password", "refresh_token")
       .accessTokenValiditySeconds(120)
       .scopes("read_profile", "read_contacts");
     }
    }
```

8. Notice that the Authorization Server we are configuring has support for the Authorization Code, Password, and refresh token grant types. The refresh token can also be considered a grant type because it also describes how to request for new access tokens. In addition, to retrieve a refresh token, we use the same endpoint used to retrieve an access token, that is `/oauth/token`.
9. Also notice the usage of the `accessTokenValiditySeconds` method from `ClientDetailsServiceConfigurer` which is defining the expiration time of the access token to happen 2 minutes after the token is issued.
10. As we have used the Password grant type besides the other two, we need to inject an `AuthenticationManager` and set up the injected `AuthenticationManager` on `AuthorizationServerEndpointsConfigurer`. To do so, add the following snippet of code within the `OAuth2AuthorizationServer` class:

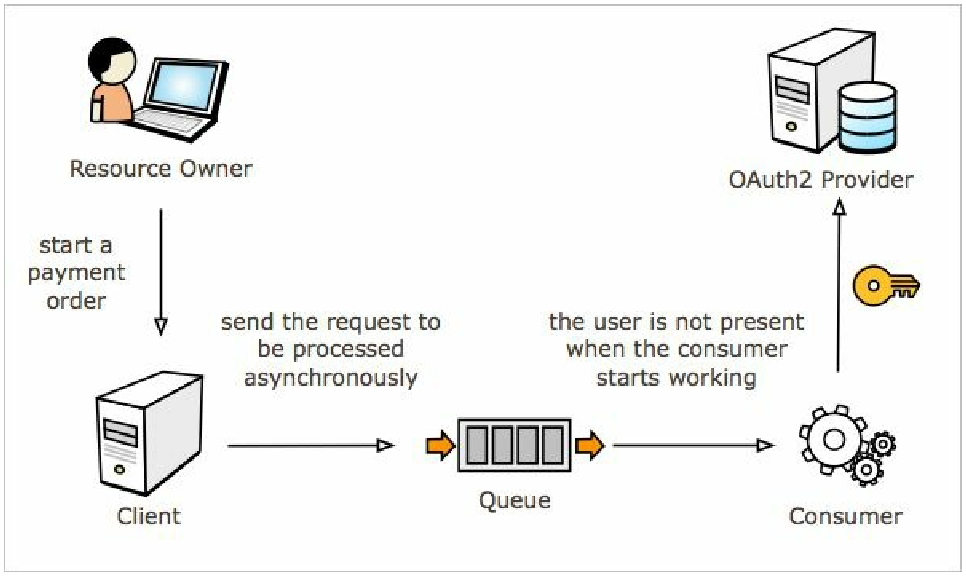
```
    @Autowired
```

```
        private AuthenticationManager authenticationManager;

        @Override
        public void configure(AuthorizationServerEndpointsConfigurer endpoints
                throws Exception {
            endpoints.authenticationManager(authenticationManager);
        }
```

11. Now run the application through your IDE actions or by running the Maven `mvn spring-boot:run` command.

How it works...

Imagine if every time an access token became invalid because of expiration time, the user will have to go through all the process of authenticating against the Authorization Server and granting all the permissions again. Besides the fact of the user experience being compromised, the user might not be present at a specific time. Once the user has granted permission for third-party applications to access resources on its behalf, this third-party application can use the resources even if the user is not logged in. Take a look at the following image that describes a fictitious scenario to better understand how an application can access a user's resources when the user is not present:


Resource Owner
OAuth2 Provider
start a payment order
send the request to be processed asynchronously
the user is not present when the consumer starts working
Client
Queue
Consumer

As you can see in the preceding image, when the consumer starts processing the payment order against the OAuth 2 Provider, which might be a payment provider by itself, the user is not present at all. The consumer would not be able to ask the user to authenticate and authorize the issue of a new access token. Furthermore, all the processing is happening on the server side.

Given the need for refresh tokens, Spring Security OAuth2 allows you to configure this feature by defining one more authorized grant types as a `refresh_token`, as follows:

```
.authorizedGrantTypes("authorization_code", "password", "refresh_token")
```

And to help you to start testing the refresh token usage, we have also defined a pretty short expiration time for the access token of 120 seconds (2 minutes), as follows:

```
.accessTokenValiditySeconds(120)
```

Make sure the application is running and let's start by requesting an access token. You can use the Authorization Code flow, but here I am using the Password grant type for practical reasons. So to retrieve an access token, we can send the following request to the Authorization Server:

```
curl -X POST --user clientapp:123456 http://localhost:8080/oauth/token -H "ac
```

The result now must have one more field which is the `refresh_token`, and is as shown here:

```
{
  "access_token":"91541ac7-8d63-4106-9660-c1847fd4b37e",
  "token_type":"bearer",
  "refresh_token":"985436a9-85cc-45ce-90d4-66a840a1a5dd",
  "expires_in":119,
  "scope":"read_profile"
}
```

Try to access the user's profile using the issued access token to see if everything is working fine and wait for 2 minutes to try a new request. Send the following request after 2 minutes:

```
curl -X GET http://localhost:8080/api/profile -H "authorization: Bearer 91541
```

The result should be as follows:

```
{
  "error":"invalid_token",
  "error_description":"Access token expired: 91541ac7-8d63-4106-9660-c1847fd4
}
```

Now it's time to request for a new access token using the previously issued refresh token. Send the following request using the command line:

```
curl -X POST --user clientapp:123456 http://localhost:8080/oauth/token -H "cc
```

The result must be a brand new access token that can be used to keep accessing the user's resources (which in this case is the user's profile).

There's more...

As presented by this recipe, the grant types allowed for using refresh tokens are the Authorization Code and Password grant types. Notice that both grant types are aimed at confidential clients. That is, applications capable of using both described grant types are client types which are able to store confidential data in a safe manner. When working with public clients, you are not supposed to use refresh tokens because public clients can't store the refresh token safely.

The request for an access token when using refresh tokens must also be encrypted by SSL/TLS, as mentioned before, initiating any interaction between the OAuth 2.0 Provider and the client.

TIP

If you are using a custom `UserDetailsService`*, you have to inject it within the* `AuthorizationServer` *configuration class and have to set up the* `UserDetailsService` *property for* `AuthorizationServerEndpointsConfigurer` *(the same as way we did for* `AuthenticationManager`*). This has to be set up because when trying to refresh an access token, the Resource Owner could have redefined her credentials which could invalidate the permission granted before.*

See also

- Protecting resources using the Authorization Code grant type
- Using the Resource Owner Password Credentials grant type as an approach for OAuth 2.0 migration

Using a relational database to store tokens and client details

This recipe provides you with a storage strategy that is more production-like instead of using an in-memory registry. The strategy presented in this recipe helps you how to use a **Relational Database Management System** (**RDBMS**) to store all client details and data related to tokens.

Getting ready

To run this recipe, you need the MySQL database besides the other tools, such as your preferred IDE and tools that allows to interact with the OAuth 2.0 Provider. The source code for this recipe is available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/rdbm-server.

How to do it...

Perform the following steps to set up a database as the store for the client details and tokens repository:

1. Create the initial project using Spring Initializr as we did for other recipes in this book. Go to https://start.spring.io/ and define the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `rdbm-server`
   - Add `Web`, `JPA`, `MySQL` and `Security` as dependencies for this project
2. After creating the `rdbm-server` project, import it to your IDE. If using Eclipse, import it as a Maven project.

3. Open the `pom.xml` file and add the following dependencies:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. Open the `application.properties` file and add the following content to configure the user's credentials as well as the database connection properties:

```
security.user.name=adolfo
security.user.password=123
spring.datasource.url=jdbc:mysql://localhost/oauth2provider
spring.datasource.username=oauth2provider
spring.datasource.password=123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Di
spring.jpa.properties.hibernate.hbm2ddl.auto=validate
```

5. To start using the database configured before, you first need to create the database by itself by running the following SQL commands into the MySQL console:

```
CREATE DATABASE oauth2provider;
CREATE USER 'oauth2provider'@'localhost' IDENTIFIED BY '123';
GRANT ALL PRIVILEGES ON oauth2provider.* TO 'oauth2provider'@'localhos
```

6. Now run the following SQL commands to start using the `oauth2provider` database and to create the table that will be used to store all the data related to the client registration:

```
create table oauth_client_details (
  client_id VARCHAR(256) PRIMARY KEY,
  resource_ids VARCHAR(256),
  client_secret VARCHAR(256),
  scope VARCHAR(256),
  authorized_grant_types VARCHAR(256),
  web_server_redirect_uri VARCHAR(256),
  authorities VARCHAR(256),
  access_token_validity INTEGER,
  refresh_token_validity INTEGER,
  additional_information VARCHAR(4096),
  autoapprove VARCHAR(256)
);
```

7. To store all the issued access tokens, create the following table:

```
create table oauth_access_token (
  token_id VARCHAR(256),
  token LONG VARBINARY,
  authentication_id VARCHAR(256) PRIMARY KEY,
  user_name VARCHAR(256),
  client_id VARCHAR(256),
  authentication LONG VARBINARY,
  refresh_token VARCHAR(256)
);
```

8. Also run the following commands so Spring Security OAuth2 can store the user's approvals and issue refresh tokens respectively:

```
create table oauth_approvals (
    userId VARCHAR(256),
    clientId VARCHAR(256),
    scope VARCHAR(256),
    status VARCHAR(10),
    expiresAt TIMESTAMP,
    lastModifiedAt TIMESTAMP
);
create table oauth_refresh_token (
    token_id VARCHAR(256),
    token LONG VARBINARY,
    authentication LONG VARBINARY
);
```

9. All the configurations that we were defining within the `OAuth2AuthorizationServer` through the `ClientDetailsServiceConfigurer` instance, can now be done by inserting a new row into the

`oauth_client_details` table by running the following SQL command:

```
INSERT INTO oauth_client_details
    (client_id, resource_ids, client_secret, scope, authorized_grant_t
    web_server_redirect_uri, authorities, access_token_validity, refre
    additional_information, autoapprove)
VALUES
    ('clientapp', null, '123456',
    'read_profile,read_posts', 'authorization_code',
    'http://localhost:9000/callback',
    null, 3000, -1, null, 'false');
```

10. To improve the client credentials protection, let's save an encrypted client secret. Create a temporary main method (in any class you want), just to generate an encrypted version of the client secret value as presented in the following code:

```
public static void main(String[] args) {
    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(4);
    String clientSecret = "123456";
    clientSecret = encoder.encode(clientSecret);
    System.out.println(clientSecret);
}
```

11. Run the preceding source code and copy the client secret that will be printed on standard output. Then, run the following update command on MySQL database console (The following update command is already using the right encrypted value for `123456` password):

```
UPDATE oauth_client_details
SET client_secret = '$2a$04$PXzjIdEJkglftLx.z9BaB.LmvgSOtOq14acON3HCWA
WHERE client_id = 'clientapp';
```

12. Create the `OAuth2AuthorizationServer` class, by injecting one `DataSource` attribute (which must come from the `javax.sql` package) and override the method presented in the following code to set up the `ClientDetailsServiceConfigurer` to use the present `dataSource` attribute:

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends
        AuthorizationServerConfigurerAdapter {

    @Autowired
    private DataSource dataSource;

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) thro
```

```
                clients.jdbc(dataSource);
            }
        }
```

13. Now declare the following beans within the `OAuth2AuthorizationServer` class to define the `TokenStore`, `ApprovalStore` and `PasswordEncoder` respectively:

```
        @Bean
        public TokenStore tokenStore() {
            return new JdbcTokenStore(dataSource);
        }
        @Bean
        public ApprovalStore approvalStore() {
            return new JdbcApprovalStore(dataSource);
        }
        @Bean
        public PasswordEncoder passwordEncoder() {
            return new BCryptPasswordEncoder(4);
        }
```

14. Just declaring the previous beans is not enough for Spring Security OAuth2 to start using `JDBC DataSource` to store tokens and approve data entries. To do so, you need to override the following `configure` methods:

```
        @Override
        public void configure(AuthorizationServerEndpointsConfigurer endpoints
                throws Exception {
            endpoints
                .approvalStore(approvalStore())
                .tokenStore(tokenStore());
        }
        @Override
        public void configure(AuthorizationServerSecurityConfigurer security)
            throws Exception {
            security.passwordEncoder(passwordEncoder());
        }
```

15. The Resource Server is similar to the others declared for the previous recipes, as presented in the following code:

```
        @Configuration
        @EnableResourceServer
        public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
            @Override
            public void configure(HttpSecurity http) throws Exception {
                http
                    .authorizeRequests()
                        .anyRequest().authenticated().and()
                    .requestMatchers()
                        .antMatchers("/api/**");
```

```
          }
      }
```

16. Create the `UserController` and `UserProfile` classes declared for the previous recipes with the same content and the application will be ready to be started.
17. Run the application through your IDE actions or by running the Maven `mvn spring-boot:run` command.

How it works...

This recipe showed you how to use a JDBC `DataSource` to allow for client details registration in a persistent manner. Besides the client details, the OAuth 2.0 Provider configured also has the capability of saving access tokens and approvals on the database as well. We did the client details configuration just by defining the injected `DataSource` with `jdbc` property of `ClientDetailsServiceConfigurer`. By doing it, Spring Security OAuth2 will configure the internal class, `JdbcClientDetailsServiceBuilder` that is in charge of creating the `JdbcClientDetailsService` which in turn implements `ClientDetailsService`. The `ClientDetailsService` interface defines just the following method that serves the purpose of retrieving the client's registration data:

```
ClientDetails loadClientByClientId(String clientId) throws ClientRegistratior
```

The `ClientDetailsService` and `ClientDetails` interfaces defines the contract required by Spring Security OAuth2 to manipulate and define client registration data respectively. Now we are left with `TokenStore` and `ApprovalStore` interfaces. The `TokenStore` defines the contract to store, retrieve, remove, and even read authentication data related to tokens (which can be access tokens or refresh tokens). The `ApprovalStore` interface defines methods to add, retrieve, and revoke the Resource Owner's approvals. All these interfaces allow for extension points that you can use to create custom storage strategies for client details and tokens within the application.

Now that you know some important extension points provided by Spring Security OAuth2, what happens when we run the application and perform the authorization and token request phases? Start the application by running the `mvn spring-boot:run` command and go to the following URL:

```
http://localhost:8080/oauth/authorize?client_id=clientapp&redirect_uri=http:/
```

The Authorization Server will ask you for your credentials to authenticate you and then you should approve the scope required by the `rdbm-server` application. After granting permission to the `clientapp` application , you will be redirected to the following URL with a different value for code parameter:

```
http://localhost:9000/callback?code=1mcN2S
```

When you as the Resource Owner had approved the `clientapp` to client access your profile, a row will be created in the `oauth_approvals` table. Run the following query in your MySQL console to check for a new row.

```
select * from oauth_approvals;
```

Now if you request for a new access token by running the following CURL command, you will receive a new access token generating a new row into the `oauth_access_token` table:

```
curl -X POST --user clientapp:123456 http://localhost:8080/oauth/token -H "cc
```

Run the following SQL query to check for a new access token inside the `oauth_access_token` table.

```
select * from oauth_access_token;
```

There's more...

When using the database as we did for this recipe, you might notice that the access token inserted into the `oauth_access_token` table is different from what you've received in response to the CURL command to request for a new access token. That's because, Spring Security OAuth2 serializes the `DefaultOAuth2AccessToken` object which represents an access token by itself.

Another thing to notice in this recipe is how the Resource Server knows we are using a database to store access tokens. To validate access tokens the Resource Server needs to check for the received access token against the persisted one on the database. Such an operation is performed by `ResourceServerTokenServices` one which relies on the current `TokenStore` available in the Spring context. As both the Resource Server and Authorization Server are running at the same context, the `TokenStore` available will be the `JdbcTokenStore` bean that we have declared within the Authorization Server configuration.

See also

- Protecting resources using the Authorization Code grant type
- Using Redis as a token store

Using Redis as a token store

This recipe will show you how to use Redis to store access tokens and also approval information. Unlike the previous recipe, we won't use Redis to store client details because this kind of data must be persistent and Redis uses a memory data structure to store data.

Getting ready

To run this recipe, you need a Redis database (make sure to set up Redis authentication for production environment) along with the other tools, such as your preferred IDE and tools that allow you to interact with the OAuth 2.0 Provider. The source code for this recipe is available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/redis-server.

How to do it...

Perform the following steps to set up Redis to store tokens:

1. As the Redis database is a prerequisite, you need to download and install Redis on your machine. Download the latest stable version from https://redis.io/download.
2. After downloading, go to the `Download` directory and run the following commands to compile Redis (notice that Redis is not officially supported for Windows, but as per Redis documents you can look for more details at https://github.com/MicrosoftArchive/redis:

```
tar xzf redis-4.0.1.tar.gz
cd redis-4.0.1
make
```

3. After compiling Redis, you are ready to run the `redis-server` and `redis-cli` commands available within the `src` Redis directory.

4. Run the `redis-server` command to make Redis available for this recipe.
5. Create the initial project using Spring Initializr, as we did for other recipes in this book. Go to https://start.spring.io/ and define the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `redis-server`
   - Add `Web`, `Redis`, and `Security` as dependencies for this project
6. After creating the `redis-server` project, import it to your IDE. If using Eclipse, import it as a Maven project.
7. Open the `pom.xml` file and add the following dependency for Spring Security OAuth2:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

8. Open the `application.properties` file and add the following content to configure the user's credentials and Redis connection:

```
security.user.name=adolfo
security.user.password=123
spring.redis.url=redis://localhost:6379
```

9. Create the `UserController` and `UserProfile` classes declared in the previous recipes with the same content into the package `com.packt.example.redisserver.api`.
10. Create the following class within the `com.packt.example.redisserver.config` package to configure the Resource Server details:

```
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests().anyRequest().authenticated()
        .and()
            .requestMatchers().antMatchers("/api/**");
    }
}
```

11. And now, create the `OAuth2AuthorizationServer` within the config sub-package with the following content:

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends
  AuthorizationServerConfigurerAdapter {

  @Autowired
  private AuthenticationManager authenticationManager;

  @Override
  public void configure(AuthorizationServerEndpointsConfigurer endpoir
      endpoints.authenticationManager(authenticationManager);
  }

  @Override
  public void configure(ClientDetailsServiceConfigurer clients)
    throws Exception {
    clients.inMemory()
        .withClient("clientapp").secret("123456")
        .authorizedGrantTypes("password", "authorization_code")
        .scopes("read_profile", "read_contacts");
 }
}
```

12. Until this point, we are configuring the Authorization Server to store both client credentials and tokens using the in-memory strategy. To start

using Redis, inject an instance of `RedisConnectionFactory` within `OAuth2AuthorizationServer`, as follows:

```
@Autowired
private RedisConnectionFactory connectionFactory;
```

13. With an instance of `RedisConnectionFactory`, we are ready to declare one bean of type `TokenStore` which uses Redis as the store strategy:

```
@Bean
public TokenStore tokenStore() {
    RedisTokenStore redis = new RedisTokenStore(connectionFactory);
    return redis;
}
```

14. Now that we have the `RedisTokenStore`, let's define the token store strategy by setting up the `AuthorizationServerEndpointsConfigurer` within the `OAuth2AuthorizationServer` class, as presented in the following code (it already has the configuration we did for `authenticationManager`):

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints
    throws Exception {
    endpoints
        .authenticationManager(authenticationManager)
        .tokenStore(tokenStore());
}
```

15. Run the application through your IDE actions or by running the Maven command, `mvn spring-boot:run`.

How it works...

Looking at the configuration we did to use Redis as the token store strategy, we just had to declare the appropriate bean, `RedisTokenStore`, which implements the `TokenStore` interface. This recipe shows you how Spring Security OAuth2 allows for extensibility by simply implementing some interfaces and using it within some `configurers`.

To check how `RedisTokenStore` persists the access tokens and related data, start the redis-server application and try to request for an access token using one of the authorized grant types declared, which are Authorization Code and Password Credentials. For practical reasons, I will make the access token request through the Password Credentials, as you can see in the following CURL command:

```
curl -X POST --user clientapp:123456 http://localhost:8080/oauth/token -H "ac
```

After running the previous command, the response should be a new access token, as follows:

```
{
    "access_token":"edf78a75-7aab-48f5-a6c5-0a0684208d74",
    "token_type":"bearer",
    "expires_in":43140,
    "scope":"read_profile"
}
```

Then run the `redis-cli` command to start exploring Redis to check if the respective key was created and after starting the Redis console, run the `keys *` command. You will see some keys related to the access token recently issued, as presented next:

```
1) "auth_to_access:643c3ce5f7876962d689f63a66a48d83"
2) "access:edf78a75-7aab-48f5-a6c5-0a0684208d74"
3) "client_id_to_access:clientapp"
4) "uname_to_access:clientapp:adolfo"
5) "auth:edf78a75-7aab-48f5-a6c5-0a0684208d74"
```

And if you want to know the value from one of the presented keys, just enter the `get <key_name>` command where the `key_name` could be

See also

- Protecting resources using the Authorization Code grant type
- Using the Resource Owner Password Credentials grant type as an approach for OAuth 2.0 migration
- Using the relational database to store tokens and client details

Implementing client registration

This recipe shows you how to use Spring Security OAuth2 to streamline the development of OAuth 2.0 client registration so that you do not have to worry about writing a code that interacts with the database. As you will learn, Spring Security OAuth2 provides an abstraction, leaving you in charge of creating just controller code and the respective views.

Getting ready

To run this recipe, we have to create an OAuth 2.0 Provider project which has at least one registered user (which is the Resource Owner), and we have to use one relational database (MySQL for this recipe) and configure it appropriately, as we did for `rdbms-server` project that was created for the recipe, *Using a relational database to store tokens and client details*.

The source code for this recipe is available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/oauth2provider.

How to do it...

Perform the following steps to create an OAuth 2.0 Provider that allows for client registration:

1. Create the initial project using Spring Initializr as we did for other recipes in this book. Go to https://start.spring.io/ and define the following data:
    - Set up the Group as `com.packt.example`
    - Define the Artifact as `oauth2provider`
    - Add `Web`, `JPA`, `MySQL`, `Security`, and `Thymeleaf` as dependencies for this project
2. After creating the `oauth2provider` project, import it to your IDE. If using Eclipse, import it as a Maven project.
3. Open the `pom.xml` file and add the following dependencies:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>3.3.5</version>
</dependency>
```

4. Add the following content into `application.properties` file to set up the user's credentials, `datasource`, and `JPA` properties, and the `Thymeleaf` property to avoid caching content (it's helpful when in development mode so you do not have to restart an application every time when something changes within the HTML files):

```
security.user.name=adolfo
security.user.password=123

spring.datasource.url=jdbc:mysql://localhost/oauth2provider
spring.datasource.username=oauth2provider
spring.datasource.password=123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Di
spring.jpa.properties.hibernate.hbm2ddl.auto=validate
spring.thymeleaf.cache=false
```

5. After setting up the properties, create the `UserController` and `UserProfile` classes within the `com.packt.example.oauth2provider.api` package, the same way as we did for other recipes in this chapter (look at the first recipe or go to GitHub to see the contents of these classes). These classes will provide the OAuth 2.0 protected API.
6. Then create the Resource Server and Authorization Server configuration, the same way we did for the recipe *Using a relational database to store tokens and client details*. But for didactical purposes, remove the client secret encryption configuration within the Authorization Server.
7. After creating the `OAuth2AuthorizationServer` class with the same content from the `rdbms-server` project, add the following bean declaration at the end of the class body:

```
    @Bean
    public ClientRegistrationService  clientRegistrationService() {
        return new JdbcClientDetailsService(dataSource);
    }
```

8. Create the `com.packt.example.oauth2provider.client` package that will hold all the classes related to the client registration.

9. To define the client type that someone can register at `oauth2provider` application, let's create the following `enum` class within the previously created package:

```
public enum ClientType {
    PUBLIC, CONFIDENTIAL
}
```

10. As we want to provide a way for client registration, we need to map all the attributes to an entity to be persisted and retrieved from the database. When using Spring Security OAuth2, we have to implement the `ClientDetails` interface so that `ClientRegistrationServer` can recognize the entity we are talking about. Create the `Application` class within the previous created package, implementing the `ClientDetails` interface, as presented in the following code:

```
public class Application implements ClientDetails {
}
```

11. Notice that we have to implement a lot of methods from the `ClientDetails` interface. Each method will return data that will be provided by the following properties that you must add at the beginning of the `Application` class body:

```
public class Application implements ClientDetails {
    private String clientId;
    private String clientSecret;
    private ClientType clientType;
    private Set<String> resourceIds = new HashSet<>();
    private Set<String> scope = new HashSet<>();
    private Set<String> webServerRedirectUri = new HashSet<>();
    private int accessTokenValidity;
    private Map<String, Object> additionalInformation = new HashMap<>(
    // all methods hidden
}
```

12. Create the setters presented in the following code, so that we can set up all the values needed to register client details. The setters presented must be created within the `Application` class:

```
public void setName(String name) {
    additionalInformation.put("name", name);
}
public void setClientType(ClientType clientType) {
    additionalInformation.put("client_type", clientType.name());
}
public void setClientId(String clientId) {
    this.clientId = clientId;
}
public void setClientSecret(String clientSecret) {
    this.clientSecret = clientSecret;
}
public void setAccessTokenValidity(int accessTokenValidity) {
    this.accessTokenValidity = accessTokenValidity;
}
```

13. Now add the following methods to allow us to set up collection-based attributes:

```
public void addRedirectUri(String redirectUri) {
    this.webServerRedirectUri.add(redirectUri);
}
public void addScope(String scope) {
    this.scope.add(scope);
}
public void addResourceId(String resourceId) {
    this.resourceIds.add(resourceId);
}
```

14. To finish the `Application` class creation, make sure the getters looks like the following:

```
    public String getClientId()
    { return clientId; }
    public Set<String> getResourceIds()
    { return resourceIds; }
    public boolean isSecretRequired()
    { return clientType == ClientType.CONFIDENTIAL; }
    public String getClientSecret()
    { return clientSecret; }
    public boolean isScoped()
    { return scope.size() > 0; }
    public Set<String> getScope()
    { return scope; }
    public Set<String> getRegisteredRedirectUri()
    { return webServerRedirectUri; }
    public Collection<GrantedAuthority> getAuthorities()
    { return new HashSet<>(); }
    public Integer getAccessTokenValiditySeconds()
    { return accessTokenValidity; }
    public Integer getRefreshTokenValiditySeconds()
    { return null; }
    public boolean isAutoApprove(String scope)
    { return false; }
    public Map<String, Object> getAdditionalInformation()
    { return additionalInformation; }
    public Set<String> getAuthorizedGrantTypes()
    {
        Set<String> grantTypes = new HashSet<>();
        grantTypes.add("authorization_code");
        grantTypes.add("refresh_token");
        return grantTypes;
    }
```

15. Before creating the controller that will manage the client registration flow, create the `BasicClientInfo` class as presented in the following code. This class maps all attributes to the registration form fields so that we can effectively persist the client information (I have omitted getters and setters just for brevity, but you must declare them):

```
    public class BasicClientInfo {
        private String name;
        private String redirectUri;
        private String clientType;
       // getters and setters hidden
    }
```

16. Now, let's create the main class for this recipe, which is `ClientController`, as presented in the following code (this class must be created in the same package as `BasicClientInfo`, `Application`, and `ClientType`:

```
@Controller
@RequestMapping("/client")
public class ClientController {
    @Autowired
    private ClientRegistrationService clientRegistrationService;
}
```

17. Notice that the registration endpoints will be rooted by the `/client` path. All the actions will be mapped alongside the `/client` path (also notice that we are not creating RESTful endpoints because that's not the purpose of the recipe). Next, the following steps will present you with all the actions that the `ClientController` class will provide and the respective view (which is HTML files). Add the following method that will be responsible for returning the registration view to the web browser:

```
@GetMapping("/register")
public ModelAndView register(ModelAndView mv) {
    mv.setViewName("client/register");
    mv.addObject("registry", new BasicClientInfo());
    return mv;
}
```

18. Create the `register.html` file inside the `src/main/resources/templates/client` folder with the following content:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
<title>oauth2server</title>
<link href="../webjars/bootstrap/3.3.5/css/bootstrap.min.css"
  rel="stylesheet" media="screen"></link>
<link href="/bootstrap-select.min.css" rel="stylesheet"></link>
</head>
<body>
  <div class="container">
    <div class="jumbotron">
      <h1>OAuth2 Provider</h1>
    </div>
    <h2>Create your application (client registration)</h2>
    <form action="#" th:action="@{/client/save}" th:object="${registry
      method="post">
      <div class="form-group">
        <label for="nome">Name:</label> <input class="form-control"
          id="name" type="text" th:field="*{name}" />
        <div th:if="${#fields.hasErrors('name')}" th:errors="*{name}">
          name</div>
      </div>
      <div class="form-group">
        <label for="redirectUri">Redirect URL:</label> <input
          class="form-control" id="redirectUri" type="text"
          th:field="*{redirectUri}" />
        <div th:if="${#fields.hasErrors('redirectUri')}"
          th:errors="*{redirectUri}">Callback URL to receive the
          authorization code</div>
      </div>
      <div class="form-group">
        <label for="clientType">Type of application:</label>
        <div>
          <select id="clientType" class="selectpicker"
            th:field="*{clientType}">
            <option value="PUBLIC">Public</option>
            <option value="CONFIDENTIAL">Confidential</option>
          </select>
        </div>
      </div>
      <div class="form-group">
        <button class="btn btn-primary" type="submit">Register</buttor
        <button class="btn btn-default" type="button"
          onclick="javascript: window.location.href='/'">Cancel</buttc
      </div>
    </form>
  </div>
</body>
<script src="/jquery.min.js"></script>
<script src="/bootstrap-select.min.js"></script>
<script src="../webjars/bootstrap/3.3.5/js/bootstrap.min.js"></script>
</html>
```

19. Notice that we need some assets such as CSS and JavaScript files. Download the JavaScript libraries that I have provided at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/oauth2provider/src/main/resources/static and copy each of these files to the `src/main/resources/static` folder of the `oauth2provider` project.
20. To receive the client details information provided by the user, create the following method at the end of the `ClientController` class body:

```
@PostMapping("/save")
public ModelAndView save(@Valid BasicClientInfo clientDetails, Bindinc
    if (bindingResult.hasErrors()) { return new ModelAndView("client/r
    Application app = new Application();
    app.setName(clientDetails.getName());
    app.addRedirectUri(clientDetails.getRedirectUri());
    app.setClientType(ClientType.valueOf(clientDetails.getClientType()
    app.setClientId(UUID.randomUUID().toString());
    app.setClientSecret(UUID.randomUUID().toString());
    app.setAccessTokenValidity(3000);
    app.addScope("read_profile");
```

```
        app.addScope("read_contacts");
        clientRegistrationService.addClientDetails(app);

        ModelAndView mv = new ModelAndView("redirect:/client/dashboard");
        mv.addObject("applications", clientRegistrationService.listClientD
        return mv;
    }
```

21. Add the following method to allow the user to remove a registered application:

```
    @GetMapping("/remove")
    public ModelAndView remove(@RequestParam(value = "client_id", requirec
    {
        clientRegistrationService.removeClientDetails(clientId);
        ModelAndView mv = new ModelAndView("redirect:/client/dashboard");
        mv.addObject("applications", clientRegistrationService.listClientD
        return mv;
    }
```

22. And to allow the logged user to see all registered applications, create the following method inside `ClientController`:

```
    @GetMapping("/dashboard")
    public ModelAndView dashboard(ModelAndView mv) {
        mv.addObject("applications", clientRegistrationService.listClientD
        return mv;
    }
```

23. Now, create the HTML file named `dashboard.html` inside `src/main/resources/templates/client` to present all the registered clients:

```
    <!DOCTYPE html>
    <html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>oauth2server</title>
        <link href="../webjars/bootstrap/3.3.5/css/bootstrap.min.css"
            rel="stylesheet" media="screen"></link>
        <link href="/bootstrap-select.min.css" rel="stylesheet"></link>
    </head>
    <body>
    <div class="container">
        <div class="jumbotron"><h1>OAuth2 Provider</h1></div>
        <h2>Registered applications</h2>
        <div th:if="${applications != null}">
            <table class="table">
                <tr>
                    <td>Application name</td>
                    <td>client type</td>
                    <td>client ID</td>
                    <td>client secret</td>
                    <td>Delete app</td>
                </tr>
                <tr th:each="app : ${applications}">
                    <td th:text="${app.additionalInformation['name']}"></t
                    <td th:text="${app.additionalInformation['client_type'
                    <td th:text="${app.clientId}">client_id</td>
                    <td th:text="${app.clientSecret}">client_secret</td>
                    <td><a class="btn btn-danger" href="#"
                       th:href="@{/client/remove(client_id=${app.clientId}
                </tr>
            </table>
        </div>
        <a class="btn btn-default" href="/client/register">Create a new ap
    </div>
    </body>
    <script src="/jquery.min.js"></script>
    <script src="/bootstrap-select.min.js"></script>
    <script src="../webjars/bootstrap/3.3.5/js/bootstrap.min.js"></script>
    </html>
```

24. Before trying to run the application, make sure you have created the `oauth2provider` database with the respective tables. If you have not created the database yet, you can run the SQL commands available at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/blob/master/Chapter02/oauth2provider/database.sql.
25. The application now is ready to be executed by running the `mvn spring-boot:run` command or through the IDE.

How it works...

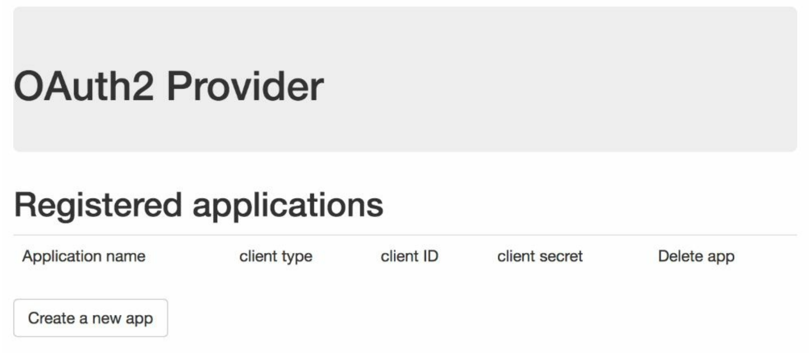
This recipe presents you with a classic Create, Retrieve, Update, and Delete (CRUD) application for users to register applications the same way Facebook and other providers do. But the most important concept to be considered here is the `ClientRegistrationService` interface. By injecting this Spring Security OAuth2 provided bean, we do not need to know how to persist and retrieve client details from the database at all. In addition, we can also switch the type of database to use without modifying the controller by itself. That's what gives flexibility and extensibility for Spring projects. Notice that we are using just methods to add, remove, and retrieve client details, but the `ClientRegistrationService` interface also provides two more methods for updating client details and client secrets:

```
public interface ClientRegistrationService {
 void addClientDetails(ClientDetails clientDetails) throws ClientAlreadyExist
 void updateClientDetails(ClientDetails clientDetails) throws NoSuchClientExc
 void updateClientSecret(String clientId, String secret) throws NoSuchClientE
 void removeClientDetails(String clientId) throws NoSuchClientException;
 List<ClientDetails> listClientDetails();
}
```

But you may ask how the service we are wiring to `ClientController` knows how to use the JDBC interface to interact with the database that we have created before? That's because of the bean declaration we have done within `OAuth2AuthorizationServer`, as follows:

```
@Bean
public ClientRegistrationService clientRegistrationService() {
    return new JdbcClientDetailsService(dataSource);
}
```

To see the client registration process in action, start the application and go to `http://localhost:8080/client/dashboard` so you can see the following page (the application will try to authenticate you, so enter the user credentials that you have set up within the `application.properties` file):

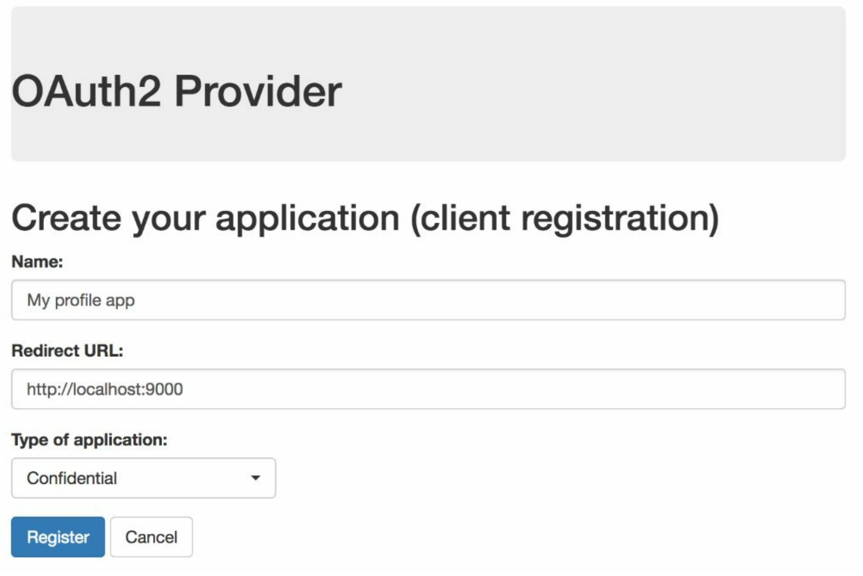


OAuth2 Provider

Registered applications

Application name	client type	client ID	client secret	Delete app

Then click on Create a new app and fill out the form content with relevant data (you can use the same client details information presented in the following screenshot):



OAuth2 Provider

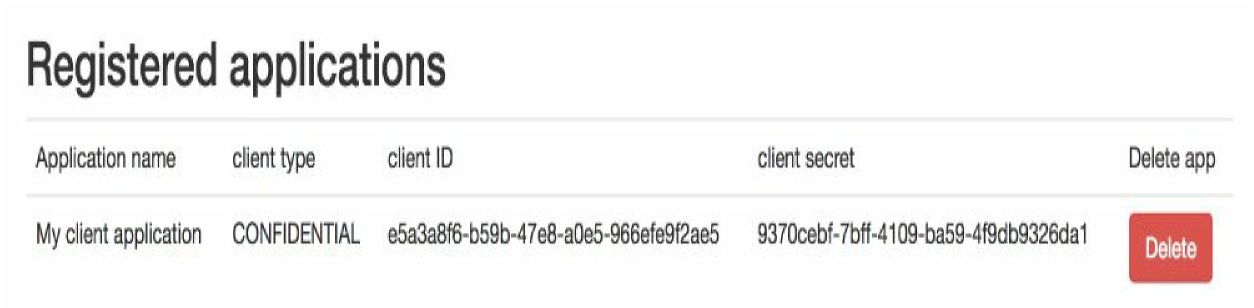
Create your application (client registration)

Name: My profile app

Redirect URL: http://localhost:9000

Type of application: Confidential

Then click on Register and you will be redirected to the dashboard with the new client registered details with client ID and client secret information available, as follows:



Registered applications

Application name	client type	client ID	client secret	Delete app
My client application	CONFIDENTIAL	e5a3a8f6-b59b-47e8-a0e5-966efe9f2ae5	9370cebf-7bff-4109-ba59-4f9db9326da1	Delete

Now you can copy the client ID and client secret to start requesting an access token and use it to retrieve the user's profile data the same way we did for the other recipes. Just notice that we are providing the Authorization Code grant type and supporting refresh tokens. You will notice that by looking at the `getAuthorizedGrantTypes` method from the `Application` class:

```
public Set<String> getAuthorizedGrantTypes() {
    Set<String> grantTypes = new HashSet<>();
    grantTypes.add("authorization_code");
    grantTypes.add("refresh_token");
    return grantTypes;
}
```

See also

- Protecting resources using the Authorization Code grant type
- Using relational database to store tokens and client details

Breaking the OAuth 2.0 Provider in the middle

This recipe will show how you can create an OAuth 2.0 Provider, dividing Authorization Server and Resource Server responsibilities into different projects.

Getting ready

To run this recipe, you will need to create two different applications to implement the Authorization Server and the Resource Server responsibilities. As the Authorization Server is in charge of issuing access tokens and Resource Server has to validate access tokens, both applications need a shared database. In that way, the Resource Server can query for an access token that was persisted in some database structure by the Authorization Server. For this recipe, you will need Redis as the strategy for database sharing.

How to do it...

Perform the following steps to start creating the Authorization Server and Resource Server separately:

1. The first thing to do is to create both applications. Go to Spring Initializr and generate the project for the Authorization Server named (Artifact) as `authorization-server` and generate another project for the Resource Server named (Artifact) as `resource-server`. With Spring Initializr, add `Web`, `Security`, and `Redis` as dependencies for both projects (the group name is up to you but I am using `com.packt.example`).
2. After generating those projects, unpack them and import them to your IDE (if you are using Eclipse, you can import it as a Maven project).
3. Now open the `pom.xml` file from both applications and add the following dependency for Spring Security OAuth2:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. Open the `application.properties` file from the `authorization-server` project and add the following content:

```
security.user.name=adolfo
security.user.password=123
spring.redis.url=redis://localhost:6379
```

5. Open the `application.properties` file from the `resource-server` project and add the following content (as we are running two applications at the same time, they must listen to different server ports):

```
server.port=8081
spring.redis.url=redis://localhost:6379
```

6. To configure the Authorization Server, we don't have to write too much code. Just create the `OAuth2AuthorizationServer` class with the following content inside `authorization-server` project (notice that we are using `Redis` as the `TokenStore`):

```
    @Configuration @EnableAuthorizationServer
    public class OAuth2AuthorizationServer extends AuthorizationServerConf
        @Autowired private RedisConnectionFactory connectionFactory;
        @Autowired private AuthenticationManager authenticationManager;
        public void configure(AuthorizationServerEndpointsConfigurer endpo
            endpoints
                .authenticationManager(authenticationManager)
                .tokenStore(tokenStore());
        }
        @Bean public TokenStore tokenStore() {
            return new RedisTokenStore(connectionFactory);
        }
        public void configure(ClientDetailsServiceConfigurer clients) thro
            clients.inMemory()
                .withClient("clientapp").secret("123456")
                .redirectUris("http://localhost:9000/callback")
                .authorizedGrantTypes("authorization_code" ,"password")
                .scopes("read_profile", "read_contacts");
        }
    }
```

7. If you want to test the Authorization Server, the application is ready to be executed, but do not forget to start Redis before running the Authorization Server.
8. Now we have to create the API that will be protected by OAuth 2.0 within the `resource-server` application. To do so, create the `UserController` and `UserProfile` classes the same way we did for the first recipe. You can create these classes within the `com.packt.example.resourceserver.api` package.

9. And now, create the `OAuth2ResourceServer` class inside the `com.packt.example.resourceserver.config` package from `resourece-server` project, with the following content:

```
    @Configuration @EnableResourceServer
    public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
        @Autowired
        private RedisConnectionFactory connectionFactory;
        @Override
        public void configure(ResourceServerSecurityConfigurer resources)
            resources.tokenStore(tokenStore());
        }
        @Bean
        public TokenStore tokenStore() {
            return new RedisTokenStore(connectionFactory);
        }
        @Override
        public void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests().anyRequest().authenticated()
            .and()
                .requestMatchers().antMatchers("/api/**");
        }
    }
```

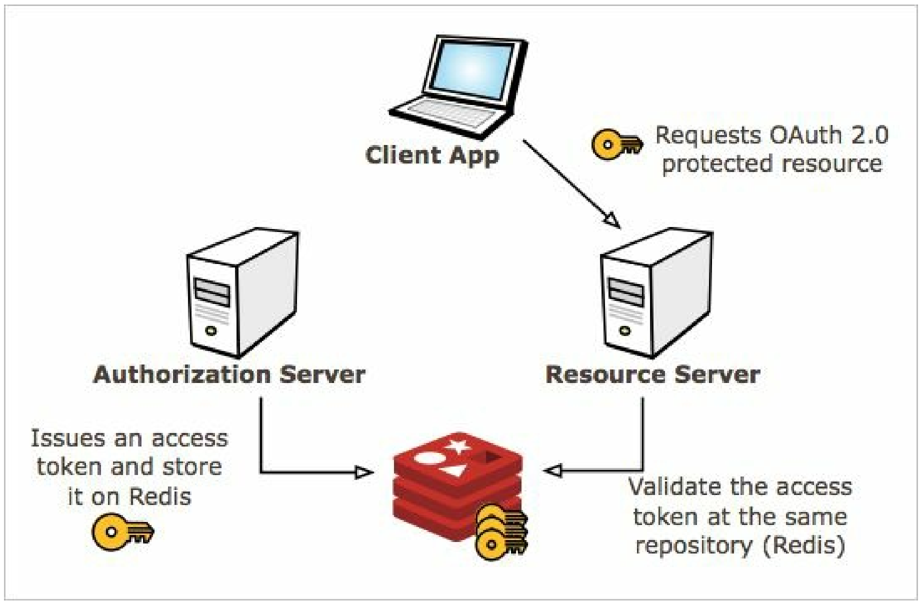
10. After creating the previous configuration, the Resource Server is ready to be executed. If you have retrieved an access token from the running Authorization Server, you can access the user's profile by sending a request using port `8081`, as presented in the following command:

```
        curl -X GET http://localhost:8081/api/profile -H "authorization: Beare
```

How it works...

By running both applications, that is, the Authorization Server and Resource Server projects, at the same time, we can request for the access token for the application that's running on port `8080` and we can access the user's profile by sending a request to port `8081`. It's recommended to run the OAuth2 Provider broken into two applications like we did for this recipe so we can decrease the surface attack. Moreover, we can scale applications separately depending on specific needs.

It's really important to notice that for this architecture to work, we need to use database sharing so the access tokens issued by the Authorization Server can be queried by the Resource Server when validating access tokens. Take a look at the following image that depicts our current architecture:


Client App
Requests OAuth 2.0 protected resource
Authorization Server
Resource Server
Issues an access token and store it on Redis
Validate the access token at the same repository (Redis)

Bear in mind that in the same way we are using Redis as the shared database, you can also use any other NoSQL or relational database that are more appropriate for your scenario. Consider the performance when comparing between databases, because depending on how much the Resource Server will be validating access tokens, you might need to scale up as is appropriate.

In previous recipes that use databases to store access tokens, we were configuring just the Authorization Server to know how to store access tokens. But now that we are creating the Resource Server as a separate application, how do you configure this application to know where to look for access tokens so the Resource Server can evaluate any given access token?

Look at the following code that is present on the Resource Server configuration class:

```
@Autowired
private RedisConnectionFactory connectionFactory;

@Override
public void configure(ResourceServerSecurityConfigurer resources) throws Exce
    resources.tokenStore(tokenStore());
}

@Bean
public TokenStore tokenStore() {
    return new RedisTokenStore(connectionFactory);
}
```

As you can see, we are configuring a `TokenStore` the same way we did for the Authorization Server configuration. The difference is that we are setting the token store to an instance of `ResourceServerSecurityConfigurer`.

See also

- Protecting resources using the Authorization grant type
- Using the relational database to store tokens and client details
- Using Redis as a token store

Using Gatling to load test the token validation process using shared databases

This recipe shows you how to evaluate the performance of a chosen architecture for the OAuth 2.0 Provider created for the last recipe that relies on shared databases as a `TokenStore`.

Getting ready

To run this recipe, you will have to use Gatling to create one load testing project. Gatling relies on the Scala language, but don't worry about it because I have already prepared a base project with Maven that will be helpful for this recipe. Afterwards, we need some IDE to edit all the files appropriately. You can use the Scala IDE for Eclipse, which is available at http://scala-ide.org/ or you can use IntelliJ (to use IntelliJ you must install the Scala plugin by navigating to Preferences/Plugins). To run this recipe, we will also need the applications created for the last recipe to be running.

How to do it...

Perform the following steps to create and load test the OAuth 2.0 Provider that uses Redis as a shared token storage:

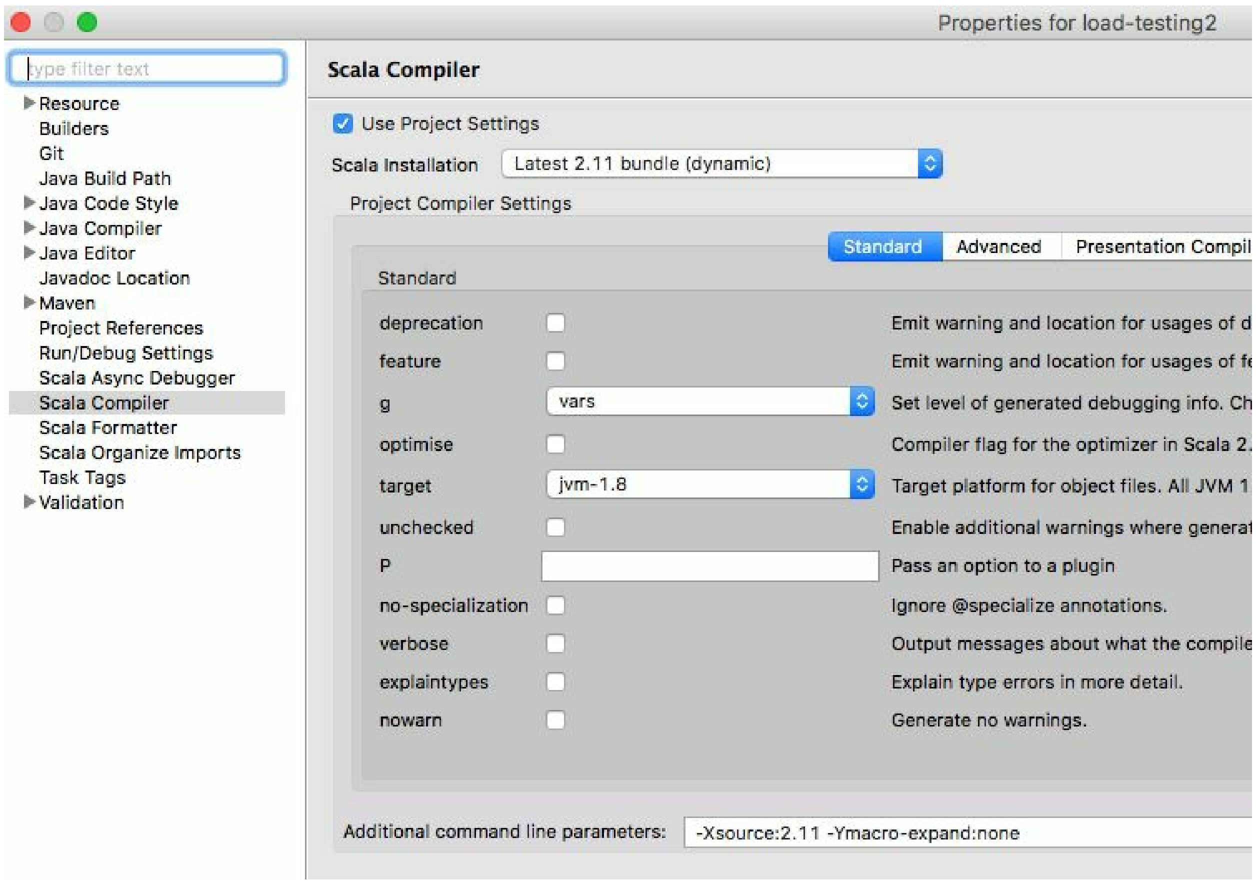
1. Clone or download the project available at https://github.com/adolfoweloy/scala-maven-skel. As you will notice, this project is just a skeleton for the recipe that we will be creating now. If you want to clone this project using GitHub, just run the following command:

```
git clone git@github.com:adolfoweloy/scala-maven-skel
```

2. Now rename the `scala-maven-skel` project to load-testing by running the following command (this command works for Linux or macOS X, so if using Windows just use the user interface):

```
mv scala-maven-skel load-testing
```

3. Go to the `load-testing` directory and open `pom.xml` with any editor you want (I recommend using vim or any lightweight editor before importing the project).
4. Make sure the `artifactId` tag has the name `load-testing` instead of `scala-maven-skel`.
5. Now open the Scala IDE for Eclipse and import the project as a Maven project (the same way we import projects to Eclipse when using Java).
6. After importing the project, you might notice some errors because of language compatibility. Open the project's preferences and click on the Scala Compiler, as presented in the following screenshot:



Scala Compiler

☑ Use Project Settings

Scala Installation: Latest 2.11 bundle (dynamic)

Project Compiler Settings

Standard

deprecation	☐	Emit warning and location for usages of d
feature	☐	Emit warning and location for usages of fe
g	vars	Set level of generated debugging info. Ch
optimise	☐	Compiler flag for the optimizer in Scala 2.
target	jvm-1.8	Target platform for object files. All JVM 1
unchecked	☐	Enable additional warnings where generat
P		Pass an option to a plugin
no-specialization	☐	Ignore @specialize annotations.
verbose	☐	Output messages about what the compile
explaintypes	☐	Explain type errors in more detail.
nowarn	☐	Generate no warnings.

Additional command line parameters: -Xsource:2.11 -Ymacro-expand:none

7. Then check the Use Project Settings option and choose Scala Installation to be Latest 2.11 bundle (dynamic), as presented in the previous screenshot.
8. Click on the Apply button, then OK.
9. Now that the compatibility problems have gone, open the `pom.xml` file and add the following dependencies for Gatling:

```
        <dependency>
            <groupId>io.gatling</groupId>
            <artifactId>gatling-app</artifactId>
            <version>2.2.5</version>
        </dependency>
        <dependency>
            <groupId>io.gatling</groupId>
            <artifactId>gatling-recorder</artifactId>
            <version>2.2.5</version>
        </dependency>
        <dependency>
            <groupId>io.gatling.highcharts</groupId>
            <artifactId>gatling-charts-highcharts</artifactId>
            <version>2.2.5</version>
        </dependency>
```

10. Add the following dependencies because we need to interact with OAuth 2.0 Provider endpoints:

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.3</version>
</dependency>
<dependency>
    <groupId>org.json4s</groupId>
    <artifactId>json4s-native_2.11</artifactId>
    <version>3.5.2</version>
</dependency>
```

11. And, finally, to be able to run Gatling load tests using Maven, add the following plugin inside the `plugin` tag within your `pom.xml` file:

```
<plugin>
 <groupId>io.gatling</groupId>
 <artifactId>gatling-maven-plugin</artifactId>
 <version>2.2.4</version>
 <executions>
  <execution>
   <goals>
    <goal>execute</goal>
   </goals>
  </execution>
 </executions>
</plugin>
```

12. The load test we are creating will interact with the OAuth 2.0 Provider automatically instead of the using CURL or Postman tools. For practical reasons, we will use the Resource Owner Password Credentials grant type because if we use Authorization grant type, we would need more sophisticated configuration. To retrieve access tokens using the Password grant type, create a Scala object named `OAuth` within `oauth2` package inside the `src/main/scala` directory. Make sure this object has the following content:

```
object OAuth {
  def getToken(): String = {
    val url = "http://localhost:8080/oauth/token"
    val auth = java.util.Base64.getEncoder()
      .encodeToString("clientapp:123456".getBytes("UTF-8"));
    val post = new HttpPost(url)
    post.addHeader("Content-Type", "application/x-www-form-urlencoded"
    post.addHeader("Authorization", "Basic " + auth)
    val client = new DefaultHttpClient
    val attributes = new ArrayList[NameValuePair](1)
```

```
        attributes.add(new BasicNameValuePair("grant_type", "password"))
        attributes.add(new BasicNameValuePair("username", "adolfo"))
        attributes.add(new BasicNameValuePair("password", "123"))
        attributes.add(new BasicNameValuePair("scope", "read_profile"))
        post.setEntity(new UrlEncodedFormEntity(attributes))
        val response = client.execute(post)

        val responseString = new BasicResponseHandler().handleResponse(res
        responseString
    }
}
```

13. Don't forget to add the following import statements to the recently created `OAuth` object:

```
import java.io._
import org.apache.commons._
import org.apache.http._
import org.apache.http.client._
import org.apache.http.client.methods.HttpPost
import org.apache.http.impl.client.DefaultHttpClient
import java.util.ArrayList
import org.apache.http.message.BasicNameValuePair
import org.apache.http.client.entity.UrlEncodedFormEntity
import org.apache.http.impl.client.BasicResponseHandler
```

14. Now, inside `src/test/scala`, create the `oauth2provider` package, then create the `AuthorizationScenarios` object within the newly created package with the following content:

```
import oauth2.OAuth
import io.gatling.core.Predef._
import io.gatling.http.Predef._
import org.json4s.DefaultFormats
import org.json4s.native.JsonMethods.parse
object AuthorizationScenarios {
  case class Token(access_token:String)
  implicit val formats = DefaultFormats
  val jsValue = parse(OAuth.getToken())
  val accessToken = "Bearer " + jsValue
    .extract[Token].access_token
  var scenario1 = scenario("Validation of the access token (strategy1)
      .exec(http("Validate access token scenario 1")
        .get("/api/profile")
        .header("Authorization", accessToken)
      )
}
```

15. Then we must create a simulation, as follows, by using the previously created scenario:

```
import scala.concurrent.duration._
import io.gatling.core.Predef._
import io.gatling.http.Predef._
import oauth2.OAuth
import org.json4s.DefaultFormats
import org.json4s.native.JsonMethods.parse

class ValidationSimulation extends Simulation {
  val httpConf = http.baseURL("http://localhost:8081")
  val authorizationScenarios = List(
    AuthorizationScenarios.scenario1.inject(rampUsers(100) over(10 sec
    setUp(authorizationScenarios).protocols(httpConf)
}
```

16. In the next step, you just need to create the `gatling.conf` file with the following content. Notice that this file must be created inside the `src/test/resources` project's folder:

```
gatling {
  core {
    outputDirectoryBaseName = "result"
    runDescription = "Token validation using Redis as shared database"
    encoding = "utf-8"
    simulationClass = "oauth2provider.ValidationSimulation"
    directory {
      data = src/test/resources
      results = target/gatling
      bodies = src/test/resources
      binaries = target/classes
    }
  }
  charting {
    indicators {
      lowerBound = 5
      higherBound = 10
    }
  }
}
```

17. Now go to your terminal and run the `mvn gatling:execute` command to start load testing the application (make sure that both the Authorization Server and the Resource Server are running).

How it works...

After creating the project for load testing and having everything configured, the first thing we did was to create an object called `OAuth`. We have created this object as a helper to allow us to retrieve an access token before running the load test, which, in this case, has the purpose of sending requests to `/api/profile` so you could evaluate the response times for access token validation. As you might notice, we are using some Java classes besides using Scala (that's because Scala runs on the JVM).

The next step we did was to create an object to describe the scenario which basically creates an HTTP request to the `/api/profile` endpoint using the retrieved access token from the Authorization Server that we had configured before. The following snippet of code shows how this HTTP request is being created:

```
var scenario1 = scenario("Validation of the access token (strategy1)")
    .exec(http("Validate access token scenario 1")
    .get("/api/profile")
    .header("Authorization", accessToken)
)
```

Finally, we have created one simulation (notice that `ValidationSimulation` extends `Simulation`) for the previously declared scenario describing how to run the load test with the following strategy:

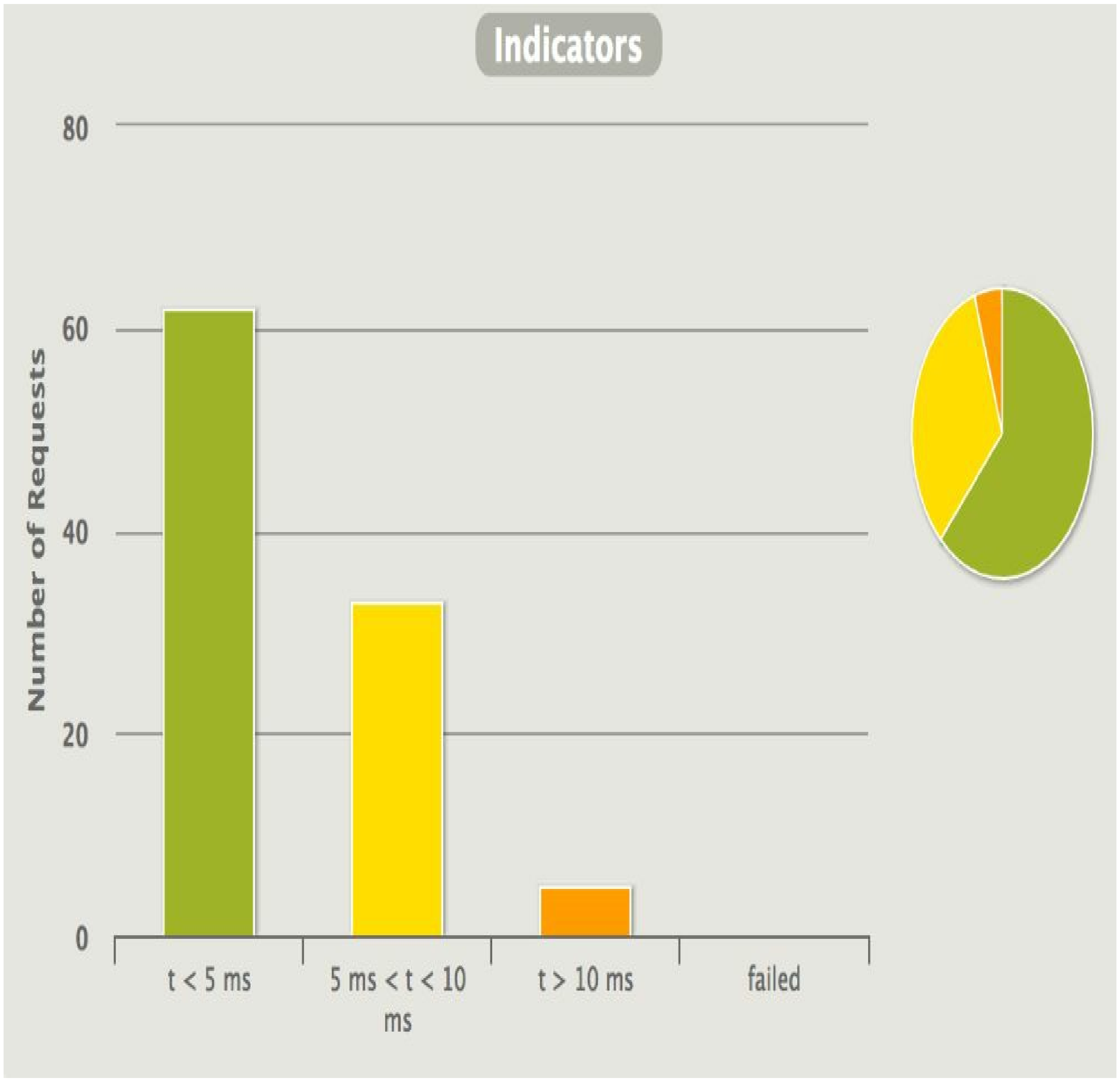
```
AuthorizationScenarios.scenario1.inject(rampUsers(100) over(10 seconds)))
```

The strategy described, basically tells Gatling to run all the scenarios for 100 users in 10 seconds. So these 100 users (users here means we are simulating requests from the client) will progressively send requests every second until 10 seconds have passed. By using this configuration, we are telling Gatling to send 10 requests every second for 10 seconds.

In addition to the code written for this recipe, you also had to create the configuration file that describes some directory information for Gatling to know how to run all the application. Also notice that we have defined some boundaries for response times that we are considering for this simulation. The lower bound was 5 and the higher bound was 10. These boundaries means the response time boundaries in seconds. After running `mvn gatling:execute` on my machine, for example, I got the following result on my console output:

```
================================================================================
---- Global Information --------------------------------------------------------
> request count 100 (OK=100 KO=0 )
> min response time 2 (OK=2 KO=- )
> max response time 22 (OK=22 KO=- )
> mean response time 5 (OK=5 KO=- )
> std deviation 3 (OK=3 KO=- )
> response time 50th percentile 4 (OK=4 KO=- )
> response time 75th percentile 5 (OK=5 KO=- )
> response time 95th percentile 10 (OK=10 KO=- )
> response time 99th percentile 18 (OK=18 KO=- )
> mean requests/sec 10 (OK=10 KO=- )
---- Response Time Distribution ------------------------------------------------
> t < 5 ms 62 ( 62%)
> 5 ms < t < 10 ms 33 ( 33%)
> t > 10 ms 5 ( 5%)
> failed 0 ( 0%)
================================================================================
```

All of these results can also be read by opening the generated report presented at the end of the execution. If you open this report on your web browser, you might see the following graphics:



This graphic report is just one of the many others that you can evaluate when using Gatling.

See also

- Using the Resource Owner Password Credentials grant type as an approach for OAuth 2.0 migration
- Using Redis as token store
- Breaking the OAuth 2.0 Provider in the middle

Using OAuth 2.0 Protected APIs

In this chapter, we will cover the following recipes:

- Creating an OAuth 2.0 client using the Authorization Code grant type
- Creating an OAuth 2.0 client using the Implicit grant type
- Creating an OAuth 2.0 client using the Resource Owner Password Credentials grant type
- Creating an OAuth 2.0 client using the Client Credentials grant type
- Managing refresh tokens on the client side
- Accessing an OAuth 2.0 protected API with RestTemplate

Introduction

The last chapter presented you with how to create your own OAuth 2.0 Provider by implementing both the Authorization Server and Resource Server. All the interactions with the OAuth 2.0 Provider were made using direct requests through the CURL command-line tool. This chapter will show you how to create client applications using Spring Security OAuth2 and the `RestTemplate` interface. As this chapter is focused on client applications (that is, the third-party application that can be granted permissions by the Resource Owner), you will also see how to manage refresh tokens at the client side.

All recipes in this chapters does not use TLS/SSL when interacting with the OAuth Provider just for didactical purposes. When running in production, you must use TLS/SSL to protect every communication between the Client and the OAuth Provider.

Creating an OAuth 2.0 client using the Authorization Code grant type

This recipe presents you with how to create a client application that interacts with the OAuth 2.0 Provider by using the Authorization Code grant type.

Getting ready

To run this recipe, make sure you have an OAuth 2.0 Provider running on your machine. I recommend you to use the project `auth-code-server` presented in the first recipe from Chapter 2, *Implementing Your Own OAuth 2.0 Provider* which is available at GitHub through https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/auth-code-server. To have an OAuth 2.0 Provider running, you will also need Java 8, Maven, and your preferred IDE.

How to do it...

The next steps will present with you how to create the client application that uses the Authorization Code grant type to retrieve an access token and to interact with the user's profile endpoint:

1. Create the project using Spring Initializr. Go to https://start.spring.io/ and fill out the form using the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `client-authorization-code`
   - Add `Web`, `Security`, `Thymeleaf`, `JPA` and `MySQL` as dependencies for this project (you can choose each of these dependencies through the Spring Initializr main's page)
2. After creating the `client-authorization-code` project, import it to your IDE. If using Eclipse, import it as a Maven project.
3. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. The client application we are writing for this recipe needs to store access token related information and needs to create an association between the access token and the current logged user. To allow it, run the following SQL commands:

```
CREATE DATABASE clientdb;
CREATE USER 'clientdb'@'localhost' IDENTIFIED BY '123';
GRANT ALL PRIVILEGES ON clientdb.* TO 'clientdb'@'localhost';
use clientdb;
create table client_user(
    id bigint auto_increment primary key,
    username varchar(100),
    password varchar(50),
    access_token varchar(100) NULL,
    access_token_validity datetime NULL,
    refresh_token varchar(100) NULL
);
insert into client_user (username, password) value ('aeloy', 'abc');
```

5. Now, open the `application.properties` file and add the following content to configure the data source and to add a custom port for this application to run at the same time as the OAuth Provider:

```
server.port=9000
spring.http.converters.preferred-json-mapper=jackson
spring.datasource.url=jdbc:mysql://localhost/clientdb
spring.datasource.username=clientdb
spring.datasource.password=123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Di
spring.jpa.properties.hibernate.hbm2ddl.auto=validate
```

6. This client application will provide a web interface to allow the user to see some fictitious data and her profile (which is stored within the OAuth Provider). There will be two simple pages: the first one will just provide one link to allow the user to navigate to the profile's page, and the second will be the page which shows the user's profile. Let's create the first web page as `index.html` inside the `src/main/resources/templates` directory:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymel
<head><title>client app</title></head>
<body><a href="/dashboard">Go to your dashboard</a></body>
</html>
```

7. Then create the file `dashboard.html` within `src/main/resources/templates` with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymel
<head><title>client app</title></head>
<body>
  <h1>that's your dashboard</h1>
  <table>
    <tr><td><b>That's your entries</b></td></tr>
    <tr th:each="entry : ${user.entries}">
      <td th:text="${entry.value}">value</td>
    </tr>
  </table>
  <h3>your profile from [Profile Application]</h3>
  <table>
    <tr>
        <td><b>name</b></td>
        <td th:text="${profile.name}">username</td>
    </tr>
    <tr>
        <td><b>email</b></td>
        <td th:text="${profile.email}">email</td>
    </tr>
  </table>
</body>
</html>
```

8. To support the flow that will be managed by one controller, we need to create some base classes. So create the class `Entry` within the package `com.packt.example.clientauthorizationcode.user` as follows (this class will be used to represent some user's fictitious data):

```
public class Entry {
    private String value;
    public Entry(String value)
     { this.value = value; }
    public String getValue()
     { return value; }
}
```

9. Create the class `ClientUser` with the following content within the same package as the `Entry` class. This class will represent the user's data inside the database (add getters and setters, because it was hidden for brevity). Notice that when importing all the referenced classes, your IDE will ask you to chose from what package to import `@Transient` annotation. Import `@Transient` annotation from `javax.persistence` package:

```
@Entity
public class ClientUser {
    @Id</span> @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String accessToken;
    private Calendar accessTokenValidity;
    private String refreshToken;
    @Transient
    private List<Entry> entries = new ArrayList<>();
    // getters and setters hidden for brevity
}
```

10. The `ClientUser` class previously declared, is an entity which can be managed by JPA. So, let's declare a repository using the facilities provided by Spring Data JPA. To do this, create the class `UserRepository` within the same package as `ClientUser` as follows:

```
public interface UserRepository extends CrudRepository<ClientUser, Lor
    Optional<ClientUser> findByUsername(String username);
}
```

11. Create the following class within the same package as the previous classes, because we will need it to map all the fields retrieved by the `/api/profile` endpoint from the OAuth Provider application (also, declare the getters and setters):

```
public class UserProfile {
    private String name;
    private String email;
    // getters and setters hidden for brevity
}
```

12. Before creating the controller class, let's first create some classes that are in charge of managing the user's details from a security perspective. Create the class `ClientUserDetails` which implements the`UserDetails` interface from Spring Security. This class should be created within the package `com.packt.example.clientauthorizationcode.security`:

```
public class ClientUserDetails implements UserDetails {
    private ClientUser clientUser;
    public ClientUserDetails(ClientUser user) { this.clientUser = user
    public ClientUser getClientUser() { return clientUser; }
    @Override public Collection<? extends GrantedAuthority> getAuthori
        return new HashSet<>();
    }
    @Override public String getPassword() { return clientUser.getPassw
    @Override public String getUsername() { return clientUser.getUserr
    @Override public boolean isAccountNonExpired() { return true; }
    @Override public boolean isAccountNonLocked() { return true; }
    @Override public boolean isCredentialsNonExpired() { return true;
    @Override public boolean isEnabled() { return true; }
}
```

13. Complementary to the `ClientUserDetails` class, create the class `ClientUserDetailsService` within the same package as `ClientUserDetails`. As you might notice, the following class implements the `UserDetailsService` interface, which allows Spring Security to load user related data when authenticating any user:

```
@Service
public class ClientUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepository users;
    @Override
    public UserDetails loadUserByUsername(String username) throws User
        Optional<ClientUser> optionalUser = users.findByUsername(userr
        if (!optionalUser.isPresent()) {
            throw new UsernameNotFoundException("invalid username or p
        }
```

```
            return new ClientUserDetails(optionalUser.get());
        }
    }
```

14. Now, to allow Spring Security to start using the `UserDetailsService` that we have implemented in the previous step, create the following class to configure security details within the same package as `UserDetailsService`.

```
    @Configuration @EnableWebSecurity
    public class SecurityConfiguration extends WebSecurityConfigurerAdapte
        @Autowired
        private UserDetailsService userDetailsService;
        @Override
        protected void configure(AuthenticationManagerBuilder auth) throws
            auth.userDetailsService(userDetailsService);
        }
        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests().antMatchers("/", "/index.html")
                .permitAll().anyRequest().authenticated().and()
                .formLogin().and()
                .logout().permitAll();
        }
    }
```

15. At this moment we are ready to create the controller which will manage all the requests to the respective client application endpoints. Create the class `UserDashboard` within the package `com.packt.example.clientauthorizationcode.user`:

```
    @Controller
    public class UserDashboard {
    }
```

16. The first path to provide will be the root path of the application which will be in charge of presenting the `index.html` page. Add the following method inside `UserDashboard`:

```
    @GetMapping("/")
    public String home() {
        return "index";
    }
```

17. Before adding the method responsible for rendering `dashboard.html`, add the following attribute, which is the `OAuth2RestTemplate` bean, into the class `UserDashboard`. The `OAuth2RestTemplate` is provided by Spring Security OAuth2 and will be configured further:

```
        @Autowired
        private OAuth2RestTemplate restTemplate;
```

18. Now add the following methods within `UserDashboard`. These methods will collaborate to retrieve the user's profile from the OAuth 2 Provider application that must be running on port `8080`:

```
    @GetMapping("/dashboard")
    public ModelAndView dashboard() {
        ClientUserDetails userDetails = (ClientUserDetails) SecurityContex
                .getContext().getAuthentication().getPrincipal();
        ClientUser clientUser = userDetails.getClientUser();
        clientUser.setEntries(Arrays.asList(new Entry("entry 1"),new Entry
        ModelAndView mv = new ModelAndView("dashboard");
        mv.addObject("user", clientUser);
        tryToGetUserProfile(mv);
        return mv;
    }

    private void tryToGetUserProfile(ModelAndView mv) {
        String endpoint = "http://localhost:8080/api/profile";
        try {
            UserProfile userProfile = restTemplate.getForObject(endpoint,
            mv.addObject("profile", userProfile);
        } catch (HttpClientErrorException e) {
            throw new RuntimeException("it was not possible to retrieve us
        }
    }
```

19. When using the Authorization Code grant type, if the current user does not have any access token associated with its account, the `OAuth2RestTemplate` instance will start redirecting the user to the OAuth Provider so she can be authenticated and granted the required permissions. After granting all the permissions, the user is redirected back to the redirect URI. To accept the request of the redirect URI, we need to declare the appropriate method as follows. Declare the following method callback within the `UserDashboard` class:

```
    @GetMapping("/callback")
    public ModelAndView callback() {
        return new ModelAndView("forward:/dashboard");
    }
```

20. Now we have all the required logic to run the client application, but to properly interact with the OAuth Provider we must do some configurations and have to declare how to save and retrieve any access token issued by the Authorization Server. Let's start by declaring the

`ClientTokenServices` class that will provide CRUD operations on token related data. This class should be created within the `com.packt.example.clientauthorizationcode.oauth` package:

```
@Service
public class OAuth2ClientTokenSevices implements ClientTokenServices {
```

21. Add the following attribute to inject `UserRepository` and add the following private method to retrieve token information related to the current logged user:

```
@Autowired private UserRepository users;

private ClientUser getClientUser(Authentication authentication) {
    ClientUserDetails loggedUser = (ClientUserDetails) authentication.
    Long userId = loggedUser.getClientUser().getId();
    return users.findOne(userId);
}
```

22. Now, add the following method to retrieve any previously saved access tokens:

```
@Override
public OAuth2AccessToken getAccessToken(OAuth2ProtectedResourceDetails
    Authentication authentication) {
    ClientUser clientUser = getClientUser(authentication);
    String accessToken = clientUser.getAccessToken();
    Calendar expirationDate = clientUser.getAccessTokenValidity();
    if (accessToken == null) return null;
    DefaultOAuth2AccessToken oAuth2AccessToken = new DefaultOAuth2Acce
    oAuth2AccessToken.setExpiration(expirationDate.getTime());
    return oAuth2AccessToken;
}
```

23. Then add the following method to save an issued access token:

```
@Override
public void saveAccessToken(OAuth2ProtectedResourceDetails resource,
    Authentication authentication, OAuth2AccessToken accessToken) {
    Calendar expirationDate = Calendar.getInstance();
    expirationDate.setTime(accessToken.getExpiration());
    ClientUser clientUser = getClientUser(authentication);
    clientUser.setAccessToken(accessToken.getValue());
    clientUser.setAccessTokenValidity(expirationDate);
    users.save(clientUser);
}
```

24. And finally, add the last method to allow the removing of an access token (this might be performed because of token expiration or because

of any other reason why an access token might become invalid):

```
@Override
public void removeAccessToken(OAuth2ProtectedResourceDetails resource,
    Authentication authentication) {
    ClientUser clientUser = getClientUser(authentication);
    clientUser.setAccessToken(null);
    clientUser.setRefreshToken(null);
    clientUser.setAccessTokenValidity(null);
    users.save(clientUser);
}
```

25. To finish creating all the OAuth 2.0 configurations to the client application, create the class `ClientConfiguration` with the following content within the same package as `OAuth2ClientTokenServices` that we previously created:

```
@Configuration @EnableOAuth2Client
public class ClientConfiguration {
    @Autowired
    private ClientTokenServices clientTokenServices;
    @Autowired
    private OAuth2ClientContext oauth2ClientContext;

}
```

26. Declare the following bean inside `ClientConfiguration`:

```
@Bean
public OAuth2ProtectedResourceDetails authorizationCode() {
    AuthorizationCodeResourceDetails resourceDetails = new Authorizati
    resourceDetails.setId("oauth2server");
    resourceDetails.setTokenName("oauth_token");
    resourceDetails.setClientId("clientapp");
    resourceDetails.setClientSecret("123456");
    resourceDetails.setAccessTokenUri("http://localhost:8080/oauth/tok
    resourceDetails.setUserAuthorizationUri("http://localhost:8080/oau
    resourceDetails.setScope(Arrays.asList("read_profile"));
    resourceDetails.setPreEstablishedRedirectUri(("http://localhost:90
    resourceDetails.setUseCurrentUri(false);
    resourceDetails.setClientAuthenticationScheme(AuthenticationScheme
    return resourceDetails;
}
```

27. And declare the `OAuth2RestTemplate` bean that will be responsible for managing all requests to OAuth 2.0 protected resources:

```
@Bean
public OAuth2RestTemplate oauth2RestTemplate() {
    OAuth2ProtectedResourceDetails resourceDetails = authorizationCode
    OAuth2RestTemplate template = new OAuth2RestTemplate(resourceDetai
    AccessTokenProviderChain provider = new AccessTokenProviderChain(
            Arrays.asList(new AuthorizationCodeAccessTokenProvider()))
    provider.setClientTokenServices(clientTokenServices);
    template.setAccessTokenProvider(provider);
    return template;
}
```

28. One important thing that must be done is to override the name of the `JSESSIONID` cookie that's created by default when implementing web applications. Add the following method inside the class `ClientAuthorizationCodeApplication` and make sure that this class implements `ServletContextInitializer` :

```
    @Override
    public void onStartup(ServletContext servletContext) throws ServletExc
        servletContext.getSessionCookieConfig().setName("client-session");
    }
```

How it works...

The most important classes created in this recipe were `ClientConfiguration` and `OAuth2ClientTokenServices`, which add all that is required for our client application to access OAuth's protected resources. The other classes are more focused on the functionality by itself, which is to retrieve some fictitious user's entries and her profile, that in turn is held by the OAuth profile application.

Notice that `ClientConfiguration` class, is annotated with `@Configuration` and `@EnableOAuth2Client` because we are both enabling our application as an OAuth 2.0 Client and we are declaring some other beans. `@EnableOAuth2Client` annotation imports another annotation called `OAuth2ClientConfiguration` which in turn configures the context needed for an access token request.

The `ClientConfiguration` class declares two other important elements for an OAuth 2.0 client which is implemented using Spring Security OAuth 2.0. They are `OAuth2ProtectedResourceDetails` and `OAuth2RestTemplate`. The former allows us to set up all the required data that is used for authorization and token request phases. As you might realize, we are using the `AuthorizationCodeResourceDetails` implementation of `OAuth2ProtectedResourceDetails`. Spring Security OAuth2 provides the resource details of concrete classes for each grant type described by the OAuth 2.0 protocol.

When setting up the client ID, client secret, the URI to obtain the user's authorization, and the URI to obtain access token, the application has conditions to use the other declared bean, which is `OAuth2RestTemplate`. That's one of the most important elements used in most of the recipes in this chapter, because it's through this `RestTemplate` implementation that the OAuth's dance starts executing, thus allowing for an access token request.

And what about `OAuth2ClientTokenServices`? This class is declared as a service by the use of the `@Service` annotation, which means that it can be injected whenever you want (since the place you are injecting is managed by Spring's context). The class `OAuth2ClientTokenServices` implements the interface `ClientTokenServices` .

By implementing `ClientTokenServices`, the client application is able to retrieve, save, or remove an access token from its database, allowing you to choose where to save this data. For example, you might use another kind of database.

One important thing that we have done is to define another name for `JSESSIONID` cookie. We did this because as the OAuth Provider and the client application are running at the same location (which is localhost), one application will override the cookie created by the other application, leading us to the impossibility of saving states between redirections performed by the OAuth 2.0 protocol.

Now, to see all of the code implemented here in practice, start both the OAuth Provider and the client application that we have created in this recipe. After starting both the client application and the server, go to `http://localhost:9000` and click on `Go to your dashboard` link that will be rendered when visiting the `index.html` page. Then you will be prompted to log in using the credentials defined for the client application, which was `aeloy` for username and `abc` for the password (these credentials were created through SQL commands in the recipe, but you may use anything that you want).

After logging in you will be redirected to the `auth-code-server` application where you will need to enter the credentials defined at the Authorization Server, which in my case was `adolfo` and `123` for the username and password respectively. Grant the required permissions and you will be redirected back to the URI defined by the `redirect_uri` parameter that will be handled by the `callback` method defined within the `UserDashboard` class. After being redirected back to the `client-authorization-code` application, you should see the dashboard page with the user's profile data.

Creating an OAuth 2.0 client using the Implicit grant type

This recipe will present you with how to implement a client application that uses the Implicit grant type to retrieve an access token and to retrieve a user's profile which is protected by the OAuth 2.0 protocol.

Getting ready

To run this recipe, make sure you have an OAuth 2.0 Provider running on your machine. I recommend you use the project `implicit-server` presented in Chapter 2, *Implementing Your Own OAuth 2.0 Provider*. This project is available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/implicit-server. In addition, this recipe requires the use of the database and table created for the first recipe, which was `clientdb` and `client_user` respectively. To have an OAuth 2.0 Provider running, you will need Java 8, Maven, jQuery, and your preferred IDE (remember that most examples are presented using Eclipse).

How to do it...

The next steps will present you with creating client applications that use the Implicit grant type to retrieve an access token and to interact with the user's profile endpoint:

1. Create the project using Spring Initializr. Go to https://start.spring.io/ and fill out the form using the following data:
    - Set up the Group as `com.packt.example`
    - Define the Artifact as `client-implicit`
    - Add `Web`, `Security`, `Thymeleaf`, `JPA`, and `MySQL` as dependencies for this project (you can choose each of these dependencies through the Spring Initializr's main page)
2. After creating the `client-implicit` project, import it to your IDE.
3. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. This application relies on the database to store the user's credentials. Make sure you have the database `clientdb` and the table `client_user` that was created in the first recipe. In addition, make sure you have created an entry for the `client_user` table so you will be able to log in to the `client-implicit` application (also clean the table fields `access_token`, `access_token_validity` and `refresh_token`).
5. Now open the `application.properties` file and add the same content that was defined in the first recipe to configure the data source and server port.

6. This Client application will provide a web interface to allow the user to see some fictitious data and her profile (which is stored within the OAuth Provider). There will be two simple pages with similar content used in the first recipe. There will be the files `index.html` and `dashboard.html` inside the `src/main/resources/templates` directory. Go back

to recipe one to see the content for these files, or look at the source code available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter03/client-implicit.

7. As the profile data will be retrieved on the client side, we will use Ajax and the content will be presented dynamically. To do so, change the table for the profile data of `dashboard.html` to look like follows:

```
<table>
  <tr>
      <td><b>name</b></td>
      <td><span class="name"></span></td>
  </tr>
  <tr>
      <td><b>email</b></td>
      <td><span class="email"></span></td>
  </tr>
</table>
```

8. In addition, insert the following script tags after the body tag within the `dashboard.html` file:

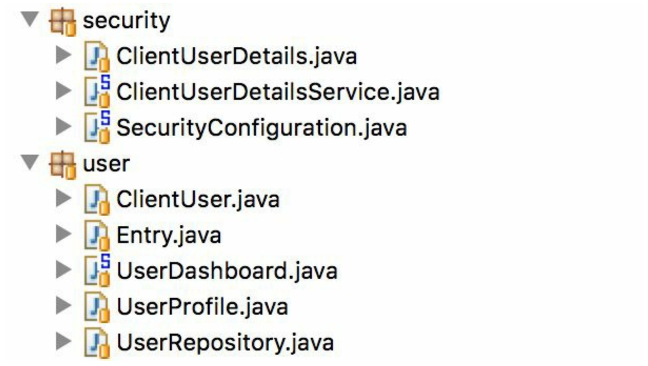
```
<script src="/jquery-2.2.0.min.js"></script>
<script src="/profile.js"></script>
```

9. Download the jQuery minified version from https://code.jquery.com/jquery-2.2.0.min.js, add it to the `src/main/resources/static` folder, and create the `profile.js` file within the same folder. Add the following JavaScript code within `profile.js`:

```
$(function() {
  var fragment = window.location.hash;
  var parameters = fragment.slice(1).split('&');
  var oauth2Token = {};
  $(parameters).each(function(idx, param) {
    var keyValue = param.split('=');
    oauth2Token[keyValue[0]] = keyValue[1];
  });
  $.ajax({
      url: 'http://localhost:8080/api/profile',
      beforeSend: function(xhr) {
          xhr.setRequestHeader("Authorization", "Bearer " + oauth2Toke
      },
      success: function(data){
         $('.name').text(data.name);
         $('.email').text(data.email);
         window.location.replace("#");
      },
      error: function(jqXHR, textStatus, errorThrown)   {
          console.log(textStatus);
```

```
            }
        });
    });
```

10. Before creating the OAuth 2.0 configuration classes, you must create all the classes that were also created for the first recipe. We need those classes because we want the same features implemented for this recipe (which is to retrieve the user's profile from the OAuth 2.0 Provider). So make sure you have declared all the classes presented in the following screenshot, within the `client-implicit` project, using the same source code presented in `client-authorization-code` project from first recipe (all the source code for these classes is also available on GitHub):



```
▼ security
    ▶ ClientUserDetails.java
    ▶ ClientUserDetailsService.java
    ▶ SecurityConfiguration.java
▼ user
    ▶ ClientUser.java
    ▶ Entry.java
    ▶ UserDashboard.java
    ▶ UserProfile.java
    ▶ UserRepository.java
```

11. Regarding the class `UserDashboard`, we need to make some changes because of the flow defined by the Implicit grant type. Make sure the `UserDashboard` class has the following content:

```
@Controller
public class UserDashboard {
    @Autowired
    private OAuth2RestTemplate restTemplate;

    @GetMapping("/")
    public String home() { return "index"; }

    @GetMapping("/callback")
    public ModelAndView callback() {
        ClientUser clientUser = getClientUserData();
        ModelAndView mv = new ModelAndView("dashboard");
        mv.addObject("user", clientUser);
        return mv;
    }

    @GetMapping("/dashboard")
    public ModelAndView dashboard() {
        ClientUser clientUser = getClientUserData();
        ModelAndView mv = new ModelAndView("dashboard");
```

```
            mv.addObject("user", clientUser);
            startOAuth2Dance();
            return mv;
        }

        private void startOAuth2Dance() { restTemplate.getAccessToken(); }

        private ClientUser getClientUserData() {
           ClientUserDetails userDetails = (ClientUserDetails) SecurityCor
                    .getContext().getAuthentication().getPrincipal();
            ClientUser clientUser = userDetails.getClientUser();
            clientUser.setEntries(Arrays.asList(new Entry("entry 1"), new
            return clientUser;
        }
    }
```

12. Now let's move on to OAuth configurations. The first thing to do is to create the `OAuth2ClientTokenServices` within the sub-package `oauth`. This class must implement `ClientTokenServices` and must be equal to the `OAuth2ClientTokenServices` that we created for `client-authorization-code` in the first recipe.
13. Then create the class `ClientConfiguration`, which will have some changes to suit to the Implicit grant type. This time, the implementation for `OAuth2ProtectedResourceDetails` to be created will be of the `ImplicitResourceDetails` type as follows:

```
    @Configuration @EnableOAuth2Client
    public class ClientConfiguration {
        @Autowired private ClientTokenServices clientTokenServices;
        @Autowired private OAuth2ClientContext oauth2ClientContext;

        @Bean
        public OAuth2ProtectedResourceDetails implicitResourceDetails() {
            ImplicitResourceDetails resourceDetails = new ImplicitResource
            resourceDetails.setId("oauth2server");
            resourceDetails.setTokenName("oauth_token");
            resourceDetails.setClientId("clientapp");
            resourceDetails.setUserAuthorizationUri("http://localhost:808
            resourceDetails.setScope(Arrays.asList("read_profile"));
            resourceDetails.setPreEstablishedRedirectUri("http://localhost
            resourceDetails.setUseCurrentUri(false);
            resourceDetails.setClientAuthenticationScheme(AuthenticationSc
            return resourceDetails;
        }
    }
```

14. Add the following method to declare the `OAuth2RestTemplate` bean inside `ClientConfiguration`:

```
    @Bean
```

```
public OAuth2RestTemplate oauth2RestTemplate() {
    OAuth2ProtectedResourceDetails resourceDetails = implicitResourceD
    OAuth2RestTemplate template = new OAuth2RestTemplate(resourceDetai
    AccessTokenProviderChain provider = new AccessTokenProviderChain(
            Arrays.asList(new CustomImplicitAccessTokenProvider()));
    provider.setClientTokenServices(clientTokenServices);
    template.setAccessTokenProvider(provider);
    return template;
}
```

15. Notice that we are supplying an instance of `CustomImplicitAccessTokenProvider` as a provider for the Implicit grant type. In order for this code to compile, create the following class within the same package of `ClientConfiguration`:

```
public class CustomImplicitAccessTokenProvider implements AccessTokenP
    @Override public OAuth2AccessToken obtainAccessToken(OAuth2Protect
        AccessTokenRequest request) throws RuntimeException {
        ImplicitResourceDetails resource = (ImplicitResourceDetails) d
        Map<String, String> requestParameters = getParametersForTokenR
        UserRedirectRequiredException redirectException = new UserRedi
                resource.getUserAuthorizationUri(), requestParameters)
        throw redirectException;
    }
    @Override public boolean supportsResource(OAuth2ProtectedResourceD
        return resource instanceof ImplicitResourceDetails
                && "implicit".equals(resource.getGrantType());
    }
    @Override public OAuth2AccessToken refreshAccessToken(
        OAuth2ProtectedResourceDetails resource, OAuth2RefreshToken re
        AccessTokenRequest request) throws UserRedirectRequiredExcepti
            return null;
   }
    @Override public boolean supportsRefresh(OAuth2ProtectedResourceDe
        return false;
    }
    private Map<String, String> getParametersForTokenRequest(
            ImplicitResourceDetails resource, AccessTokenRequest reque
        // need to create the parameters to request an access token
    }
}
```

16. The previous code is still incomplete because we did not create the parameters required to request a new access token through the authorization endpoint (which is done when using the Implicit grant type). Add the following snippet of code inside the private method `getParametersForTokenRequest` which belongs to the `CustomImplicitAccessTokenProvider` class:

```
private Map<String, String> getParametersForTokenRequest(
            ImplicitResourceDetails resource, AccessTokenRequest request)
        Map<String, String> queryString = new HashMap<String, String>();
        queryString.put("response_type", "token");
        queryString.put("client_id", resource.getClientId());
        if (resource.isScoped()) {
            queryString.put(
                "scope", resource.getScope().stream().reduce((a, b) -> a +
        }
        String redirectUri = resource.getRedirectUri(request);
        if (redirectUri == null) throw new IllegalStateException("No redir
        queryString.put("redirect_uri", redirectUri);
        return queryString;
    }
```

17. Now, open the class `ClientImplicitApplication` and make sure it implements the `ServletContextInitializer`, overriding the method `onStartup` as follows (remember that we are using this strategy to avoid session collision):

```
    @Override
    public void onStartup(ServletContext servletContext) throws ServletExc
        servletContext.getSessionCookieConfig().setName("client-session");
    }
```

*Before running the application, go to How it works... section because this client application needs that the Resource Server provides **Cross-origin Resource Sharing (CORS)** support.*

How it works...

This recipe has some subtleties regarding the `ClientConfiguration` class. The first thing to notice is that we are declaring an `OAuth2ProtectedResourceDetails` bean of type `ImplicitResourceDetails`. By using this concrete class to satisfy the Implicit grant type requirements, notice that we are not setting up the client secret and the token endpoint. This makes sense because the Implicit grant type uses the authorization endpoint to deliver the access token implicitly in such a way that the client does not need to make a request for the `/oauth/token` endpoint.

The other interesting point to note is that we are using an instance of `CustomImplicitAccessTokenProvider` to create the `OAuth2RestTemplate` bean. When `OAuth2RestTemplate` tries to request an access token or tries to access some of OAuth's protected resources, the `obtainAccessToken` method from `CustomImplicitAccessTokenProvider` triggers the user's redirection to the authorization endpoint. The token extraction has to be done on the client side (for example, using JavaScript).

Notice that for this client application to run without any problems, the Resource Server must support **Cross-origin Resource Sharing** (**CORS**). So, make sure the source code for the Resource Server configuration looks as follows (you have to edit the class `OAuth2ResourceServer.java` from the `implicit-server` project recommended for this recipe):

```
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
  @Override public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
         .anyRequest().authenticated().and()
         .requestMatchers().antMatchers("/api/**").and()
         .cors();
   }
}
```

In addition, you also need to add the following annotation above the method declaration for the profile endpoint of the implicit-server application. The following snippet of code shows the usage of the `@CrossOrigin` annotation:

```
@Controller
public class UserController {
    @CrossOrigin
    @RequestMapping("/api/profile")
    public ResponseEntity<UserProfile> profile() {
        // body of this method hidden for brevity
    }
}
```

Start the applications `implicit-server` and `client-implicit` and go to `http://localhost:9000`. After logging in with the same credentials used for the first recipe and after granting permissions, you will be redirected back to `client-implicit` with the access token presented through a URI fragment (which is extracted by the `profile.js` JavaScript). Notice that the fragment is cleaned so as to prevent the user from looking up the access token.

There's more...

Notice that instead of using the `ImplicitAccessTokenProvider` from Spring Security OAuth2, we have created the class `CustomImplicitAccessTokenProvider`. We did this because the implementation provided by the framework is using a different approach to retrieve access tokens through the Implicit grant type. Instead of only using the authorization endpoint, the implementation from Spring Security OAuth2 is using the token endpoint which does not adhere to OAuth 2.0's specifications. There might be good reasons to do so, but as this book follows what is defined by OAuth 2.0's specifications, this recipe presented a customized version for the `ImplicitAccessTokenProvider`.

See also

- Creating an OAuth 2.0 client using the Authorization Code grant type

Creating an OAuth 2.0 client using the Resource Owner Password Credentials grant type

This recipe shows you how to create an OAuth 2.0 client application using the Resource Owner Password Credentials grant type.

Getting ready

To run this recipe, make sure you have an OAuth 2.0 Provider running on your machine. I recommend you use the project `password-server` presented in Chapter 2, *Implementing Your Own OAuth 2.0 Provider*. This project is available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/password-server. In addition, this recipe requires the use of the databases and tables created for the first recipe which were `clientdb` and `client_user` respectively. Besides having an OAuth 2.0 Provider running, you will need Java 8, Maven, and your preferred IDE (remember that most examples are presented using Eclipse).

How to do it...

The next steps will help you create a client application that uses the Resource Owner Password Credentials grant type to retrieve an access token and to interact with the user's profile endpoint:

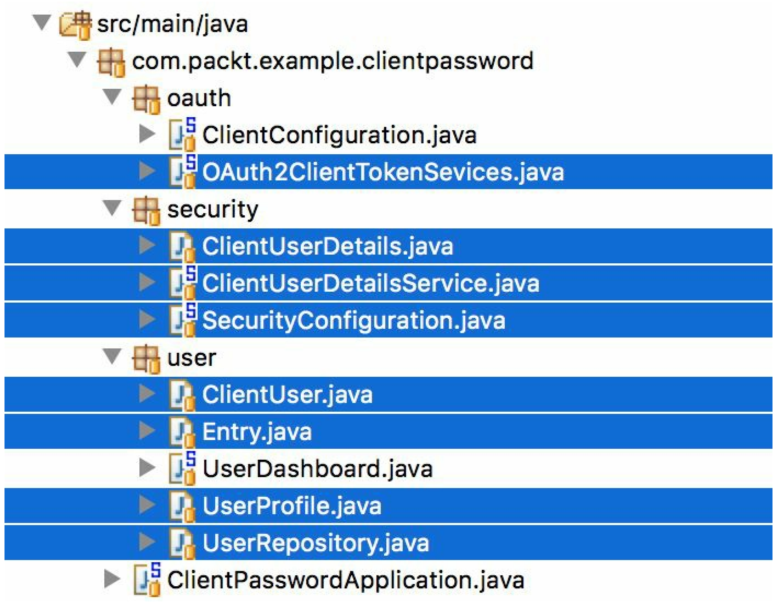
1. Create the project using Spring Initializr. Go to https://start.spring.io/ and fill out the form using the following data:
    - Set up the Group as `com.packt.example`
    - Define the Artifact as `client-password`
    - Add `Web`, `Security`, `Thymeleaf`, `JPA` and `MySQL` as dependencies for this project (you can choose each of these dependencies through the Spring Initializr's main page)
2. After creating the `client-implicit` project, import it to your IDE.
3. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. This application relies on the database to store user's credentials. Make sure you have the database `clientdb` and the table `client_user` that were created in the *Creating an OAuth 2.0 Client using the Authorization Code grant type* recipe. In addition, make sure you have created an entry for the `client_user` table so you are able to log in to the `client-password` application.
5. Now open the `application.properties` file and add the same content that is defined in the *Creating an OAuth 2.0 client using the Authorization Code grant type* recipe to configure the datasource and server port.
6. This client application will provide the same web interfaces that we provided for the first recipe where you were presented with how to create a Client using the Authorization Code grant type. So copy `index.html` and `dashboard.html` from the `client-authorization-code` project inside the folder `src/main/resources/templates`, from the `client-password`

project.

7. Make sure you have created all classes highlighted in the following screenshot. These classes should have the same content present as in the `client-authorization-code` project. Just copy those classes to their respective packages as presented below (do not copy the classes that are not highlighted because this recipe will present you with how you can create them):



8. Now create the class `UserDashboard` within the package presented in the previous screenshot and add the following content:

```
@Controller
public class UserDashboard {
    @Autowired
    private OAuth2RestTemplate restTemplate;
    @Autowired
    private AccessTokenRequest accessTokenRequest;
    @GetMapping("/")
    public String home() { return "index"; }
    @GetMapping("/callback")
    public ModelAndView callback() {
        return new ModelAndView("forward:/dashboard");
    }
    @GetMapping("/dashboard")
    public ModelAndView dashboard() {
        ClientUserDetails userDetails = (ClientUserDetails) SecurityCc
                .getContext().getAuthentication().getPrincipal();
        ClientUser clientUser = userDetails.getClientUser();
        clientUser.setEntries(Arrays.asList(new Entry("entry 1"), new
        ModelAndView mv = new ModelAndView("dashboard");
        mv.addObject("user", clientUser);
        tryToGetUserProfile(mv);
        return mv;
    }
    private void tryToGetUserProfile(ModelAndView mv) {
        accessTokenRequest.add("username", "adolfo");
        accessTokenRequest.add("password", "123");
        String endpoint = "http://localhost:8080/api/profile";
        try {
            UserProfile userProfile = restTemplate.getForObject(endpoi
            mv.addObject("profile", userProfile);
        } catch (HttpClientErrorException e) {
            throw new RuntimeException("it was not possible to retriev
        }
    }
}
```

9. Then create the class `ClientConfiguration` as follows:

```
    @Configuration @EnableOAuth2Client
    public class ClientConfiguration {
        @Autowired
        private ClientTokenServices clientTokenServices;
        @Autowired
        private OAuth2ClientContext oauth2ClientContext;
        @Bean
        public OAuth2ProtectedResourceDetails passwordResourceDetails() {
            ResourceOwnerPasswordResourceDetails resourceDetails = new Res
            resourceDetails.setId("oauth2server");
            resourceDetails.setTokenName("oauth_token");
            resourceDetails.setClientId("clientapp");
            resourceDetails.setClientSecret("123456");
            resourceDetails.setAccessTokenUri("http://localhost:8080/oauth
            resourceDetails.setScope(Arrays.asList("read_profile"));
            resourceDetails.setClientAuthenticationScheme(AuthenticationSc
            return resourceDetails;
        }
        @Bean
        public OAuth2RestTemplate oauth2RestTemplate() {
            OAuth2ProtectedResourceDetails resourceDetails = passwordResou
            OAuth2RestTemplate template = new OAuth2RestTemplate(resourceD
                    oauth2ClientContext);
            AccessTokenProviderChain provider = new AccessTokenProviderCha
                    Arrays.asList(new ResourceOwnerPasswordAccessTokenProv
            provider.setClientTokenServices(clientTokenServices);
            template.setAccessTokenProvider(provider);
            return template;
        }
    }
```

How it works...

Besides declaring the appropriate bean to configure resource details using the class `ResourceOwnerPasswordResourceDetails`, we are also declaring one bean of type `OAuth2RestTemplate` which uses the respective provider for the current grant type being addressed. That is, we are setting up an instance of `ResourceOwnerPasswordAccessTokenProvider` for the `OAuth2RestTemplate`. These kinds of configurations are not so special now that you have learned how to configure the Authorization Code and Implicit grant types. But there is something interesting happening inside the `UserDashboard` class, within the method `tryToGetUserProfile`. Let's take a look at the following snippet of code:

```
private void tryToGetUserProfile(ModelAndView mv) {
    accessTokenRequest.add("username", "adolfo");
    accessTokenRequest.add("password", "123");
    // code hidden for brevity
}
```

We are setting up the user's credentials because we are using the Resource Owner Password Credentials grant type. By doing so, we can define the user's credentials through an instance of `AccessTokenRequest` which is created by Spring Security OAuth2 with a request scope that is preserved by the `DefaultOAuth2ClientContext` instance which is session scoped.

Start both client and server applications, go to `http://localhost:9000`, and start the same navigation process that you have done for the previous recipes (make sure to clean the token related fields on `client_user` table from `clientdb` database and make sure to clean your project to prevent against cache issues).

There's more...

You might have realized that when setting up the user's credentials before trying to access OAuth's protected resources at the `UserDashboard` class, we were using hard coded credentials. A better approach here would be to ask for the user's credentials (these must be the credentials the user has at the Authorization Server instead of the credentials that she has in the client application) and start setting up the username and password dynamically. Try to do it as an exercise and do not forget to throw the credentials away after receiving the access token issued by the Authorization Server. It's really important to not persist the user's sent credentials.

See also

- Creating an OAuth 2.0 client using the Authorization Code grant type

Creating an OAuth 2.0 client using the Client Credentials grant type

This recipe will help you create a client application using the Client Credentials grant type. The project shown in this recipe retrieves the user's profile from the running OAuth provider available. Unlike the other grant types, when using Client Credentials the application will access OAuth's protected resources on its own behalf, so the client application doesn't need any user's approval.

Getting ready

To run this recipe, make sure you have an OAuth 2.0 Provider running on your machine. I recommend you to use the project `client-credentials-server` presented in Chapter 2, *Implementing Your Own OAuth 2.0 Provider*. If you have any doubt about the source code presented in this recipe, do not hesitate to download the source code available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter02/client-credentials-server. Besides having an OAuth 2.0 Provider running, you will need Java 8, Maven, and your preferred IDE (remember that most examples are presented using Eclipse).

How to do it...

The next steps will present you with how you can create a client application that uses the Client Credentials grant type:

1. Create the project using **Spring Initializr**. Go to https://start.spring.io/ and fill out the form using the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `client-client-credentials`
   - Add `Web`, `Security`, and `Thymeleaf` as dependencies for this project
2. After creating the `client-client-credentials` project, import it to your IDE.
3. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

4. Add the following content to the `application.properties` file:

```
server.port=9000
security.user.name=admin
security.user.password=123
spring.http.converters.preferred-json-mapper=jackson
```

5. This Client application will provide the same web interfaces that we provided for *Creating an OAuth 2.0 client using the Authorization Code grant type* recipe. So, copy `index.html` and `dashboard.html` from the `client-authorization-code` project into the folder `src/main/resources/templates` of the `client-client-credentials` project.
6. Change the `HTML` body content of `dashboard.html` file to what follows:

```
<table>
  <tr>
      <td><b>name</b></td>
      <td><b>email</b></td>
  </tr>
  <tr th:each="user : ${users}">
      <td th:text="${user.name}">name</td>
```

```
        <td th:text="${user.email}">email</td>
    </tr>
</table>
```

7. For this recipe, instead of persisting access tokens related to data inside any database, let's keep this information available through the user's session. To do this, create the following class within the `oauth` sub-package:

```
@Repository
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class SettingsRepository {
    private String accessToken;
    private Calendar expiresIn;
    public String getAccessToken() { return accessToken; }
    public void setAccessToken(String accessToken) { this.accessToken
    public Calendar getExpiresIn() { return expiresIn; }
    public void setExpiresIn(Calendar expiresIn) { this.expiresIn = ex
}
```

8. Create the class `OAuth2ClientTokenServices` with the following content using `SettingsRepository` instead of `UsersRepository`:

```
@Service
public class OAuth2ClientTokenSevices implements ClientTokenServices {
    @Autowired private SettingsRepository settings;
    public OAuth2AccessToken getAccessToken(
        OAuth2ProtectedResourceDetails resource, Authentication authen
        String accessToken = settings.getAccessToken();
        Calendar expirationDate = settings.getExpiresIn();
        if (accessToken == null) return null;
        DefaultOAuth2AccessToken oAuth2AccessToken = new DefaultOAuth2
        oAuth2AccessToken.setExpiration(expirationDate.getTime());
        return oAuth2AccessToken;
    }
    public void saveAccessToken(OAuth2ProtectedResourceDetails resourc
            Authentication authentication, OAuth2AccessToken accessTok
        Calendar expirationDate = Calendar.getInstance();
        expirationDate.setTime(accessToken.getExpiration());
        settings.setAccessToken(accessToken.getValue());
        settings.setExpiresIn(expirationDate);
    }
    public void removeAccessToken(OAuth2ProtectedResourceDetails resou
            Authentication authentication) {
        settings.setAccessToken(null);
        settings.setExpiresIn(null);
    }
}
```

9. Create the class `UserProfile` in the same way we did for the previous recipes so the results from `/api/users` endpoint declared at `client-`

credentials-server project, can be properly unmarshalled to a value object (which is UserProfile). This class might be declared within the oauth sub-package as with the other OAuth related classes.

10. Finally, the last OAuth related class to create is ClientConfiguration which will be the configuration class to declare two important beans. As you might expect, we have to declare OAuth2ProtectedResourceDetails and OAuth2RestTemplate as we did for the other recipes. Declare the class ClientConfiguration using the same annotations and inject the same dependencies that we added for the ClientConfiguration of the client-authorization-code project.

11. Then, add the OAuth2ProtectedResourceDetails bean's declaration inside ClientConfiguration:

```
@Bean
public OAuth2ProtectedResourceDetails passwordResourceDetails() {
    ClientCredentialsResourceDetails details = new ClientCredentialsRe
    details.setId("oauth2server");
    details.setTokenName("oauth_token");
    details.setClientId("clientadmin");
    details.setClientSecret("123");
    details.setAccessTokenUri("http://localhost:8080/oauth/token");
    details.setScope(Arrays.asList("admin"));
    details.setClientAuthenticationScheme(AuthenticationScheme.header)
    return details;
}
```

12. And add the OAuth2RestTemplate bean declaration as follows:

```
@Bean
public OAuth2RestTemplate oauth2RestTemplate() {
    OAuth2ProtectedResourceDetails resourceDetails = passwordResourceD
    OAuth2RestTemplate template = new OAuth2RestTemplate(resourceDetai
    AccessTokenProviderChain provider = new AccessTokenProviderChain(
            Arrays.asList(new ClientCredentialsAccessTokenProvider()))
    provider.setClientTokenServices(clientTokenServices);
    template.setAccessTokenProvider(provider);
    return template;
}
```

13. After creating the security and OAuth configurations, create the class AdminDashboard as follows:

```
@Controller
public class AdminDashboard {
    @Autowired
    private OAuth2RestTemplate restTemplate;

    @GetMapping("/")
    public String home() { return "index"; }

    @GetMapping("/callback")
    public ModelAndView callback() { return new ModelAndView("forward:

    @GetMapping("/dashboard")
    public ModelAndView dashboard() {
        ModelAndView mv = new ModelAndView("dashboard");
        String endpoint = "http://localhost:8080/api/users";
        try {
            UserProfile[] users = restTemplate.getForObject(endpoint,
            mv.addObject("users", users);
        } catch (HttpClientErrorException e) {
            throw new RuntimeException("it was not possible to retriev
        }
        return mv;
    }
}
```

How it works...

The main difference between this recipe and the others is that we have just declared an admin user which does not have any entry at the OAuth Provider application. This means that this application is not accessing resources on the user's behalf but on its own behalf. If you look at the resource details configuration, we are creating an instance of `ClientCredentialsResourceDetails` and setting up just the `/oauth/token` endpoint and client credentials, because now the client application doesn't need to redirect the user to another application (there is no OAuth 2.0 dance).

The creation of the `OAuth2RestTemplate` is also different because we are using another provider implementation which is `ClientCredentialsAccessTokenProvider`.

Start both `client-credentials-server` and `client-client-credentials` and go to `http://localhost:9000` to start interacting with the client application created in this recipe. Notice that the currently logged user (which should be the admin) does not share any credentials and isn't redirected to any other application.

See also

- Creating an OAuth 2.0 client using the Authorization Code grant type

Managing refresh tokens on the client side

This recipe will show you how to add support for refresh tokens. As you will realize, by using Spring Security OAuth2 it becomes simple to start persisting refresh tokens, and all the logic to recognize when it's time to use the refresh token to ask for a new access token will be done by Spring Security OAuth2.

Getting ready

To run this recipe, make sure you have an OAuth 2.0 Provider running on your machine. I recommend you use the project `auth-code-server` presented in Chapter 2, *Implementing Your Own OAuth 2.0 Provider*. If you have any doubts about the source code presented in this recipe, do not hesitate to download the source code available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter03/client-refresh-token. This recipe also relies on a database that was created in the *Creating an OAuth 2.0 client using the Authorization Code grant type* recipe for the `client-authorization-code` project.

The database creation commands declare the column to persist the refresh token.

How to do it...

The next steps will present you with how you can add support for refresh tokens to Client applications that are able to use refresh tokens.

Remember that refresh tokens should be supported when using Authorization Code and Resource Owner Password Credentials grant types. When using the Implicit grant type there is no need for refresh tokens because it's expected that the user is present when the access token expires and the Client application needs to ask for a new access token. And when using the Client Credentials grant type, the Client application can ask for a new access token by not relying on the user's approval.

1. For this recipe, instead of creating a new project you must use the application `client-authorization-code` project shown in the first recipe of this chapter.
2. Import the project `client-authorization-code` and open the class `OAuth2ClientTokenServices`.
3. Change the method `getAccessToken` to look as follows:

```
@Override
public OAuth2AccessToken getAccessToken(OAuth2ProtectedResourceDetails
    ClientUser clientUser = getClientUser(authentication);
    String accessToken = clientUser.getAccessToken();
    Calendar expirationDate = clientUser.getAccessTokenValidity();
    if (accessToken == null) return null;
    DefaultOAuth2AccessToken oAuth2AccessToken = new DefaultOAuth2Acce
    oAuth2AccessToken.setExpiration(expirationDate.getTime());
    oAuth2AccessToken.setRefreshToken(new
        DefaultOAuth2RefreshToken(clientUser.getRefreshToken()));
    return oAuth2AccessToken;
}
```

4. Then replace the method `saveAccessToken` from `OAuth2ClientTokenServices` with the snippet of code presented next:

```
@Override
public void saveAccessToken(OAuth2ProtectedResourceDetails resource,
        Authentication authentication, OAuth2AccessToken accessToken)
    Calendar expirationDate = Calendar.getInstance();
```

```
            expirationDate.setTime(accessToken.getExpiration());
            ClientUser clientUser = getClientUser(authentication);
            clientUser.setAccessToken(accessToken.getValue());
            clientUser.setAccessTokenValidity(expirationDate);
            clientUser.setRefreshToken(accessToken.getRefreshToken().getValue(
            users.save(clientUser);
        }
```

5. And finally, replace the `removeAccessToken` method with the following one:

```
        public void removeAccessToken(OAuth2ProtectedResourceDetails resource,
                Authentication authentication) {
            ClientUser clientUser = getClientUser(authentication);
            clientUser.setAccessToken(null);
            clientUser.setRefreshToken(null);
            clientUser.setAccessTokenValidity(null);
            users.save(clientUser);
        }
```

6. For this new client to work, the application `auth-code-server` from Chapter 2, *Implementing Your Own OAuth 2.0 Provider* needs to add support for refresh tokens. Make sure that the `configure` method from the Authorization Server configuration looks like the code presented below (notice the use of the `refresh_token` grant type and the use of the `accessTokenValiditySeconds` method):

```
        @Override
        public void configure(ClientDetailsServiceConfigurer clients)
         throws Exception {
         clients.inMemory().withClient("clientapp").secret("123456")
         .redirectUris("http://localhost:9000/callback")
         .authorizedGrantTypes("authorization_code", "refresh_token")
         .accessTokenValiditySeconds(120)
         .scopes("read_profile", "read_contacts");
        }
```

How it works...

This recipe showed how easy it can be when using some Spring Security OAuth2 configuration classes. Just by persisting the refresh tokens and setting their value to an instance of `OAuth2AccessToken` , all the logic required to send a request to the OAuth Provider when an access token expires is done by the framework behind the scenes. When `OAuth2RestTemplate` starts trying to access OAuth's protected resources, it will invoke the `obtainAccessToken` method from `AccessTokenProviderChain` which implements the logic to decide when to use the refresh token to retrieve a new access token.

See also

- Creating an OAuth 2.0 client using the Authorization Code grant type

Accessing an OAuth 2.0 protected API with RestTemplate

This recipe is useful when you don't have Spring Security OAuth2 in your project but are still using the Spring Framework. Instead of using `OAuth2RestTemplate`, this recipe shows you how to use raw `RestTemplate` implementation to interact with any OAuth 2.0 Provider.

Getting ready

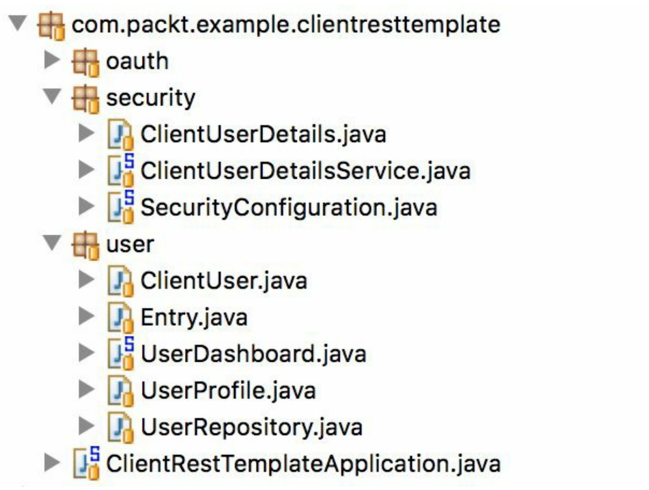
To run this recipe, make sure you have an OAuth 2.0 Provider running on your machine. I recommend you use the project `auth-code-server` presented in Chapter 2, *Implementing Your Own OAuth 2.0 Provider*. If you have any doubts about the source code presented in this recipe, do not hesitate to download the source code available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter03/client-rest-template. Besides having an OAuth 2.0 Provider running, you will need Java 8, Maven, and your preferred IDE (remember that most examples are presented using Eclipse).

How to do it...

The next steps show you how to create a client application that uses the Authorization Code grant type relying on `RestTemplate` to interact with the OAuth 2.0 Provider:

1. Create the project using **Spring Initializr**. Go to https://start.spring.io/ and fill out the form using the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `client-rest-template`
   - Add `Web`, `Security`, `Thymeleaf`, `JPA` and `Mysql` as dependencies for this project

2. After creating the `client-rest-template` project, import it to your IDE.
3. Now instead of using the facilities from Spring Security OAuth2, which allow us to just use configuration classes, we will interact directly with the provider's `/oauth` endpoints to request authorization and access tokens to be able to access the user's resources.
4. Copy all the classes that we have created to interact with the user's profile endpoint and all the classes that we have created in the `security` package from `client-authorization-code` project that was built for the first recipe. (After copying the classes, the project `client-rest-template` should have the following package structure):



```
▼ com.packt.example.clientresttemplate
  ▶ oauth
  ▼ security
    ▶ ClientUserDetails.java
    ▶ ClientUserDetailsService.java
    ▶ SecurityConfiguration.java
  ▼ user
    ▶ ClientUser.java
    ▶ Entry.java
    ▶ UserDashboard.java
    ▶ UserProfile.java
    ▶ UserRepository.java
  ▶ ClientRestTemplateApplication.java
```

5. Also, copy all the template files (`index.html` and `dashboard.html`) from `client-authorization-code` project that was built for the first recipe, to the `src/main/resources/templates` directory of `client-rest-template` project.
6. Copy the content from `application.properties` file from `client-authorization-code` project, to the `application.properties` file of `client-rest-template` project.
7. The main difference starts at this point by creating the class `OAuth2Token` as follows, which should be within the `oauth` sub-package (also create the getters and setters methods for each declared attribute):

```
public class OAuth2Token {
    @JsonProperty("access_token")
    private String accessToken;

    @JsonProperty("token_type")
    private String tokenType;

    @JsonProperty("expires_in")
    private String expiresIn;

    @JsonProperty("refresh_token")
    private String refreshToken;
    // getters and setters hidden
}
```

8. Then create the class `AuthorizationCodeConfiguration` with the following content (import `MediaType` class from `org.springframework.http` package). The `AuthorizationCodeConfiguration` class should be located in the same package as `OAuth2Token`:

```
@Component
public class AuthorizationCodeConfiguration {
    public String encodeCredentials(String username, String password)
        String credentials = username + ":" + password;
        String encoded = new String(Base64.getEncoder().encode(
                credentials.getBytes()));
        return encoded;
    }
    public MultiValueMap<String, String> getBody(String authorizationC
        MultiValueMap<String, String> formData = new LinkedMultiValueM
        formData.add("grant_type", "authorization_code");
        formData.add("scope", "read_profile");
        formData.add("code", authorizationCode);
        formData.add("redirect_uri", "http://localhost:9000/callback")
        return formData;
    }
    public HttpHeaders getHeader(String clientAuthentication) {
        HttpHeaders httpHeaders = new HttpHeaders();
        httpHeaders.setContentType(MediaType.APPLICATION_FORM_URLENCOI
        httpHeaders.setAccept(Arrays.asList(MediaType.APPLICATION_JSON
        httpHeaders.add("Authorization", "Basic " + clientAuthenticati
        return httpHeaders;
    }
}
```

9. Then create the class `AuthorizationCodeService` with the following content:

```
    @Service
    public class AuthorizationCodeTokenService {
        @Autowired
        private AuthorizationCodeConfiguration configuration;
        public String getAuthorizationEndpoint() {
            String endpoint = "http://localhost:8080/oauth/authorize";
            Map<String, String> authParameters = new HashMap<>();
            authParameters.put("client_id", "clientapp");
            authParameters.put("response_type", "code");
            authParameters.put("redirect_uri",
                    getEncodedUrl("http://localhost:9000/callback"));
            authParameters.put("scope", getEncodedUrl("read_profile"));
            return buildUrl(endpoint, authParameters);
        }
        private String buildUrl(String endpoint, Map<String, String> param
            List<String> paramList = new ArrayList<>(parameters.size());
            parameters.forEach((name, value) -> {
                paramList.add(name + "=" + value);
            });
            return endpoint + "?" + paramList.stream()
                  .reduce((a, b) -> a + "&" + b).get();
        }
        private String getEncodedUrl(String url) {
            try {
                return URLEncoder.encode(url, "UTF-8");
            } catch (UnsupportedEncodingException e) {
                throw new RuntimeException(e);
            }
        }
        public OAuth2Token getToken(String authorizationCode) {
            RestTemplate rest = new RestTemplate();
            String authBase64 = configuration.encodeCredentials(
                "clientapp", "123456");
            RequestEntity<MultiValueMap<String, String>> requestEntity = r
                configuration.getBody(authorizationCode),
                configuration.getHeader(authBase64), HttpMethod.POST,
                URI.create("http://localhost:8080/oauth/token"));
            ResponseEntity<OAuth2Token> responseEntity = rest.exchange(
                    requestEntity, OAuth2Token.class);
            if (responseEntity.getStatusCode().is2xxSuccessful())
                return responseEntity.getBody();
            throw new RuntimeException("error trying to retrieve access to
        }
    }
```

10. Given that we already have all the logic needed to retrieve an access token, let's create the code that uses the `AuthorizationCodeTokenService`

class to retrieve such access tokens to request the user's profile from `auth-code-server` application. Inject an instance of `AuthorizationCodeTokenService` and an instance of `UserRepository` inside the `UserDashboard` class as presented in the following code:

```
@Autowired
private AuthorizationCodeTokenService tokenService;
@Autowired
private UserRepository users;
```

11. Then, replace the method `callback` from the `UserDashboard`, which now should look as follows (notice the use of the `tokenService` reference):

```
@GetMapping("/callback")
public ModelAndView callback(String code, String state) {
    ClientUserDetails userDetails = (ClientUserDetails) SecurityContex
            .getContext().getAuthentication().getPrincipal();
    ClientUser clientUser = userDetails.getClientUser();
    OAuth2Token token = tokenService.getToken(code);
    clientUser.setAccessToken(token.getAccessToken());
    Calendar tokenValidity = Calendar.getInstance();
    tokenValidity.setTime(new Date(Long.parseLong(token.getExpiresIn()
    clientUser.setAccessTokenValidity(tokenValidity);
    users.save(clientUser);
    return new ModelAndView("forward:/dashboard");
}
```

12. The `dashboard` method from `UserDashboard` must be like the source code shown in the following code. Notice that the logic which decides when to redirect the user to the authorization endpoint is done here when `clientUser.getAccessTokens() == null` returns `true`:

```
@GetMapping("/dashboard")
public ModelAndView dashboard() {
    ClientUserDetails userDetails = (ClientUserDetails) SecurityContex
            .getContext().getAuthentication().getPrincipal();
    ClientUser clientUser = userDetails.getClientUser();
    if (clientUser.getAccessToken() == null) {
        String authEndpoint = tokenService.getAuthorizationEndpoint();
        return new ModelAndView("redirect:" + authEndpoint);
    }
    clientUser.setEntries(Arrays.asList(new Entry("entry 1"), new Entr
    ModelAndView mv = new ModelAndView("dashboard");
    mv.addObject("user", clientUser);
    tryToGetUserProfile(mv, clientUser.getAccessToken());
    return mv;
}
```

13. Now, to retrieve the user's profile, you can use `RestTemplate` instead of

`OAuth2RestTemplate`. Notice that it demands much more code be written than what had to done when using `OAuth2RestTemplate`:

```
private void tryToGetUserProfile(ModelAndView mv, String token) {
    RestTemplate restTemplate = new RestTemplate();
    MultiValueMap<String, String> headers = new LinkedMultiValueMap<>(
    headers.add("Authorization", "Bearer " + token);
    String endpoint = "http://localhost:8080/api/profile";
    try {
        RequestEntity<Object> request = new RequestEntity<>(
            headers, HttpMethod.GET, URI.create(endpoint));
        ResponseEntity<UserProfile> userProfile = restTemplate.exchang
            request, UserProfile.class);
        if (userProfile.getStatusCode().is2xxSuccessful()) {
            mv.addObject("profile", userProfile.getBody());
        } else {
            throw new RuntimeException("it was not possible to retriev
        }
    } catch (HttpClientErrorException e) {
        throw new RuntimeException("it was not possible to retrieve us
    }
}
```

14. Now, open the class `ClientRestTemplateApplication` and make sure it implements the`ServletContextInitializer`, overriding the method`onStartup` as follows:

```
public void onStartup(ServletContext servletContext) throws ServletExc
    servletContext.getSessionCookieConfig().setName("client-session");
}
```

How it works...

The implementation of the `UserDashboard` controller has the same methods implemented for the `client-authorization-code`, which are: `home`, `callback`, `dashboard`, and `tryToGetUserProfile`. Because `UserDashboard` is handling the user's redirection to the authorization endpoint and is also requesting new access tokens, we had to change the methods callback and dashboard.

When a request is handled by the `dashboard` method, it tries to retrieve an access token from the currently logged user and if it is `null` the redirection is fired as follows:

```
if (clientUser.getAccessToken() == null) {
    String authEndpoint = tokenService.getAuthorizationEndpoint();
    return new ModelAndView("redirect:" + authEndpoint);
}
```

Then after the user is redirected back to the redirect URI (which is handled by the `callback` method), the Authorization Code received as the method's parameters is used to obtain an access token through the use of the `tokenService` reference as follows:

```
OAuth2Token token = tokenService.getToken(code);
```

All of this implemented logic will do the same as what we achieve when using `OAuth2RestTemplate`, but now we have to write too much code and all the logic is not well encapsulated. So try to use `OAuth2RestTemplate` when it's possible.

See also

- Creating an OAuth 2.0 client using the Authorization Code grant type

OAuth 2.0 Profiles

In this chapter, we will cover the following recipes:

- Revoking issued tokens
- Remote validation using token introspection
- Improving performance using cache for remote validation
- Using Gatling to load test remote token validation
- Dynamic client registration

Introduction

By reading this chapter, you will learn how to use some distinct OAuth 2.0 profiles which enable the protocol to address custom scenarios that aren't specified by OAuth 2.0 specification (RFC 6749). These profiles are based on some extension points of OAuth 2.0, which are grant types and token types. It's essential to understand some OAuth 2.0 profiles so you can deliver real-world applications when faced with some specific scenarios.

Revoking issued tokens

This recipe helps you to revoke client access tokens, which is defined by the Token Revocation specification defined by RFC 7009 at https://tools.ietf.org/html/rfc7009. As per the specification, this OAuth 2.0 profile was created to allow clients to notify the Authorization Server that an access token or refresh token is no longer needed. This provides the Authorization Server with the ability to clear unused tokens in the database, thus avoiding useless data. Besides the revocation support, this recipe will help you to create an OAuth 2.0 Provider which also serves a user profile API, as we did for previous recipes in this book.

Getting ready

To run this recipe, you will need Java 8, Maven, MySQL, and your preferred IDE. As we will use Spring Boot, it's also recommended that you create an initial project using **Spring Initializr**.

How to do it...

The next steps will show how you can create an Authorization Server that implements a token revocation profile:

1. Create the project using Spring Initializr. Go to https://start.spring.io/ and fill out the form using the following data:
    - Set up the Group as `com.packt.example`
    - Define the Artifact as `revoke-server` and add `Web`, `Security`, `JPA`, and `MySQL` as dependencies for this project
2. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

3. Add the following content to the `application.properties` file:

```
security.user.name=adolfo
security.user.password=123

spring.datasource.url=jdbc:mysql://localhost/oauth2provider
spring.datasource.username=oauth2provider
spring.datasource.password=123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Di
spring.jpa.properties.hibernate.hbm2ddl.auto=validate

spring.jackson.property-naming-strategy=com.fasterxml.jackson.databinc
```

4. Create the user profile API for client usage by declaring the `UserController` and `UserProfile`classes with the same content from the previous chapters. You can also view or download the source code for this recipe at https://github.com/PacktPublishing/OAuth-2.0-Cookbook within the `Chapter04/revoke-server` directory.
5. Create the sub-packages, `oauth/config` and `oauth/revoke`, within `com.packt.example.revokeserver`.
6. Declare the `OAuth2AuthorizationServer` and `OAuth2ResourceServer` classes within the `oauth/config` sub-package. You can copy these classes from

`rdbm-server` project, created on *Using relational database to store tokens and client details* recipe from Chapter 2, *Implementing Your Own OAuth 2.0 Provider* (In order to ease the development of this recipe, for didactical reason, I recommend you to remove the configuration that add support for encryption of client credentials data, which was declared within `OAuth2AuthorizationServer` class. Also, make sure to use plain text password for your client credentials).

7. The Authorization Server configuration should declare a `JdbcTokenStore` using an injected `DataSource` instance because this recipe is using the database previously created in *Using relational database to store tokens and client details* recipe from Chapter 2, *Implementing Your Own OAuth 2.0 Provider*.
8. Make sure that the Resource Server is protecting the `/api/**` endpoint.
9. In order to provide a complete token revocation feature, we need to separate the responsibilities between services for the access token and refresh token revocation. Create the following interface within the `oauth/revoke` package, which describes how the revocation services must work:

```
public interface RevocationService {
    void revoke(String token);
    boolean supports(String tokenTypeHint);
}
```

10. Then implement the previous interface for access the token revocation as follows (this class should be created in the same package as the interface `RevocationService`):

```
@Service
public class AccessTokenRevocationService implements RevocationService
    @Autowired
    private ConsumerTokenServices tokenService;
    @Override
    public void revoke(String token) {
        tokenService.revokeToken(token);
    }
    @Override
    public boolean supports(String tokenTypeHint) {
        return "access_token".equals(tokenTypeHint);
    }
}
```

11. And to support the refresh token revocation, create the following class within the same package used in the previous step:

```
@Service
public class RefreshTokenRevocationService implements RevocationServic
    @Autowired
    private TokenStore tokenStore;
    @Override
    public void revoke(String token) {
        if (tokenStore instanceof JdbcTokenStore) {
            JdbcTokenStore store = (JdbcTokenStore) tokenStore;
            store.removeRefreshToken(token);
        }
    }
    @Override
    public boolean supports(String tokenTypeHint) {
        return "refresh_token".equals(tokenTypeHint);
    }
}
```

12. To support the token revocation feature, we need to create an endpoint that could be accessed by any registered client application. But instead of using two endpoints to separate when to revoke access tokens or refresh tokens, we will count on the `token_type_hint` parameter defined by the *OAuth 2.0 Token Revocation* specification. But how would the application be able to inject one instance of `RevocationService` or another one based on this parameter? The answer to this question is to create the following `Factory` class within the sub-package, `oauth/revoke` (notice that the `create` method presented below, search through all services to find the one that supports a given hint):

```
@Component
public class RevocationServiceFactory {
    @Autowired
    private List<RevocationService> revocationServices;
    public RevocationService create(String hint) {
        return revocationServices.stream()
            .filter(service -> service.supports(hint))
            .findFirst().orElse(noopRevocationService());
    }
    private RevocationService noopRevocationService() {
        return new RevocationService() {
            public boolean supports(String hint) { return false; }
            public void revoke(String token) { }
        };
    }
}
```

13. Now we can create the following controller which will provide the endpoint `/oauth/revoke` using the `RevocationServiceFactory` to retrieve the appropriate `RevocationService` instance based on the `token_type_hint`

parameter:

```
@Controller
public class TokenRevocationController {
    @Autowired
    private RevocationServiceFactory revocationServiceFactory;
    @PostMapping("/oauth/revoke")
    public ResponseEntity<String> revoke(@RequestParam Map<String, Str
        RevocationService revocationService = revocationServiceFactory
            .create(params.get("token_type_hint"));
        revocationService.revoke(params.get("token"));
        return ResponseEntity.ok().build();
    }
}
```

14. But there is another important thing to do, that is to protect this endpoint so it can only accept requests from registered client applications. To do so, let's create the `CustomPathsConfigurer` class as presented next (it should be created within the same package as `TokenRevocationController`):

```
@Configuration
public class CustomPathsConfigurer extends WebSecurityConfigurerAdapte
    @Autowired
    private ClientDetailsService clientDetailsService;
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
        ClientDetailsUserDetailsService userDetailsService
            = new ClientDetailsUserDetailsService(clientDetailsService
        auth.userDetailsService(userDetailsService);
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .requestMatchers().antMatchers("/oauth/revoke").and()
            .httpBasic().and()
            .authorizeRequests().anyRequest().authenticated().and()
            .csrf().disable();
    }
}
```

15. Now the application is ready to be executed through the `mvn spring-boot:run` command.

How it works...

As Spring Security OAuth2 still does not provide the endpoint for token revocation, this recipe presented how to allow for token revocation while trying to be as close as possible to RFC 7009, which is the specification for OAuth 2.0 Token Revocation. This implementation is relying on the `token_type_hint` parameter which, as per specification, is an optional parameter (if the Authorization Server is able to automatically detect the type of the token to be revoked).

As you might have noticed, the `RevocationServiceFactory` creates an anonymous class for a non-operation revoking process which means to do nothing when a `token_type_hint` is not recognized by the Authorization Server. In addition, if a non-valid-token is sent to be revoked, the Authorization Server will return a 200 HTTP status because it means that the token is already an invalid one (so the mission to invalidate an access token is already achieved).

Start the application and try to request an access token using any grant type allowed for the client being used. Make sure you have a valid client registered at the `oauth_client_details` table within the `oauth2provider` database. Use the client details information retrieved by the following SQL command (you can use any other `client_id` you have registered before, but I am using `clientapp` and `123456` as a plain text `client_secret`):

```
select client_id, client_secret,
  scope, authorized_grant_types,
  web_server_redirect_uri
from oauth_client_details where client_id = 'clientapp';
```

Let's suppose that you have retrieved the access token `cd12bf75-e3dc-4de1-bdae-154680393a89`; how could you request the access token revocation? To do so, send the following request and you should expect a 200 status code as an HTTP response:

```
curl -v -X POST --user clientapp:123456 http://localhost:8080/oauth/revoke -H
```

If you want to revoke a refresh token, you just have to replace the `token_type_hint` parameter value from `access_token` to `refresh_token`.

Remote validation using token introspection

As an OAuth provider can be deployed as separate entities, the Resource Server should be able to validate access tokens by querying directly at a shared database or even by asking the Authorization Server through an available endpoint. To support this approach, the community has created an OAuth 2.0 profile called *OAuth 2.0 Token Introspection*, defined by the RFC 7662 specification which is available at https://tools.ietf.org/html/rfc7662. This recipe will present how to enable the Authorization Server to support the `/oauth/check_token` endpoint and how to configure `RemoteTokenServices` at the Resource Server, which can be used to remotely validate any presented access token.

Getting ready

Because of the usage of remote token validation, you will need to create two separate applications for the Authorization Server and the Resource Server. Both applications will rely on the database to store and share the Resource Owner's information instead of using Spring Boot auto-configurations through `application.properties` as we've been doing until now for most of the recipes. Besides these details, the applications will be built with Java 8 and Spring Boot using Spring Initializr to help the bootstrap stage. You can check the source code for this recipe on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook within the `Chapter04/remote-validation` directory.

How to do it...

The following steps will show you how to implement both the Authorization Server and Resource Server to use the remote token validation approach:

1. Create two applications using Spring Initializr named `remote-authserver` and `remote-resource`, both using the group ID `com.packt.example`. Both applications must use the dependencies `Web`, `Security`, `JPA`, and `MySQL`. These dependencies can be added at Spring Initializr.
2. Import both applications to your preferred IDE and make sure you add the following dependency into `pom.xml` for both projects:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

3. Create the table which will hold the Resource Owner's information by running the following SQL commands into the `oauth2provider` database:

```
create table resource_owner(
  id bigint auto_increment primary key,
  name varchar(200),
  username varchar(60),
  password varchar(100),
  email varchar(100)
);
```

4. Then instead of configuring the user's data within `application.properties`, run the following command to insert the user's data into the `resource_owner` table:

```
insert into resource_owner (name, username, password, email)
values ('Adolfo Eloy', 'adolfo', '123', 'adolfo@mailinator.com');
```

5. Now add the following database configuration for Authorization Server and Resource Server applications:

```
spring.datasource.url=jdbc:mysql://localhost/oauth2provider
spring.datasource.username=oauth2provider
spring.datasource.password=123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Di
spring.jpa.properties.hibernate.hbm2ddl.auto=validate
```

6. Make sure the Resource Server has the following property within the `application.properties` file so both applications can run on the same machine:

```
server.port=8081
```

7. Now let's focus on the Authorization Server side by creating the configuration class within the `remote-authserver` project as follows (create this class within the `com.packt.example.remoteauthserver` package). Note that we are not presenting how to configure a `JdbcTokenStore` and what must be injected for this snippet of code to work. That's because you already know how to do it, and if you don't know, you can go back to Ch apter 2, *Implementing Your Own OAuth 2.0 Provider* or to the source code at GitHub:

```
@Configuration @EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConf
    // datasource and authentication manager injection hidden for brev
    public void configure(AuthorizationServerEndpointsConfigurer endpc
        endpoints.authenticationManager(authenticationManager)
            .tokenStore(tokenStore());
    }
    public void configure(ClientDetailsServiceConfigurer clients) thrc
        clients.jdbc(dataSource);
    }
}
```

8. Now, the most important piece of code to enable the `/oauth/check_token` endpoint must be added within the `OAuth2AuthorizationServer` class which is the following method declaration:

```
@Override
public void configure(AuthorizationServerSecurityConfigurer security)
    security.checkTokenAccess("hasAuthority('introspection')");
}
```

9. And for the Resource Owner to be recognized by Spring Security, declare the `ResourceOwner`, `ResourceOwnerRepository`, `SecurityConfiguration`, and `Users` classes within the `com.packt.example.remoteauthserver.security` package.
10. Create the `ResourceOwner` class which represents the user's data (getters

and setters were omitted but you have to declare each one):

```
@Entity
public class ResourceOwner {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String username;
    private String password;
    private String email;
}
```

11. Create the `ResourceOwnerRepository` interface as follows:

```
public interface ResourceOwnerRepository extends CrudRepository<Resour
    Optional<ResourceOwner> findByUsername(String username);
}
```

12. Create the `Users` class which implements `UserDetailsService` to allow for user info validation (import the class `User` from `org.springframework.security.core.userdetails` package):

```
@Service
public class Users implements UserDetailsService {
    @Autowired
    private ResourceOwnerRepository repository;
    @Override
    public UserDetails loadUserByUsername(String username) throws User
        ResourceOwner resourceOwner = repository.findByUsername(userna
            .orElseThrow(() -> new RuntimeException());
        return new User(resourceOwner.getUsername(),
            resourceOwner.getPassword(), new ArrayList<>());
    }
}
```

13. Now that the Authorization Server is properly configured, let's move our attention to the Resource Server by creating the user's profile API in the same way we did for the *Revoking Issued Tokens* recipe declaring the `UserController` and `UserProfile` classes within the `api` sub-package of the `remote-resource` application.
14. Also, create a sub-package called `security` in the same way we did for the `remote-authserver` project in this recipe, and add the same classes we created for the Resource Owner's validation.
15. Then create the `OAuth2ResourceServer` class using the following configuration to protect the `/api/**` endpoint:

```
@Configuration
```

```
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/api/**");
    }
}
```

16. And finally, add the following bean declarations inside the `OAuth2ResourceServer` class to add support for remote access token validation:

```
@Autowired
private Users userDetailsService;
@Bean
public RemoteTokenServices remoteTokenServices() {
    RemoteTokenServices tokenServices = new RemoteTokenServices();
    tokenServices.setClientId("resource_server");
    tokenServices.setClientSecret("abc123");
    tokenServices.setCheckTokenEndpointUrl("http://localhost:8080/oaut
    tokenServices.setAccessTokenConverter(accessTokenConverter());
    return tokenServices;
}
@Bean
public AccessTokenConverter accessTokenConverter() {
    DefaultAccessTokenConverter converter = new DefaultAccessTokenConv
    converter.setUserTokenConverter(userTokenConverter());
    return converter;
}
@Bean
public UserAuthenticationConverter userTokenConverter() {
    DefaultUserAuthenticationConverter converter = new DefaultUserAuth
    converter.setUserDetailsService(userDetailsService);
    return converter;
}
```

17. As the Resource Server needs to validate access tokens against the Authorization Server check token endpoint, the resource server needs its own credentials. These credentials is configured in Spring Security OAuth2 as client credentials although the Resource Server isn't a client component from OAuth 2.0 specification. Make sure to run the following SQL command on MySQL to create valid credentials for to Resource Server:

```
INSERT INTO oauth_client_details (
 client_id, resource_ids, client_secret, scope, authorized_grant_types
 web_server_redirect_uri, authorities, access_token_validity,
 refresh_token_validity, additional_information, autoapprove)
VALUES (
 'resource_server', '', 'abc123', 'read_profile,write_profile',
 'authorization_code', 'http://localhost:9000/callback', 'introspectic
 null, null, null, '');
```

18. Now both applications are ready to be executed wherein each one runs on different ports. Start both applications by running the `mvn spring-boot:run` command.

How it works...

Because of the OAuth Provider being created with separate projects for the Authorization Server and Resource Server, we have to interact with both applications through different ports. For the Authorization Server, we will send the requests through port `8080`, although we have to use port `8081` to access the user's profile which is provided by the Resource Server. Let's make sure that you have well suited client credentials so you can perform some further tests. Run the following SQL command within MySQL console:

```
update oauth_client_details
set authorized_grant_types = 'authorization_code,password'
where client_id = 'clientapp';
```

To understand how it works, start both applications and let's start by sending the following request to retrieve an access token:

```
curl -X POST --user clientapp:123456 http://localhost:8080/oauth/token -H "ac
```

The previous command should return an access token that we might use to retrieve the user's profile by sending the following request (notice the `8081` port usage):

```
curl -H "Authorization: Bearer f2a0394f-f88f-4e7b-98ac-5ced5ecb73ca" "http://
```

By providing a `RemoteTokenServices` bean, Spring Security OAuth2 will start checking any given access token against the endpoint provided for the `checkTokenEndpointUrl` property (which in the case of this recipe is `http://localhost:8080/oauth/check_token`).

There's more...

Although the Resource Server is using the introspection endpoint to validate an access token, both applications are sharing databases to retrieve the user's data. It's still a good approach for security reasons because by breaking the OAuth Provider in two separate applications we are reducing the surface attack. But for complex enterprise solutions where many applications have to be integrated together, the best approach would be to rely on a Federation Identity Provider.

A Federation Identity Provider allows for a unified way for user provisioning binding distinct Identity Providers within the same company, for example. It's largely used to achieve a single-sign on leading to the advantage of having one single point for user management.

Another important thing to mention about using `/oauth/check_token` from Spring is that it does not implement the RFC 7662 at all. As you might notice, the response from `/oauth/check_token` does not return the active attribute that is required by the specification, as can be viewed at https://tools.ietf.org/html/rfc7662#section-2.2.

Improving performance using cache for remote validation

This recipe will show how to improve the performance of the Resource Server when it has to remotely validate access tokens. To help on performance, we will use a cache strategy to avoid making requests to `/oauth/check_token` every time an OAuth's protected resource is requested.

Getting ready

This recipe is an improvement on the previous recipe that we will create a new Resource Server which will cache issued access tokens. So to run this recipe, we need the application `remote-authserver` created for the *Remote validation using token introspection* recipe. This recipe will rely on Redis for cache, the MySQL database, and will be developed using Java 8 with Spring Boot.

How to do it...

Follow the next steps to improve the performance of a Resource Server that uses `RemoteTokenValidation` to check for access token validity with Spring Security OAuth2:

1. You can just copy the `remote-resource` application created for the *Remote Validation using token introspection* recipe or you can also create an application using Spring Initializr. Just make sure this application is named `cache-introspection` so that we can reference the same application name throughout this recipe.

2. Also make sure to include these dependencies: `Web`, `Security`, `JPA`, and `MySQL`. Do not forget the `spring-security-auth2` dependency within the `pom.xml` file.
3. All the classes and configurations must be the same from `remote-resource`. The differences start by adding the following dependency because we will use Redis to cache access tokens:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

4. Now create a cache configuration through the following class (this class should be created within the `cache` sub-package):

```
@Configuration @EnableCaching
public class CacheConfiguration {
    @Autowired
    private RedisTemplate<Object, Object> redisTemplate;
    @Bean
    public CacheManager cacheManager() {
        RedisCacheManager cacheManager = new RedisCacheManager(redisTe
        cacheManager.setCacheNames(Arrays.asList("oauth2"));
        cacheManager.setUsePrefix(true);
        cacheManager.setDefaultExpiration(60);
        return cacheManager;
        }
    }
```

5. After creating the cache configuration, let's create an extension of `RemoteTokenServices`, which will use the cache strategy that we have defined. Create the following class within the `oauth` sub-package (I have added the import statements just for classes that will conflict when automatically importing classes within your IDE):

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.security.core.AuthenticationException;

public class CustomRemoteTokenServices extends RemoteTokenServices {
    private RemoteTokenServices remoteTokenServices;
    public CustomRemoteTokenServices(RemoteTokenServices remoteTokenSe
        this.remoteTokenServices = remoteTokenServices;
    }
    @Override
    @Cacheable("oauth2")
    public OAuth2Authentication loadAuthentication(String accessToken)
        throws AuthenticationException, InvalidTokenException {
        return remoteTokenServices.loadAuthentication(accessToken);
    }
    @Override
    public OAuth2AccessToken readAccessToken(String accessToken) {
        return remoteTokenServices.readAccessToken(accessToken);
    }
}
```

6. Then in the `OAuth2ResourceServer` configuration class, you have to declare `RemoteTokenServices` as follows:

```
@Bean
public RemoteTokenServices remoteTokenServices() {
    RemoteTokenServices tokenServices = new RemoteTokenServices();
    tokenServices.setClientId("resource_server");
    tokenServices.setClientSecret("abc123");
    tokenServices.setCheckTokenEndpointUrl("http://localhost:8080/oaut
    tokenServices.setAccessTokenConverter(accessTokenConverter());
    return new CustomRemoteTokenServices(tokenServices);
}
```

7. Start Redis by running the `redis-server` command.
8. Start the application by running the`mvn spring-boot:run`command,create an access token and try to access the user's profile API using the issued access token.
9. Start the Redis client application by running `redis-cli`.
10. Then if you run the command `keys *` within the Redis console, you

should see something similar to the following:

```
"oauth2:\xac\xed\x00\x05t\x00$f2a0394f-f88f-4e7b-98ac-5ced5ecb73ca"
```

How it works...

Basically, this recipe presents an application that works in a very similar way as presented in the *Remote validation using token introspection* recipe. The difference now is that we are wrapping an instance of `RemoteTokenServices` within the `CustomRemoteTokenServices` class. It gives us the opportunity to cache the invocation of`loadAuthenticationMethod` methodthat could not be done when using `RemoteTokenServices` directly (which is a piece of code provided by Spring Security OAuth2).

The `CustomRemoteTokenServices` class, delegates all method invocations to an instance of `RemoteTokenServices` that executes the already known flow for remote token validation. It's also important to note the usage of Redis for cache and how it was configured, as presented in the following code:

```
RedisCacheManager cacheManager = new RedisCacheManager(redisTemplate);
cacheManager.setCacheNames(Arrays.asList("oauth2"));
cacheManager.setUsePrefix(true);
cacheManager.setDefaultExpiration(60);
```

For didactical purposes we aren't protecting Redis database properly. Make sure to use SSL to protect Redis communication and/or configure authentication. When using Spring Boot you can configure the following properties:

```
# Login password for redis server
spring.redis.password=password
# Enable SSL support
spring.redis.ssl=true
```

Notice that the `cacheNames` property defined as `oauth2` is the same name that we found when running the command `keys *` within the Redis Client console. Another important thing to note is the `defaultExpiration` property. Here we are using 60 seconds, but this time should vary depending on the access token validity time. The recommended cache eviction time is to be short-lived relative to the access token expiration time. We have to do it to avoid revoking an access token that is still cached.

The usage of cache might be contradictory in some situations but it's needed when there is a lot of network traffic. This approach is also recommended by the official Spring Security OAuth2 documentation at http://projects.spring.io/spring-security-oauth/docs/oauth2.html.

See also

- Remote validation using token introspection

Using Gatling to load test remote token validation

The previous recipe, presented how to improve the performance by using cache. But although everything is working fine, how can we measure the gaining performance from cache usage? This recipe will present you an execution of a Gatling project to measure the token validation process with and without cache.

Getting ready

To run this recipe, you will need the `remote-authserver`,`remote-resource` and `cache-introspection` applicationscreated for the previous two recipes. At first, we will run the load test against `remote-authserver` and `remote-resource`. And to measure the cache advantages, we will run the same load test against the `remote-authserver` and `cache-introspection` applications. To edit the load test scripts, you need Scala IDE for Eclipse (if you don't know how to configure and use Scala IDE for Eclipse, you can check the *Using Gatling to load test the token validation process using shared databases* recipe from Chapter 2, *Implementing Your Own OAuth 2.0 Provider*).

How to do it...

To run this load test, you have to create a Scala project in the same way as we did in the *Using Gatling to load test the token validation process using shared databases*recipe from Chapter 2, *Implementing Your Own OAuth 2.0 Provider:*

1. Clone or download the project available at https://github.com/adolfoweloy/scala-maven-skel. As you will notice, this project is just a skeleton for the recipe that we will be creating now.
2. Now rename the `scala-maven-skel` project to `load-testing-remote` by running the `mv scala-maven-skel load-testing-remote` command.
3. Start the `remote-authserver` application by running `mvn spring-boot:run` within the project's directory at the terminal.
4. Go to the `load-testing-remote` directory and open `pom.xml` with any editor you want (I recommend using VIM or any lightweight editor before importing the project).
5. Make sure the `artifactId` tag has the name `load-testing-remote` instead of `scala-maven-skel`.
6. Import this project to Scala IDE for Eclipse and copy the `OAuth.scala` file from the load-testing project created in Chapter 2, *Implementing Your Own OAuth 2.0 Provider* to `load-testing-remote` within the `src/main/scala/oauth2` folder (the structure should be the same as the `load-testing` project).
7. Make sure you have all the dependencies declared for the `load-testing` project from Chapter 2, *Implementing Your Own OAuth 2.0 Provider*.
8. As we will run more than one scenario, enable multiple simulations by adding the following snippet of code inside the `plugin` tag for `gatling-maven-plugin`:

```
<configuration>
 <runMultipleSimulations>true</runMultipleSimulations>
</configuration>
```

9. Create the `gatling.conf` file inside `src/test/resources` with the following

content:

```
gatling {
  core {
    outputDirectoryBaseName = "result"
    runDescription = "Token validation using Redis as shared database"
    encoding = "utf-8"
    directory {
      data = src/test/resources
      results = target/gatling
      bodies = src/test/resources
      binaries = target/classes
    }
  }
  charting {
    indicators {
      lowerBound = 5
      higherBound = 10
    }
  }
}
```

10. Create the `AuthorizationScenarios.scala` file within the `src/test/main/oauth2provider` directory, as presented in the following code:

```
import oauth2.OAuth
import io.gatling.core.Predef._
import io.gatling.http.Predef._
import org.json4s.DefaultFormats
import org.json4s.native.JsonMethods.parse

object AuthorizationScenarios {
  case class Token(access_token:String)
  implicit val formats = DefaultFormats
  val jsValue = parse(OAuth.getToken())
  val accessToken = "Bearer " + jsValue
    .extract[Token].access_token
  var withoutCache = scenario("Without cache")
      .exec(http("Validate access token without cache")
        .get("/api/profile")
        .header("Authorization", accessToken))

  var withCache = scenario("Using cache")
      .exec(http("Validate access token with cache")
        .get("/api/profile")
        .header("Authorization", accessToken))
}
```

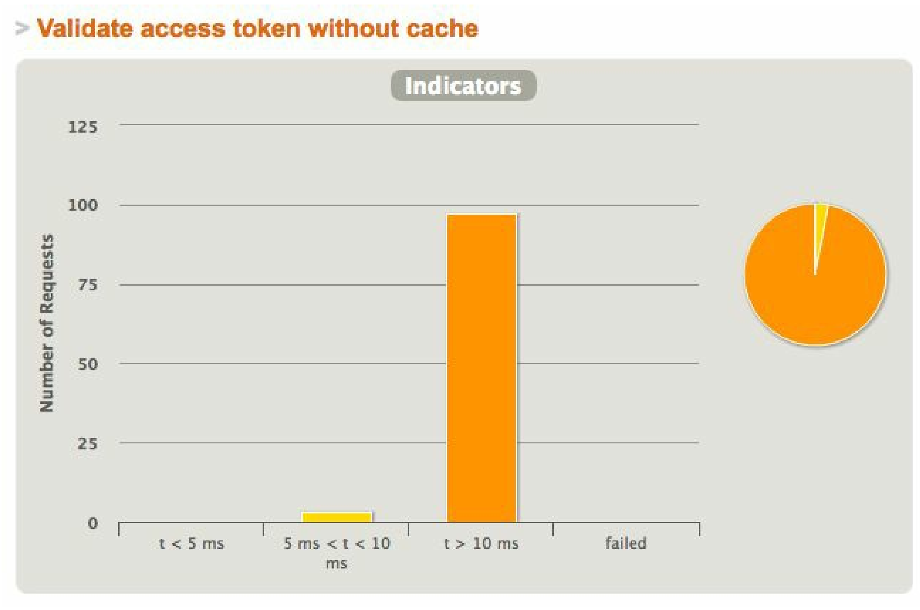
11. Create the `WithCacheSimulation.scala` file within `src/test/scala/oauth2provider` with the following content:

```
import scala.concurrent.duration._
import io.gatling.core.Predef._
import io.gatling.http.Predef._
class WithCacheSimulation extends Simulation {
  val httpConfCache = http.baseURL("http://localhost:8082")
  val withCacheScenario = List(
    AuthorizationScenarios.withCache.inject(rampUsers(100) over(10 sec
  setUp(withCacheScenario).protocols(httpConfCache)
}
```

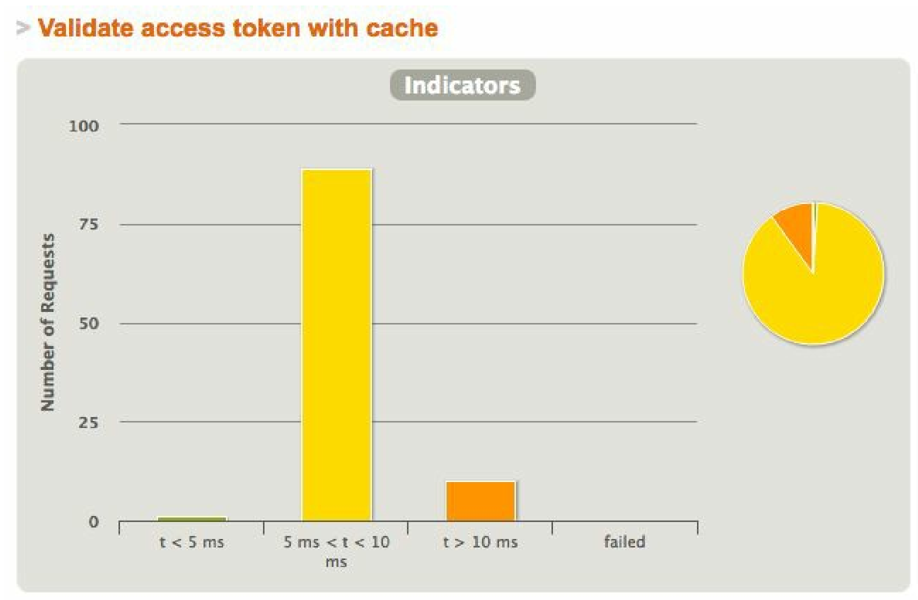
12. And for a simulation without cache, create the `WithoutCacheSimulation.scala` file within `src/test/scala/oauth2provider` as presented in the following code:

```
import scala.concurrent.duration._
import io.gatling.core.Predef._
import io.gatling.http.Predef._
class WithoutCacheSimulation extends Simulation {
  val httpConf = http.baseURL("http://localhost:8081")
  val withoutCacheScenario = List(
    AuthorizationScenarios.withoutCache.inject(rampUsers(100) over(10
  setUp(withoutCacheScenario).protocols(httpConf)
}
```

13. Then change the property `server.port` from the `8081` to `8082` from `cache-introspection` project. This property is defined within the `application.properties` file.
14. Then start the `remote-authserver`, `remote-resource`, and `cache-introspection` applications by running `mvn spring-boot:run` inside each of the respective directories.
15. Then go to the `load-testing-remote` project's directory and run the `mvn clean install` command.
16. After running the load-testing, open each generated HTML report for each simulation to compare the executions.
17. After running, on my machine, I have got the following results for `WithoutCacheSimulation`:



18. And for `WithCacheSimulation`, I have got the following results:



19. Notice the difference in that when using cache, most requests take between 5 and 10 ms to be answered instead of almost 100% of responses taking more than 10 ms.

There's more...

All the load testing in this recipe is running against a Resource Server which runs separately from the Authorization Server. Although the performance was improved by adding cache support, both the applications are running on the same machine (both are running on the localhost). Because of these conditions, we are not considering the network latency which should have been taken into consideration.

See also

- Remote validation using token introspection
- Improving performance using cache for remote validation

Dynamic client registration

This recipe will present you with steps to improve an Authorization Server by adding support for dynamic client registration. This profile might be critical for native clients that require to be deployed without packing client credentials as part of the application. It's recommended that mobile client applications use dynamic registration to increase safety.

Getting ready

To run this recipe, you will need Java 8, Maven, MySQL, and your preferred IDE. As we will use Spring Boot, it's also recommended that you create an initial project using Spring Initializr.

How to do it...

The next steps will present you with how you can create an Authorization Server that implements a use case from the dynamic client registration profile:

1. Create the project using Spring Initializr. Go to https://start.spring.io/ and fill out the form using the following data:
   - Set up the Group as `com.packt.example`
   - Define the Artifact as `dynamic-server` and add `Web`, `Security`, `JPA`, and `MySQL` as dependencies for this project
2. Open the `pom.xml` file and add the following dependency, as we will use the Spring Security OAuth2 project:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

3. So that the client can be properly registered at the Authorization Server, we need to provide a database and related tables to persist, client details and access token related data. To do so, let's use the same database described in Chapter 2, *Implementing Your Own OAuth 2.0 Provider* for the recipe *Using relational database to store tokens and client details*. When dynamically registering a client, the metadata sent to the endpoint registration will be persisted in the `oauth_client_details` table from the `oauth2provider` database.

4. Open the `application.properties` file and add the following content for the data source configuration:

```
spring.datasource.url=jdbc:mysql://localhost/oauth2provider
spring.datasource.username=oauth2provider
spring.datasource.password=123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Di
spring.jpa.properties.hibernate.hbm2ddl.auto=validate
```

5. Besides dynamic client registration, we will also provide an API for

client usage so you can try to interact with OAuth's protected resources. We will use the same API used for the previous recipes, which allows for Resource Owner profile retrieving. Create the `UserController` and `UserProfile` classes within a new sub-package named `api` and copy the content of these classes from *Protecting resources using Authorization Code grant type* recipe at Chapter 2, *Implementing Your Own OAuth 2.0 Provider*. You can also view or download the source code for this recipe at https://github.com/PacktPublishing/OAuth-2.0-Cookbook within the `Chapter04/dynamicserver` directory.

6. Let's start creating some configuration classes. The first to be created will be `WebSecurityConfiguration`, as follows. This class should be created inside the `security` sub-package:

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfiguration extends WebSecurityConfigurerAda
    protected void configure(AuthenticationManagerBuilder auth) throws
        auth.inMemoryAuthentication()
            .withUser("adolfo").password("123").authorities(new ArrayI
    }
    protected void configure(HttpSecurity http) throws Exception {
        http
            .requestMatchers().antMatchers("/clients", "/oauth/authori
            .httpBasic().and()
            .authorizeRequests().anyRequest().authenticated().and()
            .csrf().disable();
    }
}
```

7. Create the Authorization Server configuration as follows within the package `com.example.dynamicserver.oauth.config` (when importing `DataSource` class, import from `javax.sql.DataSource`):

```
@Configuration @EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConf
    @Autowired private DataSource dataSource;
    @Bean public TokenStore tokenStore() {
        return new JdbcTokenStore(dataSource);
    }
    @Bean public ApprovalStore approvalStore() {
        return new JdbcApprovalStore(dataSource);
    }
    public void configure(AuthorizationServerEndpointsConfigurer endpc
        endpoints.approvalStore(approvalStore()).tokenStore(tokenStore
    }
    public void configure(ClientDetailsServiceConfigurer clients) thrc
        clients.jdbc(dataSource);
    }
```

```
        @Bean
        public ClientRegistrationService clientRegistrationService() {
            return new JdbcClientDetailsService(dataSource);
        }
    }
```

8. Create the Resource Server configuration as follows in the same package as the Authorization Server configuration:

```
    @Configuration @EnableResourceServer
    public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
        public void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                    .antMatchers("/register").permitAll()
                    .anyRequest().authenticated().and()
                .requestMatchers().antMatchers("/api/**");
        }
    }
```

9. Before creating all the classes needed for the client registration, let's first create an important class that will help the Authorization Server to generate `client_id` and `client_secret`. Create the `RandomHelper` class as presented in the following code:

```
    package com.example.dynamicserver.util;
    @Component
    public class RandomHelper {
        private SecureRandom random;
        public RandomHelper() { random = new SecureRandom(); }
        public String nextString(int size, int radix) {
            double numberOf = Math.log10(radix) / Math.log10(2);
            int bits = (int) (size * numberOf);
            int mod = (int) (bits % numberOf);
            return new BigInteger(bits + mod, random).toString(radix);
        }
    }
```

10. Then, let's start creating the classes that represents the domain model for the dynamic client registration process. All the classes that represents the domain logic will be created within the `com.example.dynamicserver.registration` package. At first, create the `ClientRegistrationRequest` class that will be used as a data structure to carry client details for dynamic registering (do not forget to create the respective getters and setters that were omitted for brevity, and when using `@JsonProperty` make sure to use the annotation from `com.fasterxml.jackson.annotation` package):

```
public class ClientRegistrationRequest {
   @JsonProperty("redirect_uris")
   private Set<String> redirectUris = new HashSet<>();
   @JsonProperty("grant_types")
   private Set<String> grantTypes = new HashSet<>();
   @NotBlank
   @JsonProperty("client_name")
   private String clientName;
   @NotBlank
   @JsonProperty("client_uri")
   private String clientUri;
   @NotBlank
   private String scope;
   @NotBlank
   @JsonProperty("software_id")
   private String softwareId;
   // getters and setters ommited
}
```

11. And now, we will implement the `ClientDetails` interface which maps each attribute to the fields in the `oauth_client_details` table from the `oauth2provider` database. Create the `DynamicClientDetails` class as follows:

```
public class DynamicClientDetails implements ClientDetails {
    private String clientId;
    private Set<String> resourceIds = new HashSet<>();
    private Set<String> authorizedGrantTypes = new HashSet<>();
    private String clientSecret;
    private Set<String> scopes = new HashSet<>();
    private Set<String> registeredRedirectUri = new HashSet<>();
    private Set<GrantedAuthority> authorities = new HashSet<>();
    private Integer accessTokenValiditySeconds;
    private Integer refreshTokenValiditySeconds;
    private Map<String, Object> additionalInformation = new HashMap<>(
}
```

12. All the fields added for `DynamicClientDetails`, you should already know (because we have configured client details before) about from Chapter 2, *Implementing Your Own OAuth 2.0 Provider*. To be in accordance with RFC 7591 which is the specification for dynamic client registration, we have to add some additional fields. Let's create these additional fields and take advantage of the `additionalInformation` attribute to save their respective values. Add the following attributes within `DynamicClientDetails`:

```
private String softwareId;
private String tokenEndpointAuthMethod;
private Set<String> responseTypes = new HashSet<>();
private String clientName;
private String clientUri;
```

```
    private long clientSecretExpiresAt;
```

13. Create the following setters (the getters for these fields should be ordinary getters that just return their respective attribute values):

```
    public void setSoftwareId(String softwareId) {
        this.softwareId = softwareId;
        additionalInformation.put("software_id", softwareId);
    }
    public void setTokenEndpointAuthMethod(String tokenEndpointAuthMethod)
        this.tokenEndpointAuthMethod = tokenEndpointAuthMethod;
        additionalInformation.put("token_endpoint_auth_method", tokenEndpc
    }
    public void setResponseTypes(Set<String> responseTypes) {
        this.responseTypes = responseTypes;
        additionalInformation.put("response_types", getResponseTypes());
    }
    public void setClientName(String clientName) {
        this.clientName = clientName;
        additionalInformation.put("client_name", clientName);
    }
    public void setClientUri(String clientUri) {
        this.clientUri = clientUri;
        additionalInformation.put("client_uri", clientUri);
    }
    public void setClientSecretExpiresAt(long clientSecretExpiresAt) {
        this.clientSecretExpiresAt = clientSecretExpiresAt;
        additionalInformation.put("client_secret_expires_at",
            Long.toString(clientSecretExpiresAt));
    }
```

14. Create getters and setters for the remaining attributes.
15. Make sure the following attribute accessors and modifiers are the same for your `DynamicClientDetails` implementation:

```
    public boolean isSecretRequired() {
        return authorizedGrantTypes.containsAll(
            Arrays.asList(
               "authorization_code", "password", "client_credentials"));
    }

    public boolean isScoped() { return scopes.size() > 0; }

    public Set<String> getScope() { return scopes; }

    public boolean isAutoApprove(String scope) { return false; }

    public void addAuthorizedGrantTypes(String... authorizedGrantTypes) {
        for (String grantType : authorizedGrantTypes) {
            this.authorizedGrantTypes.add(grantType);
        }
    }
    public void addScope(String scope) {
        scopes.add(scope);
    }
    public void addRegisteredRedirectUri(String uri) {
        registeredRedirectUri.add(uri);
    }
```

16. Create the `DynamicClientDetailsFactory` class as follows:

```
    @Component
    public class DynamicClientDetailsFactory {
        private RandomHelper randomHelper;
        @Autowired
        DynamicClientDetailsFactory(RandomHelper randomHelper) {
            this.randomHelper = randomHelper;
        }
    }
```

17. Add the following method that is in charge of creating an instance of `DynamicClientDetails` from `ClientRegistrationRequest` that should hold data sent to the registration endpoint:

```
    public DynamicClientDetails create(ClientRegistrationRequest request)
        DynamicClientDetails clientDetails = new DynamicClientDetails();
        clientDetails.setClientName(request.getClientName());
        clientDetails.setClientUri(request.getClientUri());
        clientDetails.setSoftwareId(request.getSoftwareId());
        setClientCredentials(request, clientDetails);
        request.getRedirectUris().forEach(
                uri -> clientDetails.addRegisteredRedirectUri(uri));
        if (request.getScope() != null) {
            for (String scope : request.getScope().split("\\s")) {
                clientDetails.addScope(scope);
            }
        }
        if (request.getGrantTypes().size() == 0) {
            clientDetails.addAuthorizedGrantTypes("authorization_code");
        } else {
            request.getGrantTypes().forEach(
                grantType -> clientDetails.addAuthorizedGrantTypes(grantTy
        }
        clientDetails.setTokenEndpointAuthMethod("client_secret_basic");
        if (clientDetails.getAuthorizedGrantTypes().contains("implicit"))
            clientDetails.getResponseTypes().add("token");
        }
        if (clientDetails.getAuthorizedGrantTypes().contains("authorizatic
            clientDetails.getResponseTypes().add("code");
        }
        return clientDetails;
    }
```

18. The previous code is using a private method that was not declared yet. So let's declare the private method, `setClientCredentials`, as follows:

```
private void setClientCredentials(
    ClientRegistrationRequest clientMetadata,
    DynamicClientDetails clientDetails) {
    clientDetails.setClientId(randomHelper.nextString(10, 32));
    Set<String> grantTypes = clientMetadata.getGrantTypes();
    long otherThanImplicit = grantTypes.stream()
        .filter(grantType -> !grantType.equals("implicit")).count();
    if (otherThanImplicit > 0 || grantTypes.size() == 0) {
        clientDetails.setClientSecret(randomHelper.nextString(32, 32))
        LocalDateTime after30Days = LocalDateTime.now().plusDays(30);
        clientDetails.setClientSecretExpiresAt(
            after30Days.atZone(ZoneId.systemDefault()).toEpochSecond()
    }
}
```

19. To validate some rules that are specified by the dynamic client registration protocol, this recipe presents one specification class which is responsible to decide if a specific rule is satisfied by any given client metadata.

20. Create the `RedirectFlowSpecification` class because it's important to validate if the client to be registered is using any flow that relies on redirection and is sending at least one redirect URI to receive callbacks:

```
@Component
public class RedirectFlowSpecification {
    public boolean isSatisfiedBy(ClientRegistrationRequest clientMetac
        List<String> flowsWithRedirection = Arrays.asList(
            "authorization_code", "implicit");
        boolean hasFlowWithRedirection = clientMetadata.getGrantTypes(
            .filter(grantType -> flowsWithRedirection.contains(grantTy
            .findAny().isPresent();
        if (hasFlowWithRedirection) {
            return clientMetadata.getRedirectUris().size() > 0;
        }
        return false;
    }
}
```

21. Regarding all the validations, the Authorization Server needs to provide some error information in case something goes wrong in the registration process. Create the `RegistrationError` class presented in the following code that has the data structure proposed by the dynamic client registration protocol:

```
public class RegistrationError {
    public final static String INVALID_CLIENT_METADATA = "invalid_clie
    public final static String INVALID_REDIRECT_URI = "invalid_redirec
    public final static String INVALID_SOFTWARE_STATEMENT = "invalid_s
```

```
        public final static String UNAPPROVED_SOFTWARE_STATEMENT = "unappr
        private String error;
        private String errorDescription;
        public RegistrationError(String error) {
            this.error = error;
        }
        public String getError() {
            return error;
        }
        public String getErrorDescription() {
            return errorDescription;
        }
        public void setErrorDescription(String errorDescription) {
            this.errorDescription = errorDescription;
        }
    }
```

22. When a Client is successfully registered, the server being created must return some data about the recently created client details. So, create the class `ClientRegistrationResponse` as presented below, within the sub-package `oauth/registration` (also create the respective getters and setters for each attribute):

```
    public class ClientRegistrationResponse {
        @JsonProperty("redirect_uris")
        private Set<String> redirectUris = new HashSet<>();
        @JsonProperty("token_endpoint_auth_method")
        private String tokenEndpointAuthMethod;
        @JsonProperty("grant_types")
        private Set<String> grantTypes = new HashSet<>();
        @JsonProperty("response_types")
        private Set<String> responseTypes = new HashSet<>();
        @JsonProperty("client_name")
        private String clientName;
        @JsonProperty("client_uri")
        private String clientUri;
        private String scope;
        @JsonProperty("software_id")
        private String softwareId;
        @JsonProperty("client_id")
        private String clientId;
        @JsonProperty("client_secret")
        private String clientSecret;
        @JsonProperty("client_secret_expires_at")
        private long clientSecretExpiresAt;
        // getters and setters omitted
    }
```

23. Now create the `DynamicClientRegistrationController` class to handle all the registration process with the following attributes (this class should be created within the `com.example.dynamicserver.web` package):

```
@Controller
public class DynamicClientRegistrationController {
    @Autowired
    private ClientRegistrationService clientRegistration;
    @Autowired
    private DynamicClientDetailsFactory clientDetailsFactory;
    @Autowired
    private RedirectFlowSpecification redirectFlowSpecification;
    // methods declared on next steps
}
```

24. Then create the following method within `DynamicClientRegistrationController` to handle the `/register` endpoint:

```
@PostMapping("/register")
public ResponseEntity<Object> register(@RequestBody ClientRegistratior
    if (!redirectFlowSpecification.isSatisfiedBy(clientMetadata)) {
        RegistrationError error = new RegistrationError(
            RegistrationError.INVALID_REDIRECT_URI);
        error.setErrorDescription(
          "You must specify redirect_uri when using flows with redirec
        return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);
    }
    DynamicClientDetails clientDetails = clientDetailsFactory.create(c
    clientRegistration.addClientDetails(clientDetails);
    return new ResponseEntity<>(createResponse(clientDetails), HttpSta
}
```

25. Create the following private method to return an instance of `ClientRegistrationResponse`, which is just a data structure for presenting the newly registered client:

```
private ClientRegistrationResponse createResponse(DynamicClientDetails
    ClientRegistrationResponse response = new ClientRegistrationRespor
    response.setClientId(clientDetails.getClientId());
    response.setClientSecret(clientDetails.getClientSecret());
    response.setClientName(clientDetails.getClientName());
    response.setClientUri(clientDetails.getClientUri());
    response.setGrantTypes(clientDetails.getAuthorizedGrantTypes());
    response.setRedirectUris(clientDetails.getRegisteredRedirectUri())
    response.setResponseTypes(clientDetails.getResponseTypes());
    response.setScope(clientDetails.getScope().stream().reduce((a, b)
    response.setSoftwareId(clientDetails.getSoftwareId());
    response.setTokenEndpointAuthMethod(clientDetails.getTokenEndpoint
    response.setClientSecretExpiresAt(clientDetails.getClientSecretExp
    return response;
}
```

26. Add the following method just for testing purposes. It will allow us to check for registered clients:

```
    @GetMapping("/clients")
    public ResponseEntity<List<ClientDetails>> list() {
        return new ResponseEntity<>(clientRegistration.listClientDetails()
    }
```

How it works...

This project acts as an OAuth 2.0 Provider which allows for dynamic client registration, so it has the following features:

- It allows for the resource owner to be authenticated
- It allows for the resource owner to grant permissions for third-party application to access its resources
- It allows for dynamic client registration through the endpoint `http://localhost:8080/register`
- It protect the user's resource through OAuth 2.0

Start the application through the `mvn spring-boot:run` command and to dynamically register a client, execute the following command in your terminal:

```
curl -X POST http://localhost:8080/register -H "Content-Type: application/jsc
```

The result of this command should return the HTTP status 201 with the following body content:

```
{
    "redirect_uris": [
        "http://localhost:9000/callback"
    ],
    "token_endpoint_auth_method": "client_secret_basic",
    "grant_types": [
        "implicit",
        "authorization_code"
    ],
    "response_types": [
        "code",
        "token"
    ],
    "client_name": "flavio",
    "client_uri": "http://localhost:9000",
    "scope": "read_profile write_profile",
    "software_id": "f6b35c96-b61f-4e4a-b6bc-7ab38650ab61",
    "client_id": "lm5f0pvrhn",
    "client_secret": "c3697smh9brp9abiakqfnhoefobtjaf9",
    "client_secret_expires_at": 1505793646
}
```

Notice that we didn't have to implement any `ClientDetailsService` because we can rely on the `JdbcClientDetailsService` class, although we had to write a lot of code. At the time of this writing, the Spring Security OAuth2 project neither supports RFC 7591 nor RFC 7592 which allows for dynamic client registration management.

There's more...

This recipe has presented you with an open registration process without using any software statement to check for validity and integrity of the client's metadata information. To implement the software statement feature, you can use the signed JWT created from the client metadata, but to better understand how to use JWT go through the next chapters.

See also

- Specification for dynamic client registration described in RFC 7591 at https://tools.ietf.org/html/rfc7591.

Self Contained Tokens with JWT

This chapter will cover the following recipes:

- Generating access tokens as JWT
- Validating JWT tokens at the Resource Server side
- Adding custom claims on JWT
- Asymmetric signing of a JWT token
- Validating asymmetric signed JWT token
- Using JWE to cryptographically protect JWT tokens
- Using JWE at the Resource Server side
- Using proof-of-possession key semantics on OAuth 2.0 Provider
- Using proof-of-possession key on the client side

Introduction

This chapter introduces how **JSON Web Tokens** (**JWT**) can be safely used to carry client information to allow Resource Servers to locally validate access tokens. This chapter will guide you through the usage of JWT and its corresponding representations such as JWS and JWE that respectively define integrity protection and confidentiality of the JWT payload. There are also some advanced topics being covered, such as how to use asymmetric signatures and how the client can prove the possession of a given access token.

It's important to bear in mind that even though we are signing or encrypting the JWT payload, all the connections must be performed using TLS/SSL in production. We are not using TLS/SSL just because of didactical reasons.

Generating access tokens as JWT

Even though JSON web tokens are largely used to represent self contained access tokens through OAuth 2.0 solutions, its specifications (defined by RFC 7519 that's available at https://tools.ietf.org/html/rfc7519) clearly define that its purpose is to represent claims to be transferred between parties. Briefly, this means that JWTs can be used both for authentication or information exchange as described at https://jwt.io/introduction. Because of the structure of a JWT, it is possible for the Resource Server to extract client information and even data about the subject (which commonly means the Resource Owner) and validate the access token locally, instead of having to use a shared database or validate remotely through the usage of token introspection strategy. The claims at JWT are represented as JSON payload, which can be HMAC signed or encrypted.

*The **hash-based message authentication code (HMAC)** is a portion of information that can be used to verify some data integrity created by the process of signature generated from some cryptographic hash function such as MD5 and SHA256.*

This recipe will present you with how to use Spring Security OAuth2 and related libraries to allow the Authorization Server to issue JWT access tokens.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies as `Web` and `Security` (that will declare properly all the spring boot starters needed for this recipe). Do not forget to set up the Artifact and Group names.

How to do it...

This recipe creates the `auth-server-jwt` project, which is available at GitHub in the `Chapter05` folder. Import the generated project as a Maven project into your IDE and follow the following steps:

1. Open the `pom.xml` file and add the following extra dependencies for Spring Security OAuth2 and Spring Security JWT (note that we are declaring the most up to date version of Spring Security OAuth2 at the time of writing):

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$-->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
```

2. Add the following content to the `application.properties` file:

```
security.user.name=adolfo
security.user.password=123
```

3. Now at the first moment, declare the `OAuth2AuthorizationServer` class within the `com.packt.example.authserverjwt.oauth` package with the following content:

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConf
    @Autowired
    private AuthenticationManager authenticationManager;
    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpo
        endpoints
            .authenticationManager(authenticationManager);
    }
    @Override
    public void configure(ClientDetailsServiceConfigurer clients) thro
        clients.inMemory()
            .withClient("clientapp").secret("123456")
```

```
                    .scopes("read_profile")
                    .authorizedGrantTypes("password", "authorization_code");
            }
        }
```

4. The `OAuth2AuthorizationServer` class is declaring a simple version of Authorization Server which generates bearer tokens without any embedded content as we are about to do with JWT. To start generating JWT access tokens, declare the following beans within `OAuth2AuthorizationServer` as presented in the following code:

```
        @Bean
        public JwtAccessTokenConverter accessTokenConverter() {
            JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
            converter.setSigningKey("non-prod-signature");
            return converter;
        }
        @Bean
        public JwtTokenStore jwtTokenStore() {
            return new JwtTokenStore(accessTokenConverter());
        }
```

5. Then replace the configure method where we define the `authenticationManager` with the following source code to define the `tokenStore` and the `accessTokenConverter` instances for `AuthorizationServerEndpointsConfigurer`:

```
        @Override
        public void configure(AuthorizationServerEndpointsConfigurer endpoints
            throws Exception {
            endpoints
                .authenticationManager(authenticationManager)
                .tokenStore(jwtTokenStore())
                .accessTokenConverter(accessTokenConverter());
        }
```

6. Now the Authorization Server is ready to run and generate JWT access tokens. Let's move on to the next topics to understand how `JwtTokenStore` and `JwtAccessTokenConverter` helps us to generate JWT token that will be symmetric signed (which leads to a JWS type of JWT).

How it works...

As you might have noticed, the main differences between the declared Authorization Server and the normal version (that is, Authorization Servers that do not generate JWT access tokens), is that we are using special implementations of `TokenStore` and `AccessTokenConverter` interfaces.

Note that we are defining the signing key with the text non-prod-signature. It would be better to be declared as a property entry or any other place that is not available to read. This sample code declares the signing key directly in the source code just for didactical reasons.

The token store we are using now, does not store anything by itself. Instead of it, this class is just in charge of reading any given JWT content, and to do that the `JwtTokenStore` also counts with the `JwtAccessTokenConverter`, which implements not only the `AccessTokenConverter`, but also the `TokenEnhancer` interface.

Besides the classes being used, what is going to be different for the client application that uses the self contained access token? Basically it won't demand any changes to the client application because the access token remains opaque to the client. But structurally we can see the differences by requesting a new access token. Try running the following command to request an access token using the Resource Owner Password Credentials grant type and let's check the results:

```
curl -X POST --user clientapp:123456 -H "Content-Type: application/x-www-form
```

The response should be something similar to what follows:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDQ2NTE3MDQsInVzZXJfbmFtZSI6
```

Notice the dots between the token. These dots separate the header, the payload, and the signature of the JWT. Everything is base 64 encoded, so if you extract the payload from the previous token and decode it you can perfectly read the content (it's not encrypted, but just signed). If you want to go deep on JWT, JWS, and JWE concepts, I recommend you to read a great article written by Prabath Siriwardena, which is available at https://medium.facilelogin.com/jwt-jws-and-jwe-for-not-so-dummies-b63310d201a3.

In addition, you might have faced some delay when running the previous command. On my machine it took about five seconds to return the access token. This happens because of the initialization process of the `MacSigner`, which relies on the `javax.crypto.Mac` class that is an expensive class to be instantiated. But don't worry because this delay happens just once when you first sign an access token.

See also

The article by Prabath Siriwardena, JWT, JWS, and JWE for Not So Dummies is available at https://medium.facilelogin.com/jwt-jws-and-jwe-for-not-so-dummies-b63310d201a3.

Validating JWT tokens at the Resource Server side

In the previous recipe, you learned how to generate JWT access tokens at the Authorization Server side. Now it's time to know how to validate any given JWT access token that is symmetrically signed at the Resource Server side. When the client tries to access any OAuth 2.0 protected resource with a JWT, the Resource Server has to use the same signing key used by an Authorization Server to sign the payload, to verify if the content was not changed between requests by any malicious client or user.

Getting ready

To run this recipe, you will create a Spring Boot application with Java 8, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies as `Web` and `Security` (that will declare properly all the Spring Boot starters needed for this recipe).

How to do it...

This recipe creates the project `resource-server-jwt` which is available at GitHub in `Chapter05` folder. Import the generated project as a Maven project into your IDE and follow the following steps:

1. Open `pom.xml` file and add the following extra dependencies for Spring Security OAuth2 and Spring Security JWT in the same way we did for Authorization Server project:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$-->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
```

2. Add the following content to `application.properties` file:

```
server.port=8081
security.oauth2.resource.jwt.key-value=non-prod-signature
```

3. Declare the `OAuth2ResourceServer` class within the `com.packt.example.resourceserverjwt.oauth` package with the following content:

```
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends
    ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/api/**");
    }
}
```

4. That's it. To start validating JWS tokens when using Spring Security OAuth2 with Spring Boot, everything will be configured automatically

just by adding the `security.oauth2.resource.jwt.key-value` property within the `application.properties` file.

5. Create the following class within the `api` sub-package to provide an API to allow clients to interact:

```
@Controller
public class UserController {
    @RequestMapping("/api/profile")
    public ResponseEntity<UserProfile> myProfile() {
        String username = (String) SecurityContextHolder.getContext()
                .getAuthentication().getPrincipal();
        String email = username + "@mailinator.com";
        UserProfile profile = new UserProfile(username, email);
        return ResponseEntity.ok(profile);
    }
    public static class UserProfile {
        private String name;
        private String email;
        public UserProfile(String name, String email) {
            this.name = name;
            this.email = email;
        }
        // getters omitted
    }
}
```

6. Optionally, if you aren't using Spring Boot in your current project you could start implementing the `JwtAccessTokenConverterConfigurer` interface as presented in the following code (note that the `HttpSecurity` configuration was hidden just for didactical purpose, so to the following source code to work you need to configure `HttpSecurity`):

```
@Configuration @EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
    @Override
    public void configure(JwtAccessTokenConverter converter) {
        converter.setVerifier(verifier());
        converter.setSigningKey("non-prod-signature");
    }
    @Bean
    public SignatureVerifier verifier() {
        return new MacSigner("non-prod-signature");
    }
}
```

How it works...

According to the configurations defined for the current Resource Server, when the client sends a request using a JWT access token, the Resource Server validates the signature sent as a component of the token by itself using the same symmetric key applied by the Authorization Server when creating the access token. After validating the signature, the Resource Server is able to extract data from the payload and use any present information such as data about the Resource Owner or even about the client. RFC 7519 specifies registered claims, which defines attributes that identify the issuer, the subject, the expiration time, and the audience of the token. Spring Security JWT generates the payload using different names such as `user_name` for subject entry.

Start both the applications `resource-server-jwt` and `auth-server-jwt` created in the *Generating access tokens as JWT* recipe. After starting both applications, send a request to the Authorization Server to generate an access token using the following command:

```
curl -X POST --user clientapp:123456 -H "Content-Type: application/x-www-forn
```

Copy the access token returned as the result of running the previous command and go to https://www.base64decode.org/ to decode the payload to see what you have received from the Authorization Server. To decode the payload, extract just the second part of the access token and after submitting the content at the `base64decode` website you should see something similar to what follows:

```
{
 "exp": 1504694916,
 "user_name": "adolfo",
 "authorities": ["ROLE_USER"],
 "jti": "87913ebc-6c13-4b94-86d9-2b310f742fb2",
 "client_id": "clientapp",
 "scope": ["read_profile"]
}
```

And to check if the Resource Server is appropriately validating the access token try to access the user's profile endpoint using the JWT retrieved earlier as follows (the access token presented in the following code was shortened just to have a clear example):

```
curl -X GET -H "Authorization: Bearer eyJhbGzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiC
```

There's more...

When using JWT with Spring Security OAuth2, you might face a special case where you need to convert the authenticated user data to an implementation of `UserDetails` which could be a sophisticated structure to represent the Resource Owner who granted the access token being generated. To allow for a detailed `UserDetails` conversion, you could declare a `UserAuthenticationConverter` and set up this converter to `DefaultAccessTokenConverter` as presented in the following configuration class:

```
@Configuration
public class AccessTokenConverterConfiguration {
    @Autowired
    private UserDetailsService userDetailsService;
    @Bean
    public DefaultAccessTokenConverter defaultAccessTokenConverter() {
        DefaultAccessTokenConverter tokenConverter = new DefaultAccessTokenCc
        tokenConverter.setUserTokenConverter(userAuthenticationConverter());
        return tokenConverter;
    }
    @Bean
    public UserAuthenticationConverter userAuthenticationConverter() {
        DefaultUserAuthenticationConverter converter
            = new DefaultUserAuthenticationConverter();
        converter.setUserDetailsService(userDetailsService);
        return converter;
    }
}
```

See also

- Generating access tokens as JWT

Adding custom claims on JWT

Sometimes we need to handle some specific scenarios that deviate from generic applications. As an example, the Resource Server might need more data about the Resource Owner and to avoid round-trips between the Resource Server and the Authorization Server, a self contained access token can carry more information within the payload. This recipe will help you to handle exactly this kind of situation.

Getting ready

This recipe will be created as a Spring Boot application with Java 8, H2 database, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and reference `Web` , `JPA`, `H2`, and `Security` as dependencies (that will declare properly all the Spring Boot starters needed for this recipe). As explained earlier in the previous chapters you also need to define the Artifact and Group name.

How to do it...

This recipe creates the `custom-claims-jwt` project, which is available on GitHub in the `Chapter05` folder. Import the generated source code as a Maven project into your IDE and follow the following steps:

1. Open the `pom.xml` file and add the following extra dependencies for Spring Security OAuth2 and Spring Security JWT in the same way we did for the Authorization Server project:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$-->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
```

2. As we are going to use an in-memory database to simplify the examples, add the following content to the `application.properties` file to configure the `h2` database:

```
spring.datasource.url=jdbc:h2:mem:ironbank;DB_CLOSE_DELAY=-1;DB_CLOSE_
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialec
spring.jpa.properties.hibernate.hbm2ddl.auto=create
```

3. Create a package to hold the security configuration classes so we can better handle Resource Owner authentication and data extraction (this package can be simply named as `security`).
4. Add the following entity to represent the Resource Owner (getters and setters omitted):

```
@Entity
class ResourceOwner {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String email;
}
```

5. Create an implementation of the `UserDetails` interface as presented in the following code. Declare all the Boolean methods to return true for simplicity, and you can also return an empty list of `GrantedAuthority` entries from the `getAuthorities` method:

```
public class ResourceOwnerUserDetails implements UserDetails {
    private static final long serialVersionUID = 1L;
    private ResourceOwner wrapped;
    public ResourceOwnerUserDetails(ResourceOwner wrapped) {
        this.wrapped = wrapped;
    }
    public String getEmail() { return wrapped.getEmail(); }
    public String getPassword() { return wrapped.getPassword(); }
    public String getUsername() { return wrapped.getUsername(); }
    // other methods hidden for clarity
}
```

6. Create the following repository so we can retrieve the Resource Owner's information:

```
public interface ResourceOwnerRepository extends
    CrudRepository<ResourceOwner, Long> {
    Optional<ResourceOwner> findByUsername(String username);
}
```

7. Then implement the `UserDetailsService` interface as follows:

```
@Service
public class ResourceOwnerDetailsService
    implements UserDetailsService {
    @Autowired
    private ResourceOwnerRepository repo;
    public UserDetails loadUserByUsername(String uname)
        throws UsernameNotFoundException {
        ResourceOwner user = repo.findByUsername(uname)
           .orElseThrow(() -> new RuntimeException());
        return new ResourceOwnerUserDetails(user);
    }
}
```

8. Now create a new package to hold OAuth 2.0 related classes (the package's name could be `oauth`). Then add the following class that is in charge of adding custom claims to the JWT access token:

```
@Component
public class AdditionalClaimsTokenEnhancer
    implements TokenEnhancer {
    @Override
    public OAuth2AccessToken enhance(
        OAuth2AccessToken accessToken,
        OAuth2Authentication authentication) {
        Map<String, Object> additional = new HashMap<>();
        ResourceOwnerUserDetails user =
            (ResourceOwnerUserDetails) authentication
               .getPrincipal();
        additional.put("email", user.getEmail());
        DefaultOAuth2AccessToken token = (DefaultOAuth2AccessToken) ac
        token.setAdditionalInformation(additional);
        return accessToken;
    }
}
```

9. Create the Authorization Server configuration as follows:

```
@Configuration @EnableAuthorizationServer
public class OAuth2AuthorizationServerConfiguration
    extends AuthorizationServerConfigurerAdapter {
    @Autowired
    private AuthenticationManager authenticationManager;
    @Autowired
    private AdditionalClaimsTokenEnhancer enhancer;
    public void configure(ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory()
           .withClient("clientapp").secret("123456")
           .scopes("read_profile")
           .authorizedGrantTypes(
              "password", "authorization_code");
    }
}
```

10. Add the following bean declarations within the `OAuth2AuthorizationServerConfiguration` class:

```
@Bean
public JwtAccessTokenConverter accessTokenConverter() {
   JwtAccessTokenConverter conv = new JwtAccessTokenConverter();
   conv.setSigningKey("non-prod-signature");
   return conv;
}
@Bean
public JwtTokenStore jwtTokenStore() {
    return new JwtTokenStore(accessTokenConverter());
}
```

11. Add the following configuration inside the `OAuth2AuthorizationServerConfiguration` class:

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints
    TokenEnhancerChain chain = new TokenEnhancerChain();
    chain.setTokenEnhancers(
       Arrays.asList(enhancer, accessTokenConverter()));
    endpoints
       .authenticationManager(authenticationManager)
       .tokenStore(jwtTokenStore())
       .tokenEnhancer(chain)
       .accessTokenConverter(accessTokenConverter());
}
```

12. Create the file `data.sql` within resources directory with the following content to create an user for our application:

```
insert into resource_owner (username, password, email)
values ('adolfo', '123', 'adolfo@mailinator.com');
```

How it works...

The key strategy provided by Spring Security OAuth2 to add custom claims to an issued JWT access token, is the interface `TokenEnhancer` that was implemented by the `AdditionalClaimsTokenEnhancer` class. Because of this class we can intercept the token creation step and change whatever we need on the `OAuth2AccessToken` instance.

Regarding this recipe, as the purpose is to add custom claims, we did it by adding the user's email attribute into the additional information data structure. To check the results of this token enhancement, start the application and try to request for a new access token by using Authorization Code grant type or Resource Owner Password Credentials grant type. After receiving an access token, extract the payload component of the JWT (considering the following access token we have to extract the highlighted portion of the token):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDQ3MzgyMDMsInVzZXJfbmFtZSI6
```

After Base 64 decodes it, you will be able to notice that now the access token is carrying the user's email:

```
{
 "exp": 1504738203,
 "user_name": "adolfo",
 "jti": "2fbc5928-41a6-42dc-b88d-bebda2710de8",
 "email": "adolfo@mailinator.com",
 "client_id": "clientapp",
 "scope": [
 "read_profile"
 ]
}
```

See also

- Generating access tokens as JWT

Asymmetric signing of a JWT token

In the previous recipes, we were symmetrically signing the access token. That is, we were using the same key to sign the payload at the Authorization Server and to validate it on the Resource Server. This recipe presents you with another approach for signing JWT using asymmetric keys, where the Authorization Server uses a private key to sign the JWT payload and the Resource Server uses a public key to validate it.

Getting ready

To run this recipe, you will need to create a Spring Boot project for the Authorization Server using Java 8, Maven, Spring Web, and Spring Security. Some dependencies will be presented in the *How to do it*... section.

How to do it...

This recipe shows you how to create the Authorization Server that will be defined within the `jwt-asymmetric-server` project. This project is available on GitHub in the `Chapter05` folder and all you will need to create this project is presented in the next steps:

1. You can create the `jwt-asymmetric-server` project at Spring Initializr referencing `Web` and `Security` as the main dependencies to easily set up the project.
2. If you have created the project using Spring Initializr, import the generated source code as a Maven project into your IDE.
3. Add the following dependencies into the `pom.xml` that will bring Spring Security OAuth2 and Spring Security JWT:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$-->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
```

4. Add the following content to the `application.properties` file to configure the Resource Owner credentials:

```
security.user.name=adolfo
security.user.password=123
```

5. Now let's create the most important class to configure the Authorization Server with JWT support using an asymmetric key to sign the JWT as well. Create the `OAuth2AuthorizationServer` class as follows within the sub-package named `oauth`:

```
@Configuration @EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConf
    @Autowired
    private AuthenticationManager authenticationManager;
```

```
        @Override
        public void configure(AuthorizationServerEndpointsConfigurer endpo
           endpoints.authenticationManager(authenticationManager);
        }

        @Override
        public void configure(ClientDetailsServiceConfigurer clients) thro
            clients.inMemory()
                .withClient("clientapp").secret("123456")
                .scopes("read_profile")
                .authorizedGrantTypes(
                    "password", "authorization_code");
        }
    }
```

6. As we did for the Authorization Server that adds support for JWT with symmetric keys, we also need to create a `JwtTokenStore` and `JwtAccessTokenConverter`. So, add the following bean definition within the `OAuth2AuthorizationServer` class (notice that the keys are saved in memory because we aren't using a `KeyStore` pointing to some external file, as for example a JKS file which stands for *Java KeyStore*):

```
    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        try {
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
            SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
            keyGen.initialize(1024, random);
            KeyPair keyPair = keyGen.generateKeyPair();
            converter.setKeyPair(keyPair);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return converter;
    }
```

7. Now define the following `JwtTokenStore` bean using the `JwtAccessTokenConverter` declared in the previous step:

```
    @Bean
    public JwtTokenStore jwtTokenStore() {
        return new JwtTokenStore(accessTokenConverter());
    }
```

8. And finally replace the `AuthorizationServerEndpointsConfigurer` configuration method to what follows:

```
    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints
    throws Exception {
    endpoints.authenticationManager(authenticationManager)
        .tokenStore(jwtTokenStore())
        .accessTokenConverter(accessTokenConverter());
}
```

9. Set up the `AuthorizationServerSecurityConfigurer` object adding the following method to the `OAuth2AuthorizationServer` class:

```
        @Override
        public void configure(AuthorizationServerSecurityConfigurer security)
            throws Exception {
            security.tokenKeyAccess("permitAll()");
        }
```

10. Now our Authorization Server is ready to run on port `8080` to start issuing JWT access tokens.

How it works...

There are subtle differences on how we add support for asymmetric keys from how we configure JWT with a symmetric key. The main difference now is that we are setting up a `KeyPair` that holds the private and public key that have to be used to sign and to validate an access token, respectively. We are using the RSA algorithm which relies on private and public keys to perform cryptographic functions.

Another interesting difference is that the Authorization Server being created is providing an endpoint to retrieve the public key, which is required by the Resource Server to validate oncoming JWT access tokens. The default endpoint provided by Spring Security OAuth2 is `/oauth/token_key`.

To check how the Authorization Server issues access tokens, and how the `token_key` endpoint can be used, start the application by running `mvn spring-boot:run` on the command line and try to generate a new access token by using the authorization code or Resource Owner Password Credentials grant types. On my environment, requesting an access token gave me the following result (note that I am presenting you with an invalid access token in the following JSON result just for brevity, so generate an access token on your own machine to catch the correct results):

```
{
 "access_token": "eyJhbGciOzXVCJ9.eyJleHAiOGhvOiI1Z.XmzNfaUrkOfwX0Uym8M",
 "token_type": "bearer",
 "expires_in": 43199,
 "scope": "read_profile",
 "jti": "5e928e52-585c-4b97-b1dd-839889828a00"
}
```

The `/oauth/token_key` endpoint can be accessed by anyone without restrictions just by running the following command:

```
curl http://localhost:8080/oauth/token_key
```

Depending on your needs you might want to properly protect this endpoint with authentication and roles, but that would not be an issue in most cases as it's needed that the private key compromises the integrity of an access token. By running the previous command I have received the following result:

```
{
 "alg": "SHA256withRSA",
 "value": "-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQ
}
```

See also

- Generating access tokens as JWT

Validating asymmetric signed JWT token

In the previous recipe, you learned how to generate JWT access tokens at the Authorization Server side using asymmetric keys. Now it's time to know how to validate any given JWT access token asymmetrically signed at the Resource Server side. Now instead of statically setting up the key to validate the access token, the Resource Server will retrieve the public key through the `/oauth2/token_key` endpoint provided by the Authorization Server. It gives flexibility to the OAuth Provider and helps with maintainability.

Getting ready

To run this recipe, you will need to create a Spring Boot project for the Resource Server using Java 8, Maven, Spring Web, and Spring Security. Some dependencies will be presented in the *How to do it...* section.

How to do it...

This recipe shows you how to create the Resource Server which will be defined as the project `jwt-asymmetric-resource`. This project is available on GitHub in the `Chapter05` folder and all that you need to create this project is presented in the next steps:

1. You can create the project `jwt-asymmetric-resource` at Spring Initializr, referencing `Web` and `Security` as the main dependencies to easily set up the project.
2. If you have created the project using Spring Initializr, import the generated source code as a Maven project into your IDE.
3. Add the following dependencies into `pom.xml`, which will bring Spring Security OAuth2 and Spring Security JWT:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$-->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
```

4. Add the following content to the `application.properties` file to configure the Resource Owner credentials:

```
server.port=8081
security.user.name=adolfo
security.user.password=123
security.oauth2.resource.jwt.key-uri=http://localhost:8080/oauth/toker
```

5. Create the API that provides the user's profile through the following source code. Create the `UserController` class within the sub-package `api`:

```
@Controller
public class UserController {
    @RequestMapping("/api/profile")
    public ResponseEntity<UserProfile> myProfile() {
        String username = (String) SecurityContextHolder.getContext()
                .getAuthentication().getPrincipal();
```

```
            String email = username + "@mailinator.com";
            UserProfile profile = new UserProfile(username, email);
            return ResponseEntity.ok(profile);
        }
        public static class UserProfile {
            private String name;
            private String email;
            // constructor, getters and setters omitted for brevity
        }
    }
```

6. Then create the most important class for this recipe which is the `OAuth2ResourceServer`, to declare the Resource Server configuration:

```
    @Configuration @EnableResourceServer
    public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
        @Override
        public void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                .anyRequest().authenticated().and()
                .requestMatchers().antMatchers("/api/**");
        }
    }
```

How it works...

Basically the Resource Server configured at this recipe works in the same way as the Resource Server configured for *Validating JWT Tokens at the Resource Server side* recipe. Internally the main difference is defined by how the validation process runs because of the usage of asymmetric keys.

In addition, as the Resource Server was configured to use the `/oauth/token_key` endpoint to retrieve the public key, an instance of `JwtAccessTokenConverter` is created and has the signing key defined at the moment of its creation by the `ResourceServerTokenServicesConfiguration` internal class. The public key is retrieved by sending a request using `RestTemplate`, so if you do not use Spring Boot with the config property `security.oauth2.resource.jwt.key-uri` you will have to do it by yourself.

Start both the Authorization Server and the Resource Server, request an access token, and try to access the user's profile API with the asymmetrically signed JWT.

See also

- Generating access tokens as JWT
- Validating JWT tokens at the Resource Server side
- Asymmetric signing a JWT token

Using JWE to cryptographically protect JWT tokens

This recipe will present an advanced topic related to JWT architecture, which is **JSON Web Encryption** (**JWE**). As you have seen before in the previous recipes of this chapter, we were using the **JSON Web Signature** (**JWS**) approach, which promotes integrity protection. With JWE, we start providing confidentiality to JWT tokens issued by the Authorization Server. This recipe is important to learn so you can add another layer of security for your application.

Getting ready

To run this recipe, you will need to create a Spring Boot application to configure the Authorization Server using Java 8, MySQL, Maven, and Nimbus (which provides JWE and encryption capabilities). The other dependencies will be described in the *How to do it...* section.

How to do it...

This recipe shows you how to create the Authorization Server that will be defined as the `jwe-server` project. This project is available on GitHub in the `Chapter05` folder and all that you will need to create this project is presented in the next steps:

1. You can create the `jwe-server` project at Spring Initializr, referencing `Web` and `Security` dependencies to easily set up the project.

2. If you have created the project using Spring Initializr, import the generated source code as a Maven project into your IDE and add the following dependencies into the `pom.xml`:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$ -->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
<dependency>
  <groupId>com.nimbusds</groupId>
  <artifactId>nimbus-jose-jwt</artifactId>
  <version>4.23</version>
</dependency>
```

3. Add the following content to the `application.properties` file to configure the Resource Owner credentials:

```
security.user.name=adolfo
security.user.password=123
```

4. Before setting up the Authorization Server, let's create some additional classes that will customize some default behaviors from Spring Security OAuth2 so our server can create JWE tokens (that is, basically JWT with encrypted payload). So, at first create the sub-package `oauth.jwt`.
5. Create the `JweTokenSerializer` class within the `oauth.jwt` sub-package, as follows:

```
public class JweTokenSerializer {
    private String encodedKeypair;
    public JweTokenSerializer(String encodedKeypair) {
        this.encodedKeypair = encodedKeypair;
    }
}
```

6. Add the following method to `JweTokenSerializer` that will be in charge to encrypt any given payload:

```
public String encode(String payload) {
    try {
        byte[] decodedKey = Base64.getDecoder().decode(encodedKeypair)
        SecretKey key = new SecretKeySpec(decodedKey, 0, decodedKey.le

        JWEHeader header = new JWEHeader(JWEAlgorithm.DIR, EncryptionM
        Payload payloadObject = new Payload(payload);

        JWEObject jweObject = new JWEObject(header, payloadObject);
        jweObject.encrypt(new DirectEncrypter(key));

        return jweObject.serialize();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

7. Add the following method within `JweTokenSerializer` to decrypt any payload:

```
public Map<String, Object> decode(String base64EncodedKey, String cont
    byte[] decodedKey = Base64.getDecoder().decode(base64EncodedKey);
    SecretKey key = new SecretKeySpec(decodedKey, 0, decodedKey.length
    try {
        JWEObject  jweObject = JWEObject.parse(content);
        jweObject.decrypt(new DirectDecrypter(key));

        Payload payload = jweObject.getPayload();
        ObjectMapper objectMapper = new ObjectMapper();
        ObjectReader reader = objectMapper.readerFor(Map.class);
        return reader.with(DeserializationFeature.USE_LONG_FOR_INTS)
                .readValue(payload.toString());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

8. Now create the `JweTokenEnhancer` as follows. This class works as a custom version for a default token enhancer used by Spring Security OAuth2 (which is the `JwtAccessTokenConverter` class) to generate JWT access tokens (import both classes `JsonParser` and `JsonParserFactory` from

`org.springframework.security.oauth2.common.util` package):

```
public class JweTokenEnhancer implements TokenEnhancer {
    public static final String TOKEN_ID = AccessTokenConverter.JTI;
    private JsonParser objectMapper = JsonParserFactory.create();
    private AccessTokenConverter tokenConverter;
    private JweTokenSerializer tokenSerializer;

    public JweTokenEnhancer(AccessTokenConverter tokenConverter,
        JweTokenSerializer tokenSerializer) {
        this.tokenConverter = tokenConverter;
        this.tokenSerializer = tokenSerializer;
    }

    @Override
    public OAuth2AccessToken enhance(OAuth2AccessToken accessToken,
        OAuth2Authentication authentication) {
        DefaultOAuth2AccessToken result = new DefaultOAuth2AccessToker
        Map<String, Object> info = new LinkedHashMap<>(accessToken.get
        String tokenId = result.getValue();
        if (!info.containsKey(TOKEN_ID)) {
            info.put(TOKEN_ID, tokenId);
        }
        result.setAdditionalInformation(info);
        result.setValue(encode(result, authentication));
        return result;
    }

    private String encode(DefaultOAuth2AccessToken accessToken,
        OAuth2Authentication authentication) {
        String content;
        try {
            content = objectMapper.formatMap(
                tokenConverter.convertAccessToken(accessToken, authent
            return tokenSerializer.encode(content);
        } catch (Exception e) {
            throw new IllegalStateException("Cannot convert access tok
        }
    }
}
```

9. Now create the `OAuth2AuthorizationServer` class as follows:

```
@Configuration @EnableAuthorizationServer
public class OAuth2AuthorizationServer
    extends AuthorizationServerConfigurerAdapter {
    @Autowired
    private AuthenticationManager authenticationManager;
    @Override
    public void configure(AuthorizationServerSecurityConfigurer securi
        throws Exception {
        security.tokenKeyAccess("permitAll()");
    }
    @Override
    public void configure(ClientDetailsServiceConfigurer clients) thro
```

```
            clients
                .inMemory().withClient("clientapp").secret("123456")
                .scopes("read_profile").authorizedGrantTypes(
                    "password", "authorization_code");
        }
    }
```

10. Declare the following String bean inside the `OAuth2AuthorizationServer` class that is the base 64 encoded version of the symmetric key generated through the usage of the **Advanced Encryption Standard *(*AES)** algorithm, which will be used for encryption and decryption of JWE:

```
    @Bean
    public String symmetricKey() {
        try {
            KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
            keyGenerator.init(128);
            SecretKey key = keyGenerator.generateKey();
            return Base64.getEncoder().encodeToString(key.getEncoded());
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
```

11. Declare the following beans within `OAuth2AuthorizationServer`:

```
    @Bean
    public TokenEnhancer tokenEnhancer() {
        return new JweTokenEnhancer(accessTokenConverter(),
            new JweTokenSerializer(symmetricKey()));
    }

    @Bean
    public JwtTokenStore jwtTokenStore() {
        return new JwtTokenStore(accessTokenConverter());
    }

    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        JwtAccessTokenConverter tokenConverter = new JwtAccessTokenConvert
        tokenConverter.setSigningKey(symmetricKey());
        return tokenConverter;
    }
```

12. Finally, set up the `AuthorizationServerEndpointsConfigurer` as follows:

```
    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints
        endpoints.authenticationManager(authenticationManager)
            .tokenStore(jwtTokenStore())
            .tokenEnhancer(tokenEnhancer())
            .accessTokenConverter(accessTokenConverter());
        }
```

13. Now the `jwe-server` is ready to be executed through the `mvn spring-boot:run` command.

How it works...

The key points for this recipe to work as desired, that is, to allow the Authorization Server to issue JWE tokens, was to create a custom version of `TokenEnhancer` and the helper class `JweTokenSerializer`, which is responsible for encryption and decryption of data. By implementing our custom `TokenEnhancer`, Spring Security OAuth2 allows you to set additional capabilities or the format of an access token being issued. It was perfect as an extension point for adding support for JWE.

An important thing to be mentioned is that, because we are configuring symmetric keys, the endpoint `oauth/token_key` is not publicly available as it is when we configure an asymmetric key. That is because at the moment, we just have one key to sign and to validate the content of the payload. On the other hand, when using an asymmetric key we have the private and public key where as the name implies, the public key can be shared with everybody without exposing the security of the application.

Start the application and try to generate an access token to see all the results, which should be something similar to what follows:

```
{
  "access_token":   "eyJlbmMiOiJBMTI4R0NNIiwiYWxnIjoiZGlyIn0..6LyaLUPq3DQREzx
  "token_type": "bearer",
  "expires_in": 43194,
  "scope": "read_profile",
  "jti": "d649f685-449a-421d-8d68-ddb5217a1905"
}
```

See also

- Generating access tokens as JWT

Using JWE at the Resource Server side

This recipe will show you how to create a Resource Server that is able to validate JWE access tokens. By using this recipe you will also be able to understand important extension points provided by Spring Security OAuth2 and how to integrate other libraries such as Nimbus to allow cryptography.

Getting ready

To run this recipe, you will need to create a Spring Boot application to configure the Resource Server using Java 8, Maven, Spring Web, Spring Security, and Nimbus (which provides JWE and encryption capabilities). Some dependencies will be described in the *How to do it...* section.

How to do it...

This recipe presents you with how you can create the Resource Server that will be defined as the project `jwe-resource`. This project is available on GitHub in the `Chapter05` folder and all you need to create this project is presented in the following steps:

1. You can create the project `jwe-resource` at Spring Initializr, referencing `Web` and `Security` dependencies to easily set up the project.
2. If you have created the project using Spring Initializr, import the generated source code as a Maven project into your IDE and add the following dependencies into the `pom.xml`:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$ -->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
<dependency>
  <groupId>com.nimbusds</groupId>
  <artifactId>nimbus-jose-jwt</artifactId>
  <version>4.23</version>
</dependency>
```

3. Add the following content to the `application.properties` file to configure the Resource Owner credentials:

```
server.port=8081

security.user.name=adolfo
security.user.password=123

security.oauth2.resource.jwt.key-uri=http://localhost:8080/oauth/toker
security.oauth2.client.client-id=clientapp
security.oauth2.client.client-secret=123456
```

4. Create the class `UserController` that will provide the user's profile API (create this class within the sub-package `api`). Don't forget to create the constructor and getters and setters for respective attributes for the

`UserProfile` class:

```
@Controller
public class UserController {

    @RequestMapping("/api/profile")
    public ResponseEntity<UserProfile> myProfile() {
        String username = (String) SecurityContextHolder.getContext()
                .getAuthentication().getPrincipal();
        String email = username + "@mailinator.com";
        UserProfile profile = new UserProfile(username, email);
        return ResponseEntity.ok(profile);
    }

    public static class UserProfile {
        private String name;
        private String email;
        // controller, getters and setters hidden for brevity
    }
}
```

5. Create the sub-package `oauth/jwt` within the main package of the project and create the `JweTokenEnhancer` and `JweTokenSerializer` classes with the same content from the *Using JWE to cryptographically protect JWT tokens* recipe. You could also copy these classes directly or even create a separate library if you are developing something for the production environment.

6. Inside the `oauth/jwt` sub-package, create the `JweTokenStore` class as follows:

```
public class JweTokenStore implements TokenStore {
    private String encodedSigningKey;
    private final TokenStore delegate;
    private final JwtAccessTokenConverter converter;
    private final JweTokenSerializer crypto;

    public JweTokenStore(String encodedSigningKey, TokenStore delegate
                         JwtAccessTokenConverter converter, JweTokenSe
        this.encodedSigningKey = encodedSigningKey;
        this.delegate = delegate;
        this.converter = converter;
        this.crypto = crypto;
    }

    @Override
    public OAuth2AccessToken readAccessToken(String tokenValue) {
        return converter.extractAccessToken(
            tokenValue, crypto.decode(encodedSigningKey, tokenValue));
    }
```

```
        @Override
        public OAuth2Authentication readAuthentication(OAuth2AccessToken t
            return readAuthentication(token.getValue());
        }

        @Override
        public OAuth2Authentication readAuthentication(String token) {
            return converter.extractAuthentication(crypto.decode(encodedSi
        }
    }
```

7. Not every method declared by the `TokenStore` interface were implemented in the previous step. Implement all the remaining methods to delegate the execution to the *decorated* version of `TokenStore`, which is being referenced as `delegate`. The following method should be used as an example so you can understand what to do for the remaining methods:

```
    @Override
    public void removeAccessToken(OAuth2AccessToken token) {
        delegate.removeAccessToken(token);
    }
```

8. Create the configuration for the Resource Server through the `OAuth2ResourceServer` class as follows:

```
    @Configuration @EnableResourceServer
    public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
        @Autowired
        private ResourceServerProperties resource;

        @Override
        public void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                .anyRequest().authenticated().and()
                .requestMatchers().antMatchers("/api/**");
        }
    }
```

9. Add the following beans declarations within the `OAuth2ResourceServer` class:

```
    @Bean
    public JweTokenStore tokenStore() {
        return new JweTokenStore(getSignKey(),
            new JwtTokenStore(jwtTokenConverter()), jwtTokenConverter(), t
    }

    @Bean
    public JweTokenSerializer tokenSerializer() {
            return new JweTokenSerializer(getSignKey());
        }

        @Bean
        public JwtAccessTokenConverter jwtTokenConverter() {
            JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
            converter.setSigningKey(getSignKey());
            return converter;
        }

        @Bean
        public String getSignKey() {
            RestTemplate keyUriRestTemplate = new RestTemplate();
            HttpHeaders headers = new HttpHeaders();
            String username = this.resource.getClientId();
            String password = this.resource.getClientSecret();
            if (username != null && password != null) {
                byte[] token = Base64.getEncoder().encode((username + ":" + pa
                headers.add("Authorization", "Basic " + new String(token));
            }
            HttpEntity<Void> request = new HttpEntity<>(headers);
            String url = this.resource.getJwt().getKeyUri();
            return (String) keyUriRestTemplate
                    .exchange(url, HttpMethod.GET, request, Map.class).getBody
                    .get("value");
        }
```

10. Then add the following configuration within the `OAuth2ResourceServer` class:

```
        @Override
        public void configure(ResourceServerSecurityConfigurer resources) thro
            resources.tokenStore(tokenStore());
        }
```

11. Now the Resource Server is ready to run and start serving the user's profile API.

How it works...

The Resource Server configured for this recipe is a little bit more complex than the Resource Server configurations created in the *Validating asymmetrically signed JWT token* and *Validating JWT tokens at the Resource Server side* recipes. That is happening because at this time, we cannot rely just on the `oauth/token_key` endpoint configuration property to start the auto-configuration process provided by Spring Boot. As we had to manually retrieve the public key, we had to declare some additional beans as `JwtAccessTokenConverter`, `JweTokenSerializer`, and `JweTokenStore`.

Check that everything is working fine by creating a new access token by running both the Authorization Server and Resource Server that supports JWE tokens. After creating an access token try to access the user's profile endpoint using the newly JWE token.

See also

- Using JWE to cryptographically protect JWT tokens

Using proof-of-possession key semantics on OAuth 2.0 Provider

This chapter will present you with the means to implement an OAuth 2.0 solution where it is possible for the client to prove to the Resource Server that it is in possession of a given key through the usage of JWT tokens. Enabling this feature, increases the safeness of an API because it does not allow the usage of an access token that does not belong to the client sending a request. It can be implemented in many different ways as per the official specification described by RFC 7800 that is available at https://tools.ietf.org/html/rfc7800. Nevertheless, this recipe presents you with how to implement proof-of-possession key semantics using asymmetric keys.

Getting ready

Throughout this recipe we will create the OAuth 2.0 Provider that is composed by the Resource Server and the Authorization Server. This time both Authorization Server and Resource Server is implemented as a unique Spring Boot application using Java 8, Maven, Spring Web, Spring Security, and Nimbus (which provides JWE and encryption capabilities). The other dependencies will be described in the *How to do it...* section.

How to do it...

The next steps will show you how to create the OAuth 2.0 Provider, which was named `pop-server`. This project is also available on GitHub in the `Chapter05` folder:

1. First of all, create the `pop-server` project as a Spring Boot application (I recommend the usage of Spring Initializr). At Spring Initializr add `Web`, `Thymeleaf` and `Security` as dependencies.
2. Import the `pop-server` project to your IDE as a Maven project and add the following dependencies into the `pom.xml` file:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$ -->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
<dependency>
  <groupId>com.nimbusds</groupId>
  <artifactId>nimbus-jose-jwt</artifactId>
  <version>4.23</version>
</dependency>
```

3. Add the following content into the `application.properties` file (the `key-value` property is being defined for the Resource Server to validate JWT tokens):

```
security.user.name=adolfo
security.user.password=123
security.oauth2.resource.jwt.key-value=non-prod
```

4. Create one sub-package named `oauth` with two other inner packages named `authorizationserver` and `resourceserver` respectively.

5. As the Authorization Server will return a JWT token in response to the token request and the application needs to support proof-of-possession key semantics, create the following implementation of `TokenEnhancer`

interface. The `PoPTokenEnhancer` class is in charge to extract the public key sent by clients during the token request phase. And finally, this public key is used as an additional information within the JWT token (create the following class within the `oauth.authorizationserver` sub-package):

```
class PoPTokenEnhancer implements TokenEnhancer {
    @Override
    public OAuth2AccessToken enhance(
        OAuth2AccessToken accessToken,
        OAuth2Authentication authentication) {
        Map<String, Object> additional = new HashMap<>();
        String publicKey = authentication.getOAuth2Request()
            .getRequestParameters().get("public_key");
        additional.put("public_key", publicKey);

        DefaultOAuth2AccessToken defaultAccessToken = (DefaultOAuth2Ac
        defaultAccessToken.setAdditionalInformation(additional);
        return accessToken;
    }
}
```

6. It's not a good idea to return a claim within the JWT token structure and within the JSON returned in response to a token request. That's just to avoid data to be duplicated, and to do so, create the following enhancer within the same package as `PoPTokenEnhancer`:

```
public class CleanTokenEnhancer implements TokenEnhancer {
    @Override
    public OAuth2AccessToken enhance(OAuth2AccessToken accessToken,
        OAuth2Authentication authentication) {
        accessToken.getAdditionalInformation().remove("public_key");
        return accessToken;
    }
}
```

7. Now create the Authorization Server configuration through the `OAuth2AuthorizationServer` class whose source code is presented as follows:

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConf

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) thro
        clients
            .inMemory()
            .withClient("clientapp").secret("123456")
            .scopes("read_profile")
            .authorizedGrantTypes("authorization_code");
        }

    }
```

8. Then add the following methods into the `OAuth2AuthorizationServer` class:

```
    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        converter.setSigningKey("non-prod");
        return converter;
    }
    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints
        TokenEnhancerChain chain = new TokenEnhancerChain();
        chain.setTokenEnhancers(Arrays.asList(
           new PoPTokenEnhancer(), accessTokenConverter(), new CleanTokenE
        endpoints.tokenEnhancer(chain);
    }
```

9. The Authorization Server is ready to generate a JWT token with the custom claim named public key that needs to be present within the access token so that the client can send it to access OAuth's protected resources through the usage of proof-of-possession key (the Resource Server will need the public key to validate the nonce field that will be sent by the client).
10. Now it's time to create the Resource Server's configurations, but first of all let's create an `Authentication` structure that contains the `nonce` attribute. This attribute will be used to prove that the client is in possession of the private key bound to the public key used to generate the nonce as well. Create the following class within the `oauth.resourceserver` sub-package (note that it's a decorator for an existent `Authentication` class):

```
    public class PoPAuthenticationToken implements Authentication {
        private Authentication authentication;
        private String nonce;
        public PoPAuthenticationToken(Authentication authentication) {
            this.authentication = authentication;
        }
        public void setNonce(String nonce) {this.nonce = nonce;}
        public String getNonce() {return nonce;}
        public Collection<? extends GrantedAuthority> getAuthorities() {
            return authentication.getAuthorities();
        }
        public Object getCredentials() {return authentication.getCredentia
        public Object getDetails() {return authentication.getDetails();}
        public Object getPrincipal() {return authentication.getPrincipal()
        public boolean isAuthenticated() {return authentication.isAuthenti
        public void setAuthenticated(boolean isAuthenticated)
```

```
            throws IllegalArgumentException {
            authentication.setAuthenticated(isAuthenticated);
        }
        public String getName() {return authentication.getName();}
    }
```

11. Create the `PoPTokenExtractor` class as follows, that has the responsibility to create an instance of `PoPAuthenticationToken` with the nonce field that must be presented in the HTTP header:

```
    public class PoPTokenExtractor implements TokenExtractor {
        private TokenExtractor delegate;
        public PoPTokenExtractor(TokenExtractor delegate) {
            this.delegate = delegate;
        }
        public Authentication extract(HttpServletRequest request) {
            Authentication authentication = delegate.extract(request);
            if (authentication != null) {
                PoPAuthenticationToken popAuthenticationToken
                    = new PoPAuthenticationToken(authentication);
                popAuthenticationToken.setNonce(request.getHeader("nonce")
                return popAuthenticationToken;
            }
            return authentication;
        }
    }
```

12. As the authentication process needs to be customized, create the `PoPAuthenticationManager` class as follows:

```
    public class PoPAuthenticationManager implements AuthenticationManager
        private AuthenticationManager authenticationManager;
        public PoPAuthenticationManager(AuthenticationManager authenticati
            this.authenticationManager = authenticationManager;
        }
        @Override
        public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
            Authentication authenticationResult = authenticationManager
                .authenticate(authentication);

            if (authenticationResult.isAuthenticated()) {
                if (authentication instanceof PoPAuthenticationToken) {
                    PoPAuthenticationToken popAuthentication
                        = (PoPAuthenticationToken) authentication;

                    String nonce = popAuthentication.getNonce();
                    if (nonce == null) {
                        throw new UnapprovedClientAuthenticationException(
                            "This request does not have a valid signed nor
                    }

                    String token = (String) popAuthentication.getPrincipal
                    try {
                        JWT jwt = JWTParser.parse(token);
                        String publicKey = jwt.getJWTClaimsSet().getClaim(
                        JWK jwk = JWK.parse(publicKey);
                        JWSObject jwsNonce = JWSObject.parse(nonce);
                        JWSVerifier verifier = new RSASSAVerifier((RSAKey)
                        if (!jwsNonce.verify(verifier)) {
                            throw new InvalidTokenException(
                              "Client hasn't possession of given token");
                        }
                    } catch (Exception e) {
                        throw new RuntimeException(e);
                    }
                }
            }
            return authenticationResult;
        }
    }
```

13. To finish the Resource Server configuration, create the `OAuth2ResourceServer` class as follows:

```
    @Configuration @EnableResourceServer
    public class OAuth2ResourceServer extends ResourceServerConfigurerAdap
        @Bean @Primary
        public DefaultTokenServices tokenServices() {
            DefaultTokenServices tokenServices = new DefaultTokenServices(
            tokenServices.setTokenStore(tokenStore());
            return tokenServices;
        }
        @Bean
        public TokenStore tokenStore() {
            JwtTokenStore tokenStore = new JwtTokenStore(accessTokenConver
            return tokenStore;
        }
        @Bean
        public JwtAccessTokenConverter accessTokenConverter() {
            JwtAccessTokenConverter converter = new JwtAccessTokenConverte
            converter.setVerifier(verifier());
            converter.setSigningKey("non-prod");
            return converter;
        }
        @Bean
        public SignatureVerifier verifier() {
            return new MacSigner("non-prod");
        }
        @Override
        public void configure(ResourceServerSecurityConfigurer resources)
            resources.tokenExtractor(new PoPTokenExtractor(new BearerToker
            OAuth2AuthenticationManager oauth = new OAuth2AuthenticationMa
            oauth.setTokenServices(tokenServices());
            resources.authenticationManager(new PoPAuthenticationManager(c
        }
        @Override
        public void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                    .anyRequest().authenticated().and()
                .requestMatchers().antMatchers("/api/**");
        }
    }
```

14. Now, as the OAuth Provider provides both Authorization Server and Resource Server, create the following API to be accessed by the client using OAuth 2.0:

```
    @Controller
    public class UserController {

        @RequestMapping("/api/profile")
        public ResponseEntity<UserProfile> myProfile() {
            String username = (String) SecurityContextHolder.getContext()
                    .getAuthentication().getPrincipal();
            String email = username + "@mailinator.com";
            UserProfile profile = new UserProfile(username, email);
            return ResponseEntity.ok(profile);
        }

        public static class UserProfile {
            private String name;
            private String email;
            public UserProfile(String name, String email) {
                this.name = name;
                this.email = email;
            }
            // getters and setters hidden for brevity
        }
    }
```

How it works...

There is a lot of things going on this recipe that basically is about the responsibilities of Authorization Server and Resource Server. The Authorization Server is ready to receive a **JSON Web Key** (**JWK**) structure containing the public key that should be embedded within the access token in response for a token request. To allow for a claim to be embedded in an access token the Authorization Server creates a **JSON Web Signature** (**JWS**) type of JWT. For didactic purposes, it's being signed with the symmetric key `non-prod`.

It would be better to not place the symmetric key directly on the source code as we did for this recipe. The recommended way would be to save the symmetric key within an external file and it is preferable for it to be encrypted.

On the other hand, we have the Resource Server validating the header attribute `nonce` besides validating the JWS token that might be presented by the client application. The main logic for both JWS and `nonce` validation is written into the `PoPAuthenticationManager` class. Notice that when trying to authenticate a request for an OAuth protected resource, the access token is validated using the same symmetric key (which is non-prod) used to sign the JWS token by the Authorization Server. After this validation, this JWT is parsed, which allows the application to extract the embedded `public_key` claim explained before. This `public_key` is used to validate the `nonce` attribute that must be generated by the client application, which is in possession of the correspondent private key.

As an exercise to increase the safeness of the application, you can manage nonces for a given client ID so that this attribute cannot be the same for further requests leveraging the real semantic of a nonce attribute. It's recommended because as the name implies, it should be a random number to be used just once.

There's more...

To interact with the OAuth Provider created in this recipe you need to have a key pair and have to share the public key with the Authorization Server and also need to generate a nonce signed with the related private key before sending requests to the Resource Server. All of these steps should be a little complicated to do manually, but don't worry because the next recipe will present you with how to write a client application that is able to interact using proof-of-possession key semantics.

See also

- Generating access tokens as JWT
- Using proof-of-possession key on the client side

Using proof-of-possession key on the client side

In the previous recipe, you can understand all the pieces required for enabling proof-of-possession key semantics at the OAuth Provider side. This recipe will help you to create an application that is able to interact with the OAuth Provider, being capable to prove that it has possession of a private key related to the public key embedded within the JWS token that can be presented to the Resource Server when interacting with protected resources.

Getting ready

This recipe presents you with how to implement the client side of a proof-of-possession key stack, which is implemented as a Spring Boot application using Java 8, Maven, Spring Web, Spring Security, and Nimbus (which provides JWE and encryption capabilities). The other dependencies will be described in the *How to do it...* section.

How to do it...

Now you will be guided to create the `pop-client` application using proof-of-possession key semantics. This project is also available on GitHub in the `Chapter05` folder:

1. Create the project `pop-client` as a Spring Boot application (I recommend the usage of Spring Initializr). At Spring Initializr add `Web` and `Security` as dependencies.
2. Import the `pop-client` project to your IDE as a Maven project and add the following dependencies into the `pom.xml` file:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$ -->
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
<dependency>
  <groupId>com.nimbusds</groupId>
  <artifactId>nimbus-jose-jwt</artifactId>
  <version>4.23</version>
</dependency>
```

3. Add the following content into the `application.properties` file (the `key-value` property is being defined for a Resource Server to validate JWT tokens):

```
server.port=9000
spring.http.converters.preferred-json-mapper=jackson
spring.thymeleaf.cache=false
security.user.name=adolfo
security.user.password=123
```

4. Create one sub-package named `oauth` and add the following class to manage the `KeyPair` required to sign the nonce header to prove the possession of a given private key:

```
@Component
public class JwkKeyPairManager {
    private final JWK clientJwk;

    public JwkKeyPairManager() {
        KeyPair keyPair = createRSA256KeyPair();
        this.clientJwk = new RSAKey.Builder((RSAPublicKey) keyPair.get
                .privateKey((RSAPrivateKey) keyPair.getPrivate())
                .keyID(UUID.randomUUID().toString()).build();
    }

    public JWK createJWK() { return clientJwk.toPublicJWK(); }

    public String getSignedContent(String content) {
        try {
            RSASSASigner signer = new RSASSASigner((RSAKey) clientJwk)
            JWSObject jwsObject = new JWSObject(
                    new JWSHeader.Builder(JWSAlgorithm.RS256)
                        .keyID(clientJwk.getKeyID()).build(),
                    new Payload(content));
            jwsObject.sign(signer);
            return jwsObject.serialize();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private KeyPair createRSA256KeyPair() {
        try {
            KeyPairGenerator generator = KeyPairGenerator.getInstance(
            generator.initialize(2048);
            return generator.generateKeyPair();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
}
```

5. Then create the following `PoPTokenRequestEnhancer` class to enable the client to send the public key to Authorization Server when requesting a new access token:

```
    @Component
    public class PoPTokenRequestEnhancer implements RequestEnhancer {
        @Autowired
        private JwkKeyPairManager keyPairManager;

        public void enhance(AccessTokenRequest request,
            OAuth2ProtectedResourceDetails resource,
            MultiValueMap<String, String> form, HttpHeaders headers) {
            form.add("public_key", keyPairManager.createJWK().toJSONString
        }
    }
```

6. Now, as the client application needs to send the signed nonce attribute to

the Resource Server when interacting with an OAuth protected resource, create the following class that implements `ClientHttpRequestInterceptor`, which intercepts any HTTP request before interacting with an external endpoint:

```
@Component
public class HttpRequestWithPoPSignatureInterceptor
    implements ClientHttpRequestInterceptor, ApplicationContextAware {
    private ApplicationContext applicationContext;

    @Autowired
    private JwkKeyPairManager keyPairManager;

    @Override
    public void setApplicationContext(ApplicationContext applicationCo
        throws BeansException {
        this.applicationContext = applicationContext;
    }

    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] bo
        ClientHttpRequestExecution execution) throws IOException {
        OAuth2ClientContext clientContext =
            applicationContext.getBean(OAuth2ClientContext.class);
        OAuth2AccessToken accessToken = clientContext.getAccessToken()
        request.getHeaders().set("Authorization", "Bearer " + accessTo
        request.getHeaders().set("nonce", keyPairManager.getSignedCont
            UUID.randomUUID().toString()));
        return execution.execute(request, body);
    }
}
```

7. Before creating the client configuration, create the following class that will be in charge of saving and retrieving access tokens for a given user:

```
@Service
public class OAuth2ClientTokenSevices implements ClientTokenServices {
    private ConcurrentHashMap<String, ClientUser> users = new Concurre
    public OAuth2AccessToken getAccessToken(OAuth2ProtectedResourceDet
        Authentication authentication) {
        ClientUser clientUser = getClientUser(authentication);
        if (clientUser.accessToken == null) return null;
        DefaultOAuth2AccessToken oAuth2AccessToken = new
           DefaultOAuth2AccessToken(clientUser.accessToken);
        oAuth2AccessToken.setAdditionalInformation(clientUser.additior
        oAuth2AccessToken.setExpiration(new Date(clientUser.expiratior
        return oAuth2AccessToken;
    }
    public void saveAccessToken(OAuth2ProtectedResourceDetails resourc
            Authentication authentication, OAuth2AccessToken accessTok
        ClientUser clientUser = getClientUser(authentication);
        clientUser.accessToken = accessToken.getValue();
        clientUser.expirationTime = accessToken.getExpiration().getTim
```

```
            clientUser.additionalInformation = accessToken.getAdditionalIr

            users.put(clientUser.username, clientUser);
        }

        @Override
        public void removeAccessToken(OAuth2ProtectedResourceDetails resou
                Authentication authentication) {
            users.remove(getClientUser(authentication).username);
        }

        private ClientUser getClientUser(Authentication authentication) {
            String username = ((User) authentication.getPrincipal()).getUs
            ClientUser clientUser = users.get(username);
            if (clientUser == null) {
                clientUser = new ClientUser(username);
            }
            return clientUser;
        }

        private static class ClientUser {
            private String username;
            private String accessToken;
            private Map<String, Object> additionalInformation;
            private long expirationTime;
            public ClientUser(String username) {
                this.username = username;
            }
        }
    }
```

8. Now create the `ClientConfiguration` class as follows:

```
    @Configuration @EnableOAuth2Client
    public class ClientConfiguration {
        @Autowired
        private ClientTokenServices clientTokenServices;
        @Autowired
        private OAuth2ClientContext oauth2ClientContext;
        @Autowired
        private HttpRequestWithPoPSignatureInterceptor interceptor;
        @Autowired
        private PoPTokenRequestEnhancer requestEnhancer;
        @Bean
        public AuthorizationCodeResourceDetails authorizationCode() {
            AuthorizationCodeResourceDetails resourceDetails
                 = new AuthorizationCodeResourceDetails();
            resourceDetails.setId("oauth2server");
            resourceDetails.setTokenName("oauth_token");
            resourceDetails.setClientId("clientapp");
            resourceDetails.setClientSecret("123456");
            resourceDetails.setAccessTokenUri("http://localhost:8080/oauth
            resourceDetails.setUserAuthorizationUri("http://localhost:8080
            resourceDetails.setScope(Arrays.asList("read_profile"));
            resourceDetails.setPreEstablishedRedirectUri(("http://localhos
            resourceDetails.setUseCurrentUri(false);
            resourceDetails.setClientAuthenticationScheme(AuthenticationSc
            return resourceDetails;
        }
        @Bean
        public OAuth2RestTemplate oauth2RestTemplate() {
            OAuth2ProtectedResourceDetails resourceDetails = authorizatior
            OAuth2RestTemplate template = new OAuth2RestTemplate(
                resourceDetails, oauth2ClientContext);

            AuthorizationCodeAccessTokenProvider authorizationCode
                = new AuthorizationCodeAccessTokenProvider();
            authorizationCode.setTokenRequestEnhancer(requestEnhancer);

            AccessTokenProviderChain provider = new AccessTokenProviderCha
                Arrays.asList(authorizationCode));

            provider.setClientTokenServices(clientTokenServices);
            template.setAccessTokenProvider(provider);
            template.setInterceptors(Arrays.asList(interceptor));
            return template;
        }
    }
```

9. As this is a Client application, we need to create something for the final user to interact with. Let's create a dashboard by defining the following classes. The first one is `UserProfile` as presented in the following code:

```
    public class UserProfile {
        private String name;
        private String email;
        // getters and setters hidden for brevity
    }
```

10. Now create the `Entry` class to represent fictitious data into the user's dashboard:

```
    public class Entry {
        private String value;
        public Entry(String value) { this.value = value; }
        public String getValue() { return value; }
    }
```

11. And create the `UserDashboard` controller as follows:

```
    @Controller
    public class UserDashboard {
        @Autowired
        private OAuth2RestTemplate restTemplate;
        @GetMapping("/")
        public String home() { return "index"; }
        @GetMapping("/callback")
        public ModelAndView callback(String code, String state) {
            return new ModelAndView("forward:/dashboard");
        }
        @GetMapping("/dashboard")
        public ModelAndView dashboard() {
            List<Entry> entries = Arrays.asList(
                    new Entry("entry 1"), new Entry("entry 2"));
            ModelAndView mv = new ModelAndView("dashboard");
            mv.addObject("entries", entries);
            tryToGetUserProfile(mv);
            return mv;
        }
        private void tryToGetUserProfile(ModelAndView mv) {
            String endpoint = "http://localhost:8080/api/profile";
            try {
                UserProfile userProfile = restTemplate.getForObject(endpoi
                mv.addObject("profile", userProfile);
            } catch (HttpClientErrorException e) {
                throw new RuntimeException("it was not possible to retriev
            }
        }
    }
```

12. Now, we just have to create the frontend. Create the `index.html` file within the `resource/templates` project's directory with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymel
<head>
    <title>client app</title>
</head>
<body>
    <a href="/dashboard">Go to your dashboard</a>
</body>
</html>
```

13. And create the `dashboard.html` file within the same directory as `index.html`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:th="http://www.thymeleaf.org">
<head><title>client app</title></head>
<body>
  <h1>that's your dashboard</h1>
  <table>
    <tr><td><b>That's your entries</b></td></tr>
    <tr th:each="entry : ${entries}">
      <td th:text="${entry.value}">value</td>
    </tr>
  </table>
  <h3>your profile from [Profile Application]</h3>
  <table>
    <tr>
        <td><b>name</b></td>
```

```
            <td th:text="${profile.name}">username</td>
        </tr>
        <tr>
            <td><b>email</b></td>
            <td th:text="${profile.email}">email</td>
        </tr>
      </table>
    </body>
    </html>
```

14. Finally, make sure that `PoPClientApplication` (or any other class annotated with `@SpringBootApplication`) implements `ServletContextInitializer` interface and its respective method as follows:

```
    @Override
    public void onStartup(ServletContext servletContext) throws ServletExc
        servletContext.getSessionCookieConfig().setName("client-session");
    }
```

How it works...

Basically, to prove that the client application is in possession of a given private key, this project firstly creates a `KeyPair` within a Spring managed component, which is the class `JwkKeyPairManager`. The `KeyPair` is maintained as a JWK instance that allows us to create a JWK with the public key to be presented to the Authorization Server. It also allows to generate a signed content with the JWK being managed to create `nonce` values to be presented to the Resource Server.

To start interacting with an OAuth Provider that is ready to work with proof-of-possession key semantics, start the `pop-server` application created in the *Using proof-of-possession key semantics on OAuth 2.0 Povider* recipe. Then start the `pop-client` application and go to `http://localhost:9000` to authenticate yourself using the credentials defined within the `application.properties` file of `pop-client` and `pop-server` projects (which in my case is `adolfo` and `123` for username and password).

See also

- Using proof-of-possession key semantics on OAuth 2.0 Provider

OpenID Connect for Authentication

In this chapter, we will cover the following recipes:

- Authenticating Google's users through Google OpenID Connect
- Obtaining user information from Identity Provider
- Using Facebook to authenticate users
- Using Google OpenID Connect with Spring Security 5
- Using Microsoft and Google OpenID providers together with Spring Security 5

Introduction

This chapter introduces you to using Spring Security OAuth2 and the new version of Spring Security to add support for OpenID Connect. OpenID comes as an additional layer that builds authentication on top of OAuth 2.0, which is an authorization protocol. After reading this chapter, you will be able to integrate with any OpenID provider, such as Google, Microsoft, and Facebook Identity Providers. This chapter also presents you with invaluable recipes using the most up-to-date version of Spring Security. Spring Security 5 integrates all security-related projects, such as OAuth 2.0, OpenID, Single Sign On, and Spring Social, which are implemented as separate projects at the moment.

All recipes in this chapter rely on registered applications on Identity Providers. As I can't share my application credentials, I have created an `application.properties.sample` *file for each project created for each recipe. If you download any source code from GitHub, you have to rename this file to* `application.properties` *and replace the credentials when presented.*

Authenticating Google's users through Google OpenID Connect

To allow users to be authenticated by third-party applications, it's required that the Identity Provider you are considering to integrate with is trustworthy. Google is a great example of an Identity Provider that meets such a requirement. In addition, Google conforms to the OpenID Connect specification and is OpenID certified, as explained in the Google Identity Platform guide, which is available at https://developers.google.com/identity/protocols/OpenIDConnect.

This recipe shows you how to use Google OpenID Connect and Spring Security OAuth 2.0 to authenticate users who have an account on Google. As described by OpenID Connect, users share their identity information instead of their resources, as allowed by OAuth 2.0.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies as `Web`, `JPA`, `H2`, `Thymeleaf`, and `Security` (which will properly declare all the Spring Boot starters needed for this recipe). Do not forget to set up the Artifact and Group names.

How to do it...

This recipe creates the `google-connect` project, which is available on GitHub in the `Chapter06` folder. Import the generated project as a Maven project into your IDE and follow these steps:

1. Open the `pom.xml` file and add the following extra dependencies for `Spring Security OAuth2`, `Spring Security JWT`, and `Thymeleaf` extras for Spring Security (note that we are declaring the most up-to-date version of Spring Security OAuth2 at the time of writing this). I am assuming that you have already imported the starters for `Web`, `JPA`, `H2`, and `Security`:

```
<dependency>
   <groupId>org.thymeleaf.extras</groupId>
   <artifactId>thymeleaf-extras-springsecurity4</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.2.0.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-jwt</artifactId>
</dependency>
```

2. Open the `application.properties` file and add the following content:

```
spring.datasource.url=jdbc:h2:mem:packt;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialec
spring.jpa.properties.hibernate.hbm2ddl.auto=create
spring.thymeleaf.cache=false
google.config.clientId=client-id
google.config.clientSecret=client-secret

openid.callback-uri=/google/callback
openid.api-base-uri=/profile/*
```

3. Note that we have to register our application at Google API console to retrieve the credentials of our application, which are `client-id` and `client-secret`. To do that, register an application, as described in the

Accessing OAuth 2.0 Google protected resources bound to user's session recipe from the first chapter. Make sure that you register `http://localhost:8080` for the authorized JavaScript origins and `http://localhost:8080/google/callback` for authorized redirect URI so the application can receive the token ID and the respective access token.

4. Also, make sure you have generated the credentials for your newly registered application. Copy both the client ID and client secret, and replace the following attributes within the `application.properties` file (your credentials will be different from what follows):

```
google.config.clientId=50012j2.apps.googleusercontent.com
google.config.clientSecret=QPe-uIBwVV
```

5. Now let's start creating the source code to manage the user authentication. Create the `openid` sub-package and the `OpenIDAuthentication` class to hold the user identity (they basically represent the most used OpenID Connect claims). Make sure to import JPA annotations from `javax.persistence` package:

```
@Entity
public class OpenIDAuthentication {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String subject;
    private String provider;
    private long expirationTime;
    private String token;

    public boolean hasExpired() {
      OffsetDateTime expirationDateTime =
          OffsetDateTime.ofInstant(
            Instant.ofEpochSecond(expirationTime),
            ZoneId.systemDefault());
        OffsetDateTime now = OffsetDateTime.now(
            ZoneId.systemDefault());
        return now.isAfter(expirationDateTime);
    }
    // getters and setters hidden for brevity
}
```

6. Create the `GoogleUser.java` class within the same package with the following content (make sure that you return `true` for the `isAccountNonExpired`, `isAccountNonLocked`, and `isEnabled` methods):

```
@Entity
public class GoogleUser implements UserDetails {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String email;
    @OneToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private OpenIDAuthentication openIDAuthentication;

    @Deprecated GoogleUser() {}
    public GoogleUser(String email,
        OpenIDAuthentication openIDAuthentication) {
        this.email = email;
        this.openIDAuthentication = openIDAuthentication;
    }
    @Override public String getUsername() { return email; }
    @Override
    public Collection<? extends GrantedAuthority>
        getAuthorities() {
        return new ArrayList<>();
    }
    @Override public boolean isCredentialsNonExpired() {
        return !openIDAuthentication.hasExpired();
    }
    // remaining getters and setters ommitted
}
```

7. Then, create the `UserRepository` class to allow the application to retrieve user data by subject (which is the user's unique ID on Google). Import `@Param`; annotation from `org.springframework.data.repository.query` package.

```
public interface UserRepository
    extends JpaRepository<GoogleUser, Long> {
    @Query("select u from GoogleUser u " +
        "inner join u.openIDAuthentication o " +
        "where o.subject = :subject")
    Optional<GoogleUser> findByOpenIDSubject(
        @Param("subject") String subject);
}
```

8. When the user finishes authentication through Identity Provider, which is Google in our case, the application will receive a token ID containing some claims (which tell you who the authenticated user is). Create the following class within the `openid` sub-package (make sure to import `ObjectMapper` class from `com.fasterxml.jackson.databind` package):

```
public class Claims {
    private String iss;
    private String sub;
    private String at_hash;
    private String email;
    private Long exp;
    public static Claims createFrom(ObjectMapper jsonMapper,
        OAuth2AccessToken accessToken) {
        try {
            String idToken = accessToken.getAdditionalInformation()
                .get("id_token").toString();
            Jwt decodedToken = JwtHelper.decode(idToken);
            return jsonMapper.readValue(
                decodedToken.getClaims(), Claims.class);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
    public String getIss() { return iss; }
    public String getSub() { return sub; }
    public String getAt_hash() { return at_hash; }
    public String getEmail() { return email; }
    public Long getExp() { return exp; }
}
```

9. Given that the application we are creating has received the user claims represented through the previous structure, we have to persist a new user if she/he still doesn't exist in the database; otherwise, we have to retrieve if she/he already exists. To do that, create the `UserIdentity` class, as presented here:

```
    @Component
    public class UserIdentity {
        @Autowired
        private UserRepository repository;

        public GoogleUser findOrCreateFrom(Claims claims) {
            Optional<GoogleUser> userAuth = repository
               .findByOpenIDSubject(claims.getSub());

            GoogleUser user = userAuth.orElseGet(() -> {
                OpenIDAuthentication openId =
                    new OpenIDAuthentication();
                openId.setProvider(claims.getIss());
                openId.setSubject(claims.getSub());
                openId.setExpirationTime(claims.getExp());
                openId.setToken(claims.getAt_hash());
                return new GoogleUser(claims.getEmail(), openId);
            });
            if (!user.isCredentialsNonExpired()) {
                user.getOpenIDAuthentication()
                    .setExpirationTime(claims.getExp());
            }
            return user;
        }
    }
```

10. Now create the `GoogleProperties` class, which holds the credentials of our application. It's a good approach to not write the credentials directly in

the source code, which is why we are using `application.properties`:

```
@Component
@ConfigurationProperties(prefix = "google.config")
public class GoogleProperties {
    private String clientId;
    private String clientSecret;
    // getters and setters hidden for brevity
}
```

11. Then, create the following `GoogleConfiguration` class, which configures an instance of `OAuth2ProtectedResourceDetails`. This configuration presents the endpoints and other parameters needed for user authentication on Google OpenID Connect as well as to request for a new token ID and access token:

```
@Configuration @EnableOAuth2Client
public class GoogleConfiguration {
    @Autowired
    private GoogleProperties properties;
    @Bean
    public OAuth2ProtectedResourceDetails resourceDetails() {
        AuthorizationCodeResourceDetails details =
          new AuthorizationCodeResourceDetails();
        details.setClientId(properties.getClientId());
        details.setClientSecret(properties.getClientSecret());
        details.setUserAuthorizationUri(
           "https://accounts.google.com/o/oauth2/v2/auth");
        details.setAccessTokenUri(
           "https://www.googleapis.com/oauth2/v4/token");
        details.setPreEstablishedRedirectUri(
           "http://localhost:8080/google/callback");
        details.setScope(Arrays.asList(
           "openid", "email", "profile"));
        details.setUseCurrentUri(false);
        return details;
    }
    @Bean
    public OAuth2RestTemplate restTemplate(
        OAuth2ClientContext context) {
        OAuth2RestTemplate rest = new OAuth2RestTemplate(
             resourceDetails(), context);
        AccessTokenProviderChain providerChain =
             new AccessTokenProviderChain(
                Arrays.asList(
                  new AuthorizationCodeAccessTokenProvider()));
        rest.setAccessTokenProvider(providerChain);
        return rest;
    }
}
```

12. As Spring Security OAuth2 does not natively provide support for

OpenID Connect, we have to create some additional classes. To orchestrate the authentication flow, create the following `OpenIdConnectFilter` class, which extends from `AbstractAuthenticationProcessingFilter`. At the time of writing this, OpenID Connect as well as OAuth 2.0-related protocols are being added to Spring Security 5. In the *Using Google OpenID Connect with Spring Security 5* recipe, we will see how to take advantage of these new features added to Spring Security:

```
@Component
public class OpenIdConnectFilter extends AbstractAuthenticationProcess
    @Override
    public Authentication attemptAuthentication(
        HttpServletRequest request, HttpServletResponse response)
        throws AuthenticationException, IOException, ServletException
        return null;
    }
}
```

13. First of all, add the following attributes to the `OpenIdConnectFilter` class:

```
@Autowired
private OAuth2RestTemplate restTemplate;
@Autowired
private UserIdentity userIdentity;
@Autowired
private UserRepository repository;
@Autowired
private ObjectMapper jsonMapper;
private ApplicationEventPublisher eventPublisher;
private final AntPathRequestMatcher localMatcher;
```

14. Then, add the following private static class to override the authentication manager for the filter we are writing now:

```
private static class NoopAuthenticationManager
    implements AuthenticationManager {
    @Override
    public Authentication authenticate(
        Authentication authentication)
        throws AuthenticationException {
        throw new UnsupportedOperationException(
            "No authentication should be done");
    }
}
```

15. Now, add the following constructor method to the `OpenIdConnectFilter` class:

```
public OpenIdConnectFilter(
    @Value("${openid.callback-uri}") String callbackUri,
    @Value("${openid.api-base-uri}") String apiBaseUri) {
    super(new OrRequestMatcher(
        new AntPathRequestMatcher(callbackUri),
        new AntPathRequestMatcher(apiBaseUri)));
    this.localMatcher = new AntPathRequestMatcher("/profile/**");
    setAuthenticationManager(new NoopAuthenticationManager());
}
```

16. As we have to publish an event of authentication to notify Spring Security, override the `setApplicationEventPublisher` method and add the following private method:

```
@Override
public void setApplicationEventPublisher(
    ApplicationEventPublisher eventPublisher) {
    this.eventPublisher = eventPublisher;
    super.setApplicationEventPublisher(eventPublisher);
}
private void publish(ApplicationEvent event) {
    if (eventPublisher!=null) {
        eventPublisher.publishEvent(event);
    }
}
```

17. Now replace the content of the `attemptAuthentication` method from `OpenIdConnectFilter` with the following content:

```
try {
    OAuth2AccessToken accessToken = restTemplate.getAccessToken();

    Claims claims = Claims.createFrom(jsonMapper, accessToken);
    GoogleUser googleUser = userIdentity.findOrCreateFrom(claims);
    repository.save(googleUser);

    Authentication authentication =
      new UsernamePasswordAuthenticationToken(
        googleUser, null, googleUser.getAuthorities());
    publish(new AuthenticationSuccessEvent(authentication));
    return authentication;
} catch (OAuth2Exception e) {
    BadCredentialsException error = new BadCredentialsException(
            "Cannot retrieve the access token", e);
    publish(new OAuth2AuthenticationFailureEvent(error));
    throw error;
}
```

18. Our filter is now ready to start the Google authentication process and save the user identity received after an authentication. But with the current version of `OpenIdConnectFilter`, what happens if the token ID

expires? Actually, the token ID is the artifact that allows our application to identify the user as authenticated by Google. If it expires, we have to start a new process of authentication. To do that, override the `doFilter` method, as follows:

```
@Override
public void doFilter(ServletRequest req, ServletResponse res,
    FilterChain chain) throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    if (localMatcher.matches(request)) {
        restTemplate.getAccessToken();
        chain.doFilter(req, res);
    } else {
        super.doFilter(req, res, chain);
    }
}
```

19. Now it's time to create the Spring Security configuration, which will use all the structures we have created in previous steps to provide OpenID Connect authentication. Create the `security` sub-package and create the `SecurityConfiguration` class, as presented here:

```
@Configuration @EnableWebSecurity
public class SecurityConfiguration extends
    WebSecurityConfigurerAdapter {
    @Value("${openid.callback-uri}")
    private String callbackUri;
    @Value("${openid.api-base-uri}")
    private String apiBaseUri;
    @Autowired
    private OpenIdConnectFilter openIdConnectFilter;
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
          .addFilterAfter(new OAuth2ClientContextFilter(),
             AbstractPreAuthenticatedProcessingFilter.class)
          .addFilterAfter(openIdConnectFilter,
             OAuth2ClientContextFilter.class)
          .authorizeRequests()
          .antMatchers("/").permitAll().and()
          .authorizeRequests()
          .antMatchers(apiBaseUri).authenticated().and()
          .authorizeRequests().anyRequest().authenticated().and()
          .httpBasic().authenticationEntryPoint(
             new LoginUrlAuthenticationEntryPoint(callbackUri))
        .and()
           .logout()
           .logoutSuccessUrl("/")
           .permitAll().and()
           .csrf().disable();
    }
}
```

20. As we have configured all the classes needed for authentication, let's start creating the profile-related classes that will give something to the user to interact with. Create the `profile` sub-package and add the following class:

```
@Entity
public class Profile {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String hobbies;
    private String profession;
    @OneToOne
    private GoogleUser user;
    // getters and setters hidden for brevity
}
```

21. Create the respective `Profile` repository:

```
public interface ProfileRepository
    extends JpaRepository<Profile, Long> {
    Optional<Profile> findByUser(GoogleUser user);
}
```

22. As the user will log in with her/his Google account, she/he will be able to see their resources bound to a Google session within the application we are creating for this recipe. So, to exercise it, create the `ProfileController` class, as presented here (this class provides a simple way for the user to persist data in their name):

```
@Controller @RequestMapping("/profile")
public class ProfileController {
    @Autowired
    private ProfileRepository profileRepository;
    @GetMapping
    public ModelAndView profile() {
        GoogleUser user = (GoogleUser) SecurityContextHolder
          .getContext().getAuthentication().getPrincipal();
        Optional<Profile> profile = profileRepository
          .findByUser(user);
        if (profile.isPresent()) {
            ModelAndView mv = new ModelAndView("profile");
            mv.addObject("profile", profile.get());
            mv.addObject("openID", user.getOpenIDAuthentication());
            return mv;
        }
        return new ModelAndView("redirect:/profile/form");
    }
}
```

23. Also, create the following methods within the `ProfileController` class to allow users to persist new data about their profile:

```
@GetMapping("/form")
public ModelAndView form() {
    GoogleUser user = (GoogleUser) SecurityContextHolder
       .getContext().getAuthentication().getPrincipal();
    Optional<Profile> profile = profileRepository.findByUser(user);
    ModelAndView mv = new ModelAndView("form");
    if (profile.isPresent()) {
        mv.addObject("profile", profile.get());
    } else {
        mv.addObject("profile", new Profile());
    }
    return mv;
}

@PostMapping
public ModelAndView save(Profile profile) {
    GoogleUser user = (GoogleUser) SecurityContextHolder
      .getContext().getAuthentication().getPrincipal();
    profile.setUser(user);
    Profile newProfile = profileRepository.save(profile);
    ModelAndView mv = new ModelAndView("redirect:/profile");
    mv.addObject("profile", newProfile);
    return mv;
}
```

24. Create the following controller to handle requests to the application's root path:

```
@Controller @RequestMapping("/")
public class HomeController {
    @GetMapping
    public String home() { return "home"; }
}
```

25. Relative to the controllers we have created, we need to provide the frontend to allow user interaction. Create the `home.html` file inside the `src/main/resources/templates` directory with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymel
  xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4
<body>
<h2>Your online profile manager</h2>
<p sec:authorize="isAuthenticated()">
    You are logged in as
    <p sec:authentication="principal.username">John Doe</p>
    <a href="#" th:href="@{/profile}">Go to profile</a>
    <a href="#" th:href="@{/logout}">Logout</a>
</p>
```

```
<div sec:authorize="not isAuthenticated()">
  <a href="#" th:href="@{/profile}">Login with Google</a>
</div>
</body>
</html>
```

26. Create the `form.html` file within the same directory as `home.html`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:th="http://www.thymeleaf.org"
  xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4
<body>
<p sec:authorize="isAuthenticated()">
    You are logged in as
    <span sec:authentication="principal.username">John Doe</span>
    <a href="#" th:href="@{/logout}">Logout</a>
    <h2>Add your profile info</h2>
    <form th:action="@{/profile}" th:object="${profile}" method="post"
        <input type="hidden" id="id" th:field="*{id}"/>
        <b>Hobbies:</b><input id="hobbies" th:field="*{hobbies}"/>
        <b>Profession:</b><input id="profession" th:field="*{professio
        <button>Send</button>
        <a th:href="@{/profile}">Cancel</a>
    </form>
</p>
</body>
</html>
```

27. Create the `profile.html` file within the same directory of `form.html`, as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymel
xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">
<body>
<p sec:authorize="isAuthenticated()">
Logged as <span sec:authentication="principal.username"></span>
</p>
<h1>That's your profile</h1>
<p>hobbie:<span th:text="${profile.hobbies}">hobbies</span></p>
<p>profession:<span th:text="${profile.profession}">profession</span><

<h2>That's your OpenID info</h2>
<p>subject:<span th:text="${openID.subject}"></span></p>
<p>provider:<span th:text="${openID.provider}"></span></p>
<p>exp time(ms):<span th:text="${openID.expirationTime}"></span></p>
<p>access token:<span th:text="${openID.token}"></span></p>

<a href="#" th:href="@{/profile/form}">Add/Edit</a>
<p><a href="#" th:href="@{/logout}">Logout</a></p>
</body>
</html>
```

28. Now we are ready to run the application through the `mvn clean spring-boot:run` command or directly from the IDE.

How it works...

As already mentioned, OpenID Connect is built on top of OAuth 2.0 to provide authentication. So, as Spring Security OAuth2 already provides the means to build an OAuth client through the usage of `OAuth2RestTemplate`, we can take advantage of it to interact with the Google OpenID Connect service.

Actually, just configuring an OAuth 2.0 client is not enough. You're required to decode the token ID response (which is a JWT that conveys all user claims) retrieved when the user is authenticated in the Identity Provider. This is done within the `OpenIdConnectFilter` class that, after a retrieving user's claims, persists identity information and notifies Spring Security that the user is authenticated.

Another thing that is important to note is where all the URIs defined within `GoogleConfiguration` come from. These URIs and other details about the OpenID Connect provider's configuration can be found at https://accounts.google.com/.well-known/openid-configuration.

To see how this works in practice, start the application and go to `http://localhost:8080`. Then, click on the Login with Google link and you will be redirected to the Google authentication page. Then, click on Go to Profile and add some profile information about the user. After saving it, you will be able to see the persisted data and user identity info that came from Google after authentication.

See also

- Using Google OpenID Connect with Spring Security 5

Obtaining user information from Identity Provider

Sometimes, it's not enough to retrieve just identity info from a third-party authenticated user. Depending on the application being developed, you might need to retrieve permissions or roles related to the authenticated user. OpenID Connect defines a special endpoint where the Relying Party can request user information. That's the user info endpoint, where the Identity Provider can return authorized data as a JSON object or even a signed JWT. This recipe will show you how to retrieve additional info about the user, how to discover the `UserInfo` endpoint, and how to update data from an already known user.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. As this recipe adds the functionality to an OpenID Connect client, it also relies on the `google-connect` project, which was created for the *Authenticating Google's users through Google OpenID Connect* recipe. So, make sure you have implemented the source code, or download it from https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter06. The complete source code for this recipe is also available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter06/google-userinfo.

How to do it...

As this recipe relies on the existing source code, you can create a new project at Spring Initializr using `Web`, `JPA`, `H2`, `Thymeleaf`, and `Security` as dependencies (do not forget to set up the Artifact and Group names) and copy all the source code from the `google-connect` project. Rather than creating a new project, you can use the `google-connect` project directly, so we can add `UserInfo` features, which are described in the following steps (for this recipe, the project name was defined as `google-userinfo`):

1. Open the `application.properties` and make sure it has the same properties that were defined for the `google-connect` project.
2. Then, make sure all the classes and templates are the same as `google-connect`.
3. As a user info attribute, we will retrieve the username and will persist on the local database. To do that, open the `OpenIDAuthentication` class and add the following attribute with corresponding getters and setters:

```
private String name;
public String getName() { return name; }
public void setName(String name) { this.name = name; }
```

4. To retrieve the username, we need to make a request to the `UserInfo` endpoint, which is an OAuth 2.0-protected resource. So, we need the endpoint by itself and an access token. The endpoint is defined within the discovery document, which is available at https://accounts.google.com/.well-known/openid-configuration for Google. The access token we will use is the one provided by the Spring Security OAuth2 context, which is represented by the `OAuth2AccessToken` class. Given that, we have all we need to request user info and create the `UserInfoService` class inside the `openid` sub-package, as shown here:

```
@Service
public class UserInfoService {

    public Map<String, String> getUserInfoFor(OAuth2AccessToken access
        RestTemplate restTemplate = new RestTemplate();

        RequestEntity<MultiValueMap<String, String>> requestEntity
```

```
                = new RequestEntity<>(getHeader(accessToken.getValue()),
                    HttpMethod.GET,
                    URI.create("https://www.googleapis.com/oauth2/v3/useri
            ResponseEntity<Map> result = restTemplate.exchange(
                    requestEntity, Map.class);

            if (result.getStatusCode().is2xxSuccessful()) {
                return result.getBody();
            }
            throw new RuntimeException(
                 "It wasn't possible to retrieve userInfo");
        }
        private MultiValueMap getHeader(String accessToken) {
            MultiValueMap httpHeaders = new HttpHeaders();
            httpHeaders.add("Authorization", "Bearer " + accessToken);
            return httpHeaders;
        }
    }
```

5. The `UserInfoService` class has to be used when we have the opportunity to retrieve user data after authentication. So, the best place to start using `UserInfoService` is within the `OpenIdConnectFilter` class. Add the following attribute to the `OpenIdConnectFilter` class:

```
    @Autowired
    private UserInfoService userInfoService;
```

6. Then, add the following private method to the `OpenIdConnectFilter` class, which will be in charge of invoking the `getUserInfoFor` method from `UserInfoService`:

```
    private String getUserNameFromUserInfo(
        OAuth2AccessToken accessToken, String subject) {
        Map<String, String> userInfo = userInfoService.getUserInfoFor(acce
        if (!userInfo.get("sub").equals(subject)) {
            throw new RuntimeException(
                "sub element of ID Token must be the same from UserInfo er
        }
        String userName = userInfo.get("name");
        return userName;
    }
```

7. Now change the body of the `attemptAuthentication` method from `OpenIdConnectFilter` to this:

```
    @Override
    public Authentication attemptAuthentication(
        HttpServletRequest request, HttpServletResponse response)
        throws AuthenticationException, IOException, ServletException {
        try {
            OAuth2AccessToken accessToken = restTemplate.getAccessToken();
            Claims claims = Claims.createFrom(jsonMapper, accessToken);
            GoogleUser googleUser = userIdentity.findOrCreateFrom(claims);

            // retrieving username from UserInfo endpoint
            String userName = getUserNameFromUserInfo(accessToken,
                googleUser.getOpenIDAuthentication().getSubject());
            googleUser.getOpenIDAuthentication().setName(userName);

            repository.save(googleUser);
            Authentication authentication = new UsernamePasswordAuthentica
                    googleUser, null, googleUser.getAuthorities());
            publish(new AuthenticationSuccessEvent(authentication));
            return authentication;
        } catch (OAuth2Exception e) {
            BadCredentialsException error = new BadCredentialsException(
                    "Cannot retrieve the access token", e);
            publish(new OAuth2AuthenticationFailureEvent(error));
            throw error;
        }
    }
```

8. Once the application is ready to retrieve the username from the Google `UserInfo` endpoint, let's present it in the profile template. Open `profile.html` and make sure that you present the username through the usage of the `openId.name` property, as presented in the new version of the `html` profile, as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymel
xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">
<body>
<p sec:authorize="isAuthenticated()">
Logged as <span sec:authentication="principal.username"></span>
</p>
<h1>That's your profile</h1>
<p>hobbie:<span th:text="${profile.hobbies}">hobbies</span></p>
<p>profession:<span th:text="${profile.profession}">profession</span><

<h2>That's your OpenID info</h2>
<p><b>name:</b><span th:text="${openID.name}"></span></p>
<p>subject:<span th:text="${openID.subject}"></span></p>
<p>provider:<span th:text="${openID.provider}"></span></p>
<p>exp time(ms):<span th:text="${openID.expirationTime}"></span></p>
<p>access token:<span th:text="${openID.token}"></span></p>

<a href="#" th:href="@{/profile/form}">Add/Edit</a>
<p><a href="#" th:href="@{/logout}">Logout</a></p>
</body>
</html>
```

9. Now run the application through the `mvn clean spring-boot:run` command

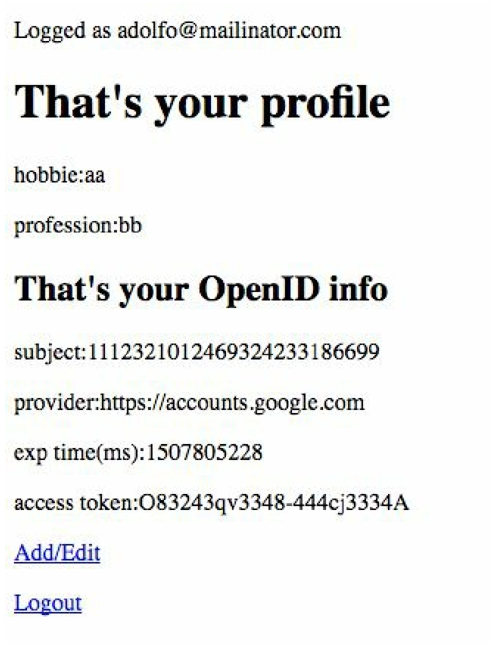
or directly from the IDE.

How it works...

The main difference between this recipe and the *Authenticating Google's users through Google OpenID Connect* recipe is that, now, this application is retrieving user data through Google's `UserInfo` endpoint. As you might notice, we are validating whether the user info retrieved really belongs to the currently authenticated user. This is done by comparing the `sub` claim retrieved from the `UserInfo` endpoint, with the `sub` claim present inside the token ID that the Identity Provider returned after authenticating the user. This is possible because as per the OpenID Connect specification, the `sub` claim must be returned with claims retrieved from the `UserInfo` endpoint.

Another important feature is that we are using the access token returned from the authentication process to be able to access the `UserInfo` endpoint. Do not confuse the `id_token` retrieved as additional information within the `OAuth2AccessToken` instance with the value of `OAuth2AccessToken`. The `id_token` from the additional information is just used to retrieve the claims that identify the user.

To check whether everything is working fine, start the application and go to `http://localhost:8080`. Once you are on the home page of our application, click on the Login with Google link, and when redirected to the Google authorization page, authenticate yourself with a valid Google account. After authenticating, you can type some fictitious data about your hobby and your profession, and after sending the form data, you would be able to see something similar to what is shown in the following screenshot:


Logged as adolfo@mailinator.com

That's your profile

hobbie:aa

profession:bb

That's your OpenID info

subject:1112321012469324233186699

provider:https://accounts.google.com

exp time(ms):1507805228

access token:O83243qv3348-444cj3334A

Add/Edit

Logout

There's more...

This recipe and the *Authenticating Google's users through Google OpenID Connect* recipe both showed you how to use an Identity Provider to authenticate users using an authentication protocol built on top of OAuth 2.0. That is OpenID Connect. You can go further into validations, using the `at_hash` claim, for example, to protect the Relying Party against a **Cross-site request forgery** (**CSRF**) attack. As we are relying on the Authorization Code grant type, there is no need to do that at all.

But there are more validations, such as checking the token ID signature. You can try it as an exercise, and if you want to know how to check for JWT integrity, take a look at the *Validating JWT tokens at the Resource Server side* or *Validating asymmetric signed JWT tokens* recipes from Chapter 5, Self Contained Tokens with JWT.

Now, you are ready to use Spring Security OAuth2 and understand how the OpenID Connect protocol works by looking at the source code for this recipe. Members from the Spring Security project are working on a new version of Spring Security that will provide most of the implementations we did for this recipe. At the time of writing this, Spring Security 5 is a milestone build, and support for OAuth 2.0 and OpenID Connect will be provided just for the client first. Support for the Resource Server is supposed to be provided next year, and for the Authorization Server, there is a lot of work to be done yet.

Further ahead, we will see how to use Spring Security 5 in the *Using Google OpenID Connect with Spring Security 5* and *Using Microsoft and Google OpenID providers together with Spring Security 5* recipes.

See also

- Authenticating Google's users through Google OpenID Connect
- Using Google OpenID Connect with Spring Security 5
- Using Microsoft and Google OpenID providers together with Spring Security 5

Using Facebook to authenticate users

Although Facebook does not implement OpenID Connect, it can also be used for authentication since it provides user identity information and the authentication context. Thus, the client application (or Relying Part) is able to recognize the user as being present and who the user claims to be. After reading this recipe, you will see how to use Spring Security OAuth2 to provide OAuth 2.0 as the basis for authentication (actually, OAuth 2.0 is about authorization, but it can be used as the basis for authentication).

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies: as `Web`, `JPA`, `H2`, `Thymeleaf`, and `Security` (which will properly declare all the Spring Boot starters needed for this recipe). Do not forget to set up the Artifact and Group names.

How to do it...

This recipe creates the `facebook-login-oauth2` project, which is available on GitHub in the `Chapter06` folder. Import the generated project as a Maven project into your IDE and follow these steps:

1. Open the `pom.xml` file and add the following extra dependencies for `Spring Security OAuth2` and `Thymeleaf` extras for Spring Security (note that we are declaring the most up-to-date version of Spring Security OAuth2 at the time of writing this). I am assuming that you have already imported the starters for `Web`, `Thymeleaf`, `JPA`, `H2`, and `Security`:

```
<dependency>
   <groupId>org.thymeleaf.extras</groupId>
   <artifactId>thymeleaf-extras-springsecurity4</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.2.0.RELEASE</version>
</dependency>
```

2. Copy the properties related to the database connection and `Thymeleaf` from the `google-connect` project, which was created in the *Authenticating Google's users through Google OpenID Connect* recipe.
3. Open the `application.properties` file and paste the properties copied from the previous step.
4. Paste the following properties related to OAuth 2.0 resource details for Facebook:

```
facebook.config.clientId=client_id
facebook.config.clientSecret=client_secret
facebook.config.userInfoUri=https://graph.facebook.com/me
facebook.config.appTokenUri=https://graph.facebook.com/v2.10/oauth/acc
facebook.config.appAuthorizationUri=https://www.facebook.com/v2.10/dia
facebook.config.redirectUri=http://localhost:8080/callback

facebook.filter.api-base-uri=/profile*/*
facebook.filter.callback-uri=/callback
```

5. Register a new application in the Facebook developers console and create Facebook credentials by choosing the Facebook login product.

Then, configure the appropriate redirect URI in the client OAuth Settings panel which must be `http://localhost:8080/callback`.

6. After configuring the Facebook login product in the Facebook developers console, go to the application dashboard and copy App ID and App Secret, which are the client ID and client secret values, respectively.
7. Update the `facebook.config.clientId` and `facebook.config.clientSecret` properties with their respective values within the `application.properties` file.
8. Now, copy the `HomeController`, `Profile`, `ProfileController`, and `ProfileRepository` classes from the `google-connect` project that was created for the *Authenticating Google's users through Google OpenID Connect* recipe. Then, paste them within the `user` sub-package in our current project (notice that some of these classes won't compile because they still relies on `GoogleUser` class which will be replaces by `FacebookUser` class later).
9. Create the `openid` subpackage so we can create the authentication-related classes.
10. Inside the `openid` subpackage, create the `FacebookLoginData` class, as follows (add getters and setters for each attribute):

```
@Entity
public class FacebookLoginData {
    @Id private String id;
    private long expirationTime;
    private long issuedAt;
    private String token;
    private String name;
    public boolean hasExpired() {
        OffsetDateTime expirationDateTime = OffsetDateTime.ofInstant(
                Instant.ofEpochSecond(expirationTime), ZoneId.systemDe
        OffsetDateTime now = OffsetDateTime.now(ZoneId.systemDefault()
        return now.isAfter(expirationDateTime);
    }
    // getters and setters hidden
}
```

11. Create the `FacebookUser` class that must implement the `UserDetails` interface and must have a `FacebookLoginData` reference, as shown here:

```
@Entity
public class FacebookUser implements UserDetails {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private FacebookLoginData openIDAuthentication;
    FacebookUser() {} // jpa eyes only
    public FacebookUser(FacebookLoginData openIDAuthentication) {
        this.openIDAuthentication = openIDAuthentication;
    }
    // other methods hidden for brevity
}
```

12. Return `true` for all Boolean methods, but for `isCredentialsNonExpired`, just return the following:

```
return !openIDAuthentication.hasExpired();
```

13. Make sure that you return at least an empty `ArrayList` inside the `getAuthorities` method from the `FacebookUser` class.
14. Implement `getUsername` method as follows:

```
@Override
public String getUsername() {
    return openIDAuthentication.getName();
}
```

14. Create getters for `id` and `openIDAuthentication` attributes.
15. Create the following repository to allow the searching of a given user by their Facebook identity:

```
public interface UserRepository
    extends JpaRepository<FacebookUser, Long> {
    @Query("select u from FacebookUser u " +
            "inner join u.openIDAuthentication o " +
            "where o.id = :facebookId")
    Optional<FacebookUser> findByFacebookId(
        @Param("facebookId") String facebookId);
}
```

16. Replace all the usages of `GoogleUser` with `FacebookUser`.

17. Now, create the `FacebookProperties` class to map the properties defined within the `application.properties` file (do not forget to create getters and setters):

```
@Component
@ConfigurationProperties(prefix = "facebook.config")
public class FacebookProperties {
    private String clientId;
    private String clientSecret;
    private String userInfoUri;
```

```
        private String appTokenUri;
        private String appAuthorizationUri;
        private String redirectUri;
        // getters and setters omitted for brevity
    }
```

18. As we are configuring an OAuth 2.0 client, let's define an instance of `OAuth2ProtectedResourceDetails` and see how `OAuth2RestTemplate` should work. To do that, create the `FacebookConfiguration` class, as follows:

```
    @Configuration @EnableOAuth2Client
    public class FacebookConfiguration {
        @Autowired
        private FacebookProperties properties;
        @Bean
        public OAuth2ProtectedResourceDetails resourceDetails() {
            AuthorizationCodeResourceDetails details
            = new AuthorizationCodeResourceDetails();
            details.setClientId(properties.getClientId());
            details.setClientSecret(properties.getClientSecret());
            details.setUserAuthorizationUri(properties.getAppAuthorizatior
            details.setAccessTokenUri(properties.getAppTokenUri());
            details.setPreEstablishedRedirectUri(properties.getRedirectUri
            details.setScope(Arrays.asList("email", "public_profile"));
            details.setClientAuthenticationScheme(AuthenticationScheme.que
            details.setUseCurrentUri(false);
            return details;
        }
        @Bean
        public OAuth2RestTemplate restTemplate(OAuth2ClientContext context
            OAuth2RestTemplate rest = new OAuth2RestTemplate(
            resourceDetails(), context);
            rest.setAccessTokenProvider(
                new AccessTokenProviderChain(
                    Arrays.asList(new AuthorizationCodeAccessTokenProvider
            return rest;
        }
    }
```

19. Similar to what we did for previous recipes, we have to create a filter to retrieve identity information about the user so the application can consider the user authenticated. But before creating the filter by itself, we need to create some required functionalities, such as the `UserInfoService` class, which will be in charge of retrieving the user identity. Create the following class within the same package as `FacebookConfiguration`:

```
    @Service
    public class UserInfoService {
        @Autowired
    private FacebookProperties properties;
    public Map<String, String> getUserInfoFor(OAuth2AccessToken access
        RestTemplate restTemplate = new RestTemplate();
        RequestEntity<Void> requestEntity = new RequestEntity<>(
                getHeader(accessToken.getValue()), HttpMethod.GET,
                URI.create(properties.getUserInfoUri()));
        ParameterizedTypeReference<Map<String, String>> typeRef =
                new ParameterizedTypeReference<Map<String, String>>()
        ResponseEntity<Map<String, String>> result = restTemplate.exch
                requestEntity, typeRef);
        if (result.getStatusCode().is2xxSuccessful()) {
            return result.getBody();
        }
        throw new RuntimeException("It wasn't possible to retrieve use
    }
    private MultiValueMap<String, String> getHeader(String accessToker
        MultiValueMap<String, String> httpHeaders = new HttpHeaders();
        httpHeaders.add("Authorization", "Bearer " + accessToken);
        return httpHeaders;
    }
}
```

20. Instead of using `UserInfoService` within the filter that will be created, we will inject it inside the `FacebookUserIdentity` class, as shown here. As the following class also needs a reference to an instance of `OAuth2AccessToken`, they can seamlessly interact with each other:

```
@Service public class FacebookUserIdentity {
    @Autowired
    private UserRepository repository;
    @Autowired
    private UserInfoService userInfoService;
    public FacebookUser findOrCreateFrom(OAuth2AccessToken accessToker
        Map<String, String> userInfo = userInfoService
           .getUserInfoFor(accessToken);
        Optional<FacebookUser> userAuth = repository
           .findByFacebookId(userInfo.get("id"));
        FacebookUser user = userAuth.orElseGet(() -> {
            FacebookLoginData loginData = new FacebookLoginData();
            loginData.setName(userInfo.get("name"));
            loginData.setId(userInfo.get("id"));
            loginData.setExpirationTime(accessToken.getExpiration().ge
            loginData.setToken(accessToken.getValue());
            return new FacebookUser(loginData);
        });
        return user;
    }
}
```

21. Now we are ready to start creating `FacebookLoginFilter`, which will be in charge of orchestrating the authentication process using the classes created earlier. Create the `FacebookLoginFilter` class with the following

initial content (both classes `Authentication` and `AuthenticationException` must be imported from `org.springframework.security.core` package):

```
@Component public class FacebookLoginFilter
  extends AbstractAuthenticationProcessingFilter {
  protected FacebookLoginFilter() {
     super("/callback");
  }
  public Authentication attemptAuthentication(
    HttpServletRequest request, HttpServletResponse response)
    throws AuthenticationException, IOException, ServletException {
        return null;
  }
}
```

22. As we don't want to use `AuthenticationManager` within our filter, create the following private static class inside `FacebookLoginFilter`:

```
private static class NoopAuthenticationManager implements Authenticati
    public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
        throw new UnsupportedOperationException(
        "No authentication should be done with this AuthenticationMana
    }
}
```

23. Then, add the following attributes and then replace the current constructor with this:

```
@Autowired private OAuth2RestTemplate restTemplate;
@Autowired private FacebookUserIdentity userIdentity;
@Autowired private UserRepository repository;
private ApplicationEventPublisher eventPublisher;
private final AntPathRequestMatcher localMatcher;

public FacebookLoginFilter(
    @Value("${facebook.filter.callback-uri}") String callbackUri,
    @Value("${facebook.filter.api-base-uri}")String apiBaseUri) {
    super(new OrRequestMatcher(
        new AntPathRequestMatcher(callbackUri),
        new AntPathRequestMatcher(apiBaseUri)));
    this.localMatcher = new AntPathRequestMatcher(apiBaseUri);
    setAuthenticationManager(new NoopAuthenticationManager());
}
```

24. When the user is considered to be authenticated, the application needs to notify Spring Security about it. Add the following methods to the `FacebookLoginFilter` class:

```
public void setApplicationEventPublisher(
            ApplicationEventPublisher eventPublisher) {
            this.eventPublisher = eventPublisher;
            super.setApplicationEventPublisher(eventPublisher);
        }
        private void publish(ApplicationEvent event) {
            if (eventPublisher!=null) {
                eventPublisher.publishEvent(event);
            }
        }
```

25. Override the `doFilter` method, as follows, so we can check for token validity when invoking `getAccessToken` from the `OAuth2ResTemplate` instance:

```
        public void doFilter(ServletRequest req, ServletResponse res,
            FilterChain chain) throws IOException, ServletException {
            HttpServletRequest request = (HttpServletRequest) req;
            if (localMatcher.matches(request)) {
                restTemplate.getAccessToken();
                chain.doFilter(req, res);
            } else {
                super.doFilter(req, res, chain);
            }
        }
```

26. Finally, replace the content of the `attemptAuthentication` method with the following:

```
        try {
            OAuth2AccessToken accessToken = restTemplate.getAccessToken();
            FacebookUser facebookUser = userIdentity.findOrCreateFrom(accessTo
            repository.save(facebookUser);
            Authentication authentication = new UsernamePasswordAuthenticatior
               facebookUser, null,
               Arrays.asList(new SimpleGrantedAuthority("ROLE_USER")));
            publish(new AuthenticationSuccessEvent(authentication));
            return authentication;
        } catch (OAuth2Exception e) {
            BadCredentialsException error = new BadCredentialsException(
                    "Cannot retrieve the access token", e);
            publish(new OAuth2AuthenticationFailureEvent(error));
            throw error;
        }
```

27. Copy all the `html` files present within the `google-connect` project from the *Authenticating Google's users through Google OpenID Connect* recipe to the `src/main/resources/templates` directory of the `facebook-login-auth2` project.
28. Open the `home.html` file and replace Google entries with Facebook to be consistent with this recipe.

29. Open the `profile.html` file and replace all of its content with the following:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymel
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecur
<body>
<p sec:authorize="isAuthenticated()">
    Logged as <span sec:authentication="principal.username"></span>
</p>
<h1>That's your profile</h1>
<p>hobbie:<span th:text="${profile.hobbies}">hobbies</span></p>
<p>profession:<span th:text="${profile.profession}">profession</span><

<h2>That's your facebook authentication info</h2>
<p>name:<span th:text="${openID.name}"></span></p>
<p>facebook id:<span th:text="${openID.id}"></span></p>
<p>exp time(ms):<span th:text="${openID.expirationTime}"></span></p>
<p>access token:<span th:text="${openID.token}"></span></p>

<a href="#" th:href="@{/profile/form}">Add/Edit</a>
<p><a href="#" th:href="@{/logout}">Logout</a></p>
</body>
</html>
```

30. Create the `SecurityConfiguration` class within the `security` sub-package, as shown here:

```
@Configuration @EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapte
    @Autowired
    private FacebookLoginFilter facebookLoginFilter;
    protected void configure(HttpSecurity http) throws Exception {
        http
          .addFilterAfter(new OAuth2ClientContextFilter(),
             AbstractPreAuthenticatedProcessingFilter.class)
          .addFilterAfter(facebookLoginFilter, OAuth2ClientContextFilt
          .authorizeRequests()
          .antMatchers("/", "/callback").permitAll().and()
          .authorizeRequests()
          .antMatchers("/profile/*").authenticated().and()
          .authorizeRequests().anyRequest().authenticated().and()
          .httpBasic().authenticationEntryPoint(
              new LoginUrlAuthenticationEntryPoint("/callback")).and()
          .logout().logoutSuccessUrl("/").permitAll().and()
          .csrf().disable();
    }
}
```

How it works...

Basically, the project created for this recipe, works using the Authorization Code flow from the OAuth 2.0 protocol, with Facebook acting as both the Authorization Server and Resource Server. Facebook is also acting as the Resource Server because we are retrieving the user identity info through the `UserInfoService` class, which makes a request to an OAuth 2.0 protected resource.

Start the application either by performing `mvn spring-boot:run` on the command line or start directly from your IDE by running the class annotated with `@SpringBootApplication`.

After starting the application, go to `http://localhost:8080` and click on the Login with Facebook link. Then, log in to Facebook using some valid credentials and you will be redirected back to the client application. When the user is redirected back to our application, we make a request to `https://graph.facebook.com/me` using the generated bearer access token.

As you might realize, we are using a classic OAuth authorization flow. But how can it be considered an authentication mechanism? That's because we are retrieving user identity information at the time of the user authorization.

See also

- Authenticating Google's users through Google OpenID Connect
- Obtaining user information from Identity Provider

Using Google OpenID Connect with Spring Security 5

OAuth 2.0 protocol and OpenID Connect are both stable specifications that are evolving through additional specifications. On the other hand, frameworks that implement both technologies are always changing quickly with regard to new requirements, famous paradigms, new versions of languages, and even to improve maintainability.

That's our case with Spring Security and Spring Security OAuth2. At the time of writing this, all the projects related to OAuth 2.0, OpenID Connect, and social connectivity are being condensed within Spring Security 5. Spring Security 5 brings in lots of advantages, improving the way we add security to our applications. But at the moment, we can just count with client implementations for OAuth-related technologies. Through this recipe, let's explore how to authenticate users with Google OpenID Connect.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies as `Web`, `Thymeleaf`, and `Security` (this will properly declare all the Spring Boot starters needed for this recipe).

As we need to use Spring Security 5, let's change the version of Spring Boot on the Spring Initializr website to ***2.0.0.M4*** *(do not forget to set up the Artifact and Group names). Make sure to use* ***2.0.0.M4*** *version of Spring Boot because as it is a milestone version it might change in future versions.*

How to do it...

This recipe creates the `google-openid-spring5` project, which is available on GitHub in the `Chapter06` folder. Import the generated project as a Maven project into your IDE and follow these steps:

When using the repositories required by version 2.0.0.M4 of Spring Boot, I had experienced an error processing the pom.xml file from Eclipse Maven Plugin (Netbeans and IntelliJ do not present this issue). To fix this problem if on Eclipse, you need to install m2 extensions plugin. To do so, go to Help/ Install New Software and add a new repository using either https://otto.takari.io/content/sites/m2e.extras/m2eclipse-mavenarchiver/0.17.2/N/LATEST/ or http://repo1.maven.org/maven2/.m2e/connectors/m2eclipse-mavenarchiver/0.17.2/N/LATEST/ URLs. This solution was found on StackOverflow web site at https://stackoverflow.com/questions/37555557/m2e-error-in-mavenarchiver-getmanifest.

1. Open the `pom.xml` file and add the following extra dependencies:

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

2. Create two sub-packages within the generated package, namely `security` and `user`.
3. Within the `security` sub-package, create the following class with getters and setters for each declared attribute:

```
@Component @ConfigurationProperties("security.oauth2.client.google")
public class GoogleRegistrationProperties {
    private String clientId;
    private String clientSecret;
    private String scopes;
    private String redirectUri;
    private String authorizationUri;
```

```
        private String tokenUri;
        private String clientName = "Google";
        private String clientAlias = "google";
        private String userInfoUri;
        private String userInfoNameAttributeKey = "name";
        private String jwkSetUri;
        private AuthorizationGrantType authorizedGrantType
                = AuthorizationGrantType.AUTHORIZATION_CODE;
    }
```

4. Then, create the `SecurityConfiguration` class shown here within the same package as `GoogleRegistrationProperties`:

```
    @Configuration @EnableWebSecurity
    public class SecurityConfiguration
        extends WebSecurityConfigurerAdapter {
    }
```

5. Add the following method within the `SecurityConfiguration` class:

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/callback").permitAll()
            .anyRequest().authenticated().and()
            .oauth2Login().and();
    }
```

6. Inject an instance of `GoogleRegistrationProperties` and create the following method in `SecurityConfiguration`:

```
    @Autowired
    private GoogleRegistrationProperties properties;
    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        ClientRegistration registration = new ClientRegistration
          .Builder(properties.getClientId())
            .authorizationUri(properties.getAuthorizationUri())
            .clientSecret(properties.getClientSecret())
            .tokenUri(properties.getTokenUri())
            .redirectUri(properties.getRedirectUri())
            .scope(properties.getScopes().split(","))
            .clientName(properties.getClientName())
            .clientAlias(properties.getClientAlias())
            .jwkSetUri(properties.getJwkSetUri())
            .authorizationGrantType(properties.getAuthorizedGrantType())
            .userInfoUri(properties.getUserInfoUri())
            .build();

        return new InMemoryClientRegistrationRepository(
            Arrays.asList(registration));
```

```
}
```

7. Let's create some controllers in the `user` sub-package, so the user can interact with our application. First, create the `DashboardController` class, as shown here:

```
@Controller @RequestMapping("/dashboard")
public class DashboardController {
    @GetMapping
    public ModelAndView profile() {
        DefaultOidcUser user = (DefaultOidcUser) SecurityContextHolder
            .getContext().getAuthentication().getPrincipal();
        ModelAndView mv = new ModelAndView("dashboard");
        mv.addObject("profile", user.getUserInfo().getClaims());
        return mv;
    }
}
```

8. And finally, create the `HomeController` class within the same package as `DashboardController`, as follows:

```
@Controller @RequestMapping("/")
public class HomeController {
    @GetMapping
    public String home() { return "home"; }
}
```

9. With respect to each controller, we need to create some view artifacts. Create the `dashboard.html` file in the `/src/main/resources/templates` folder with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:th="http://www.thymeleaf.org">
<body>
<h1>You are logged in</h1>
<p><b>Name:</b><span th:text="${profile.name}"></span></p>
<p><b>Email:</b><span th:text="${profile.email}"></span></p>
<p><b>Profile:</b><a th:href="${profile.profile}">Google + account</a>
<p><b>Picture:</b>
    <img width="100" height="100" th:src="${profile.picture}"></p>
<a href="/">Home</a>
<hr/>
</body>
</html>
```

10. Finally, create the `home.html` file within the same folder as `dashboard.html` with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:th="http://www.thymeleaf.org"
  xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4
<body>
<h2>Login with Google</h2>
<p sec:authorize="isAuthenticated()">
<div>You are logged in</div>
</p>
<a th:href="@{/dashboard}">Go to dashboard</a>
</body>
</html>
```

11. All the configurations were written dynamically through the usage of the `GoogleRegistrationProperties` class, which is annotated with `@ConfigurationProperties`. All the attributes of this class rely on attributes that must be defined in the `application.properties` file. Add the following attributes within `application.properties`:

```
spring.thymeleaf.cache=false
security.oauth2.client.google.client-secret=client-secret
security.oauth2.client.google.client-id=client-id
security.oauth2.client.google.scopes=openid,email,profile
security.oauth2.client.google.redirectUri=http://localhost:8080/oauth2
security.oauth2.client.google.authorizationUri=https://accounts.google
security.oauth2.client.google.tokenUri=https://www.googleapis.com/oaut
security.oauth2.client.google.userInfoUri=https://www.googleapis.com/o
security.oauth2.client.google.jwkSetUri=https://www.googleapis.com/oau
```

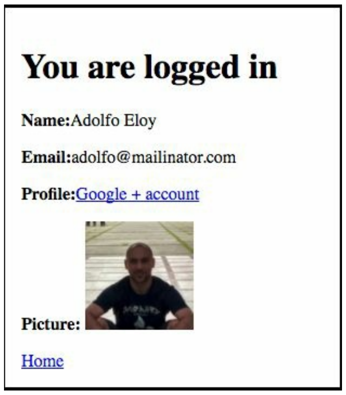
12. Replace the `client-secret` and `client-id` placeholders within the `application.properties` file with their corresponding values that must be retrieved from a registered application on Google. If you want to know how to register an application on Google, go back to Chapter 1, *OAuth 2.0 Foundations*.
13. Make sure that you have registered the Authorized Redirect URI as `http://localhost:8080/oauth2/authorize/code/google` on Google developers console.

How it works...

The authentication mechanism works in the same way presented in the *Authenticating Google's users through Google OpenID Connect* recipe. However, as you may realize, this recipe presents an easier way to implement an OpenID Connect Relying Party using Spring Security 5.

Each provider can be configured by declaring a `ClientRegistration` entry, which must be added to an instance of `ClientRegistrationRepository`. In addition, the token ID received when the user is authenticated is validated against a public key that is retrieved from JWK set URI declared on Google discovery document.

Start the application and go to `http://localhost:8080` in order to start the authentication process. Thus, click on the Google link so the application redirects you to Google's authentication page. Once you authorize our application to use the user identity info, you will be redirected back to `home.html`, where you can click on the Go to Dashboard link, which will lead you to the profile page, where you will see something similar to what is shown in the following figure:



You are logged in

Name: Adolfo Eloy

Email:adolfo@mailinator.com

Profile: Google + account

Picture:

Home

See also

- Authenticating Google's users through Google OpenID Connect

Using Microsoft and Google OpenID providers together with Spring Security 5

This recipe shows you how to use Spring Security 5 to seamlessly implement an OpenID Connect Relying Party providing more than one authentication method. After reading this recipe, you will be able to enable Google and Microsoft authentication to your website.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies as `Web`, `Thymeleaf`, and `Security` (this will properly declare all the Spring Boot starters needed for this recipe).

As we need to use Spring Security 5, let's change the version of Spring Boot on the Spring Initializr website to 2.0.0.M4 (do not forget to set up the Artifact and Group names).

Besides the fact that you have to register an application on Google, you also need to register an application on Microsoft Azure. Azure is the Microsoft Cloud platform where you can build and deploy applications. To run this recipe, you must create a free account on Microsoft Azure and register an application through Active Directory service (Don't worry you receive a temporary charge of 1 US$ when registering. This is done just to check if your credit card is in good standing. This charge should be refunded after a short period). I won't cover how you can register an application on Microsoft Azure because there are a lot of concepts, such as tenants, which would require almost a chapter. You can see how to register an application on Microsoft Azure as well as how to create credentials by reading the official documentation at https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-integrating-applications.

How to do it...

This recipe creates the `microsoft-login` project, which is available on GitHub in the `Chapter06` folder. Import the generated project as a Maven project into your IDE and follow these steps:

1. Open the `pom.xml` file and add the following extra dependencies:

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

2. Create two sub-packages within the generated package, namely `security` and `user`.
3. Within the `security` sub-package, create the following class with getters and setters for each declared attribute. This class holds Google properties for authentication with OpenID Connect. Create getters for all attributes and setters for `clientId`, `clientSecret`, and `scopes` attributes:

```
@Component
@ConfigurationProperties("security.oauth2.client.google")
  public class GoogleRegistrationProperties {
  private String clientId;
  private String clientSecret;
  private String scopes;
  private String redirectUri = "http://localhost:8080/oauth2/authorize
  private String authorizationUri = "https://accounts.google.com/o/oau
  private String tokenUri = "https://www.googleapis.com/oauth2/v4/toke
  private String clientName = "Google";
  private String clientAlias = "google";
  private String userInfoUri = "https://www.googleapis.com/oauth2/v3/u
  private String userInfoNameAttributeKey = "name";
  private String jwkSetUri = "https://www.googleapis.com/oauth2/v3/cer
  private AuthorizationGrantType authorizedGrantType =
          AuthorizationGrantType.AUTHORIZATION_CODE;
    // getters and setters omitted
}
```

4. Now add the following class, which holds Microsoft properties for authentication (you have to set up your own **tenant identifier** in tenant placeholders). Create getters for all attributes and setters for `clientId`, `clientSecret`, and `scopes` attributes:

```
@Component
@ConfigurationProperties("security.oauth2.client.microsoft")
public class MicrosoftRegistrationProperties {
private String clientId;
private String clientSecret;
private String scopes;
private String redirectUri = "http://localhost:8080/oauth2/authorize/c
private String authorizationUri = "https://login.microsoftonline.com/{
private String tokenUri = "https://login.microsoftonline.com/{tenant}/
private String clientName = "Microsoft";
private String clientAlias = "microsoft";
private AuthorizationGrantType authorizedGrantType =
      AuthorizationGrantType.AUTHORIZATION_CODE;
private String userInfoUri = "https://login.microsoftonline.com/{tenar
private String userInfoNameAttributeKey = "name";
private String jwkSetUri = "https://login.microsoftonline.com/common/c
  // getters and setters omitted
}
```

5. Open the `application.properties` file and add the following content (replace all the credentials for Google and Microsoft applications):

```
spring.thymeleaf.cache=false

security.oauth2.client.google.client-secret=<google-client-secret>
security.oauth2.client.google.client-id=<google-client-id>
security.oauth2.client.google.scopes=openid,email,profile

security.oauth2.client.microsoft.client-secret=<microsoft-client-secre
security.oauth2.client.microsoft.client-id=<microsoft-client-id>
security.oauth2.client.microsoft.scopes=openid,profile,email
```

6. Create the `SecurityConfiguration` class with the following content:

```
@Configuration @EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapte
@Autowired
    private GoogleRegistrationProperties google;
@Autowired
    private MicrosoftRegistrationProperties microsoft;
@Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
          .anyRequest().authenticated().and()
          .oauth2Login();
    }
}
```

7. Then, create the following private methods within `SecurityConfiguration` to declare `ClientRegistration` instances for Google and Microsoft identity providers:

```
private ClientRegistration createGoogleRegistration() {
    ClientRegistration registration = new ClientRegistration.Builder(
         google.getClientId())
        .authorizationUri(google.getAuthorizationUri())
        .clientSecret(google.getClientSecret())
        .tokenUri(google.getTokenUri())
        .redirectUri(google.getRedirectUri())
        .scope(google.getScopes().split(","))
        .clientName(google.getClientName())
        .clientAlias(google.getClientAlias())
        .jwkSetUri(google.getJwkSetUri())
        .authorizationGrantType(google.getAuthorizedGrantType())
        .userInfoUri(google.getUserInfoUri())
        .build();
return registration;
}
private ClientRegistration createMicrosoftRegistration() {
    ClientRegistration registration = new ClientRegistration.Builder(m
        .authorizationUri(microsoft.getAuthorizationUri())
        .clientSecret(microsoft.getClientSecret())
        .tokenUri(microsoft.getTokenUri())
        .redirectUri(microsoft.getRedirectUri())
        .scope(microsoft.getScopes().split(","))
        .clientName(microsoft.getClientName())
        .clientAlias(microsoft.getClientAlias())
        .jwkSetUri(microsoft.getJwkSetUri())
        .authorizationGrantType(microsoft.getAuthorizedGrantType())
        .userInfoUri(microsoft.getUserInfoUri())
        .clientAuthenticationMethod(ClientAuthenticationMethod.POST)
        .build();
return registration;
}
```

8. Finally, to the `SecurityConfiguration` class, add the following bean declaration, which makes use of `ClientRegistration` private methods:

```
@Bean
public ClientRegistrationRepository clientRegistrationRepository() {
return new InMemoryClientRegistrationRepository(Arrays.asList(
        createGoogleRegistration(),
        createMicrosoftRegistration()));
}
```

9. To allow the user to interact with our application, create the following class, which aggregates both home and dashboard endpoints:

```
@Controller @RequestMapping("/")
public class ApplicationController {
```

```
@GetMapping
    public String home() { return "home"; }
@GetMapping("/dashboard")
public ModelAndView dashboard() {
        DefaultOidcUser user = (DefaultOidcUser) SecurityContextHolder
          .getContext().getAuthentication().getPrincipal();
        ModelAndView mv = new ModelAndView("dashboard");
        mv.addObject("profile", user.getUserInfo().getClaims());
return mv;
    }
}
```

10. Create the respective template files within the `src/main/resources/templates` directory. The first one is `home.html`, as shown here:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecur
<body>
<h2>Use multiple identity providers</h2>
<p sec:authorize="isAuthenticated()">
<div>You are logged in</</span>div>
</p>
<a th:href="@{/dashboard}">Go to dashboard</a>
</body>
</html>
```

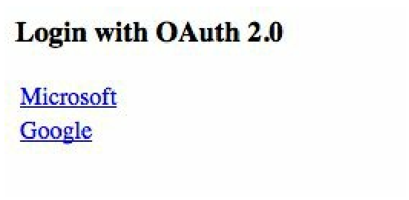
11. And the last one is `dashboard.html`, with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<body>
<div style="border: 3px solid black; width: 15%; padding: 15px">
<h1>You are logged in</h1>
<p><b>Name:</b><span th:text="${profile.name}"></span></p>
<p><b>Email:</b><span th:text="${profile.email}"></span></p>
<p><b>Profile:</b><span th:href="${profile.sub}"></span></p>
<a href="/">Home</a>
</div>
</body>
</html>
```

How it works...

Using Spring Security 5, you can work with multiple Identity Providers at the same time. As you can see, we simply had to declare two `ClientRegistration` entries: one for Google and another for Microsoft. Both `ClientRegistration` instances were grouped together within `InMemoryClientRegistrationRepository`. If you want to persist this identity information, you can also implement the `ClientRegistrationRepository` interface to save or retrieve the user identity from a database.

Start the application and go to `http://localhost:8080`. Now, you should see two links for both Identity Providers being supported, as shown here:


Login with OAuth 2.0

Microsoft
Google

Once you click on one of these options, you will be redirected to the Identity Provider's login page, and if you have never approved our application before, you should see the user consent page. After approving the usage of the user identity, you will be redirected back to our application, where you can go to the user's dashboard and see some of the retrieved claims.

See also

- Authenticating Google's users through Google OpenID Connect
- Using Google OpenID Connect with Spring Security 5

Implementing Mobile Clients

This chapter presents the following recipes:

- Preparing an Android development environment
- Creating an Android OAuth 2.0 client using an Authorization Code with the system browser
- Creating an Android OAuth 2.0 client using the Implicit grant type with the system browser
- Creating an Android OAuth 2.0 client using the embedded browser
- Using the Password grant type for the client app provided by the OAuth 2 server
- Protecting an Android client with PKCE
- Using dynamic client registration with mobile applications

Introduction

As the development of mobile applications increases, there is also an increase of API consumption, which needs to be performed securely. Native mobile applications are considered as *public* client types because of not being able to store confidential data as web applications. Because of the nature of native applications, the use of OAuth 2.0 might be hard to implement in a safe manner. In some cases, the most we can do is reduce the vulnerabilities or minimize the time window for an attack to be performed. The RFC 6749 by itself does not provide details on how to safely implement native mobile clients, which has been addressed by a recently published **Request for Comments** (**RFC**) at the time of writing, that is, RFC 8252—OAuth 2.0 for native apps.

This chapter will present you with how to implement native clients through an Android platform, using the grant types provided by OAuth 2.0 and some OAuth 2.0 profiles such as **Proof Key for Code Exchange by OAuth Public Clients** (**PKCE**) that helps in protecting the use of the Authorization Code.

Notice that we will be interacting with an Authorization Server that does not provide TLS/SSL protected connections. When running in production this must be addressed by just using TSL/SSL protected connection. We will be using raw HTTP throughout the recipes jus for didactical purposes.

Preparing an Android development environment

This recipe will present you with how to prepare the android development environment, showing you how to install Android Studio and how to configure virtual devices to start running the recipes. This recipe also presents you how you can create a new Android application through Android Studio.

Getting ready

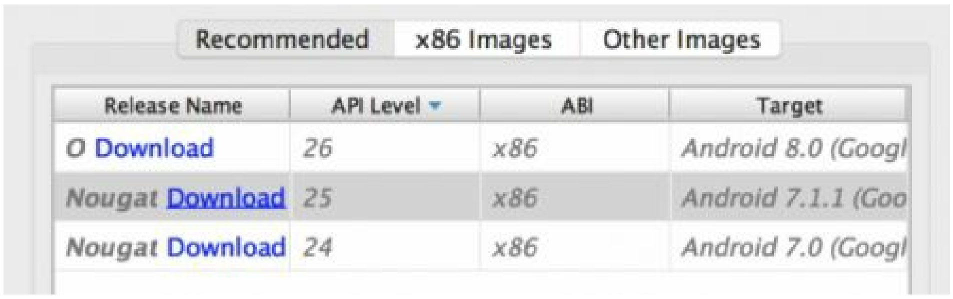
To run this recipe you need Java 8 and Maven installed, because all the client applications will interact with an API provided by an OAuth 2.0 Provider application. You also have to install Android Studio to create and run the clients that will be implemented throughout this chapter. The OAuth 2.0 Provider is already created and you can download it from GitHub (from the `server` project created within the `Chapter07` directory) at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter07/server. This server application provides both the Authorization Server and the Resource Server components, which allows the use of most grant types, as well as dynamic client registration and PKCE strategies to protect the Authorization Code from interception attacks (you will see more about PKCE in the *Protecting an Android client with PKCE* recipe).

How to do it...

Perform the following steps to start using Android Studio to create mobile native applications:

1. The first thing to do is to download the latest version of Android Studio which is available at https://developer.android.com/studio/index.html. You will be guided to download Android Studio for your currently running operational system.
2. The installation instructions for each **operational system** (**OS**) will be provided at https://developer.android.com/studio/install.html when you start the Android Studio download.
3. Open Android Studio, click on Start a new Android Studio project, define the name of the application (I have defined it as `AuthCodeApp`), and define the base package name and the project location of your preference.
4. Click on Next and select a minimum SDK. To run our recipes you can go with API 21 minimum version (Android 4.0.3).
5. Click on the Next button again until you reach the screen where you can add an Activity to your application. Select Empty Activity and click on Next again.
6. Use the defaults for Activity and Layout names and click on Finish (it may take a few minutes to start the IDE for the first time).

7. Now, go to the Run | Run App menu item of Android Studio. As we are running Android Studio for the first time, we first need to create a new virtual device to emulate smartphone. At Select Deployment Target, click on Create New Virtual Device and select an appropriate hardware.
8. For the purposes of this chapter, you can install Nexus 5x and click on Next. Then you will be prompted to select a system image (which you will probably need to install). On the Recommended tab, select Android 7.7.1, as presented in the following screenshot:



Release Name	API Level	ABI	Target
O Download	26	x86	Android 8.0 (Googl
Nougat Download	25	x86	Android 7.1.1 (Goo
Nougat Download	24	x86	Android 7.0 (Googl

Notice that if you are running Android Studio on any hardware that does not provide VT-x technology, you will need to install an ARM system image. To do so, click on Other images and select Android 7.1.1 where the ABI (Application Binary Interface) is armabi-v7a or arm64-v8a.

9. Use the default **Android Virtual Device** (**AVD**) name and click on Finish.
10. Now you are ready to run the sample application. Select the installed virtual device and click on OK to start running our recently created example. When running for the first time, Android Studio will ask you if you want to install Instant Run, which improves productivity when you change the application and want to run it without the need to build the entire APK. I recommend you install Instant Run.

You might notice some problems when running virtual devices on Windows and it might have some different causes. The most common is that you might need to enable virtual features in the BIOS configuration. This book won't cover how to solve these problems on Windows. In any case, if you have a physical device it will be better to use it instead of a virtual device.

11. After running the sample application you should just see a simple screen with the text `Hello World` at the center.

How it works...

This recipe just showed you how to install Android Studio and how to create and run your first android application. It's important to notice that you can use virtual devices as well as physical devices. Make sure your environment is running properly and do not forget to run the server application which will provide both the OAuth 2 Provider and the API to allow our native applications to retrieve the user's profile information.

As a disclaimer, when you stop the server application, I recommend you remove the mobile client apps created in the following recipes. That's because the server application is using in-memory databases and everything created for some clients will be lost if the server stops (and your client apps will start experiencing some issues because they lack some data as access tokens or client credentials).

Creating an Android OAuth 2.0 client using an Authorization Code with the system browser

This recipe presents you with how to create a native OAuth 2.0 client application for Android, which integrates with an OAuth 2.0 protected API using the Authorization Code grant type. The OAuth 2.0 specification (RFC 6749) states that public clients shouldn't use the Authorization Code grant type. On the other hand, the recently published RFC 8252 states that Authorization Code should be used in conjunction with dynamic client registration and PKCE validation (we will see more about both approaches later in this chapter).

The application created for this recipe, uses client id and client secret issued by a pre-registered client application just for brevity of the recipe. But bear in mind that, as per OAuth 2.0 specification, the Authorization Server must not issue client secret for native client that aren't specific running on a specific device.

Getting ready

To run this recipe you need the `server` application that is provided together with the book's source code examples at GitHub, within the `Chapter07` directory. If you haven't downloaded the project yet, you can download all the source code for this book at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/. Make sure that this project is running; to do so, you can just go to the `chapter-7/server` directory and run the `mvn spring-boot:run` command. In addition, you will need Android Studio to create the recipe.

How to do it...

Create a new project in Android Studio, which can be named `AuthCodeApp`, and perform the following steps (when creating the application make sure to automatically create an empty activity named `MainActivity`). The complete source code for this recipe can be download at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter07/AuthCodeApp.

1. As we need to interact with an API through HTTP requests, let's add `retrofit2` as the dependency for our project. The `retrofit2` is a great library that helps when interacting with REST APIs. To add `retrofit2` to our project, open the application `build.gradle` file and add the following content within the `dependencies` declaration:

```
compile 'com.squareup.retrofit2:retrofit:2.3.0'
compile 'com.squareup.retrofit2:converter-jackson:2.3.0'
compile 'com.squareup.okhttp3:logging-interceptor:3.9.0'
```

2. In the same `build.gradle` file, add the following content at the end of the android object declaration:

```
packagingOptions {
    exclude 'META-INF/LICENSE'
}
```

3. Now let's create the package structure for our project so we can have a well organized application. Within the project's base package, create `client`, and `presenter` packages. Within the `client` package, create `interceptor`, `oauth2` and `profile` sub-packages (we will refer to these sub-packages later, in the next steps).
4. Before creating the activities for our application, let's create all the infrastructure needed to interact with the server APIs. Within the `client/oauth2` sub-package, create the `AccessToken` class with the following content (this class will represent an OAuth 2.0 access token and will help when interacting with OAuth 2.0 protected APIs). Make sure to add all the respective getters and setters for each attribute:

```
public class AccessToken {
    @JsonProperty("access_token")
    private String value;
    @JsonProperty("token_type")
    private String tokenType;
    @JsonProperty("expires_in")
    private Long expiresIn;
    @JsonIgnore
    private Long issuedAt = new Date().getTime();
    private String scope;
    public boolean isExpired() {
       Long expirationTimeInSeconds = (issuedAt / 1000) + expiresIn;
       Long nowInSeconds = (new Date().getTime()) / 1000;
       return expirationTimeInSeconds < nowInSeconds;
    }
    // getters and setters omitted
}
```

5. Now, to store an access token, create the `TokenStore` class as presented in the following code. Although we are using `SharedPreferences` to store the access token, it's recommended to keep an access token in memory instead of it being physically stored, because anyone with root access will be able to read the `SharedPreferences` content (when considering a non-rooted device, it would not be a problem to store the access token using `SharedPreferences`). Remember that sometimes, as we increase the security of our application, we compromise the user experience, so that's a hard trade off to deal with and depends on the sensitiveness of the application being developed:

```
public class TokenStore {
    private final SharedPreferences prefs;
    public TokenStore(Context context) {
        prefs = PreferenceManager.getDefaultSharedPreferences(context)
    }
    public void save(AccessToken accessToken) {
        SharedPreferences.Editor editor = prefs.edit();
        editor.putBoolean("authorized", true);
        editor.putString("access_token", accessToken.getValue());
        editor.putString("scope", accessToken.getScope());
        editor.putString("token_type", accessToken.getTokenType());
        editor.putLong("expires_in", accessToken.getExpiresIn());
        editor.putLong("issued_at", accessToken.getIssuedAt());
        editor.commit();
    }
    public AccessToken getToken() {
        AccessToken token = null;
        boolean authorized = prefs.getBoolean("authorized", false);
        if (authorized) {
            token = new AccessToken();
            token.setValue(prefs.getString("access_token", null));
            token.setScope(prefs.getString("scope", ""));
            token.setTokenType(prefs.getString("token_type", "bearer")
            token.setExpiresIn(prefs.getLong("expires_in", -1));
            token.setIssuedAt(prefs.getLong("issued_at", -1));
        }
        return token;
    }
}
```

6. As we will use the Authorization Code grant type, let's create a class that will be in charge of managing the `state` parameter so we can avoid the CSRF attack when receiving an Authorization Code. Create the following class within the `client/oauth2` sub-package:

```
    public class OAuth2StateManager {
        private final SharedPreferences prefs;
        public OAuth2StateManager(Context context) {
            prefs = PreferenceManager.getDefaultSharedPreferences(context)
        }
        public void saveState(String state) {
            SharedPreferences.Editor editor = prefs.edit();
            editor.putString("state", state);
            editor.commit();
        }
        public String getState() {
            return prefs.getString("state", "");
        }
        public boolean isValidState(String state) {
            return this.getState().equals(state);
        }
    }
```

7. To create the authorization URI needed for the Authorization Code grant type, create the following class within the same package as `OAuth2StateManager` (the following class won't compile yet because it references `ClientAPI` that will be created further):

```
    public class AuthorizationRequest {
        public static final String REDIRECT_URI
            = "oauth2://profile/callback";
        public static Uri createAuthorizationUri(String state) {
            return new Uri.Builder()
                .scheme("http")
                .encodedAuthority(ClientAPI.BASE_URL)
                .path("/oauth/authorize")
                .appendQueryParameter("client_id", "clientapp")
                .appendQueryParameter("response_type", "code")
                .appendQueryParameter("redirect_uri", REDIRECT_URI)
                .appendQueryParameter("scope", "read_profile")
                .appendQueryParameter("state", state)
                .build();
        }
    }
```

8. Now create the following class within the `client/oauth2` sub-package, to help us in generating the body content for the token request:

```
public class AccessTokenRequest {
    public static Map<String, String> fromCode(String code) {
        Map<String, String> map = new HashMap<>();
        map.put("code", code);
        map.put("scope", "read_profile");
        map.put("grant_type", "authorization_code");
        map.put("redirect_uri", AuthorizationRequest.REDIRECT_URI);
        return map;
    }
}
```

9. We created all the needed data structures for authorization and token request phases from the Authorization Code grant type. Now we need to create the service that allows the application to interact with OAuth 2.0 endpoints. By using `retrofit2`, we just need to declare an interface as follows (declare the following interface within the `client/oauth2` sub-package):

```
public interface OAuth2API {
    @FormUrlEncoded @POST("oauth/token")
    Call<AccessToken> requestToken(
        @FieldMap Map<String, String> tokenRequest);
}
```

10. Now let's create what's needed to interact with the user profile API. Create the following class inside the `client/profile` sub-package, which declares the data structure that holds the basic user's profile:

```
public class UserProfile {
    private String name;
    private String email;
    public String getName() { return name; }
    public String getEmail() { return email; }
}
```

11. Then create the following interface that allows our application to interact with the user's profile endpoint. This interface must be created in the same package as the `UserProfile` class:

```
public interface UserProfileAPI {
    @GET("api/profile")
    Call<UserProfile> token(
        @Header("Authorization") String accessToken);
}
```

12. We have just declared some clean interfaces to interact with OAuth 2.0 and profile endpoints, but how does `retrofit2` know how to send the requests and manage the results properly? How does `retrofit2` know where the OAuth 2.0 server is located? How do we authenticate our client at the token endpoint? All of these configurations have to be present within a `Retrofit` instance which holds all the required settings. Create the class `RetrofitAPIFactory` within the `client` sub-package, and add the following content:

```
class RetrofitAPIFactory {
    private final Retrofit retrofit;
    RetrofitAPIFactory(String baseUrl,
       OAuth2ClientAuthenticationInterceptor clientAuthentication) {
        retrofit = new Retrofit.Builder()
            .baseUrl("http://" + baseUrl)
            .addConverterFactory(JacksonConverterFactory.create())
            .client(createClient(clientAuthentication))
            .build();
    }
    public Retrofit getRetrofit() { return retrofit; }
    private OkHttpClient createClient(
        OAuth2ClientAuthenticationInterceptor clientAuthentication) {
        OkHttpClient.Builder client = new OkHttpClient.Builder();
        return client.build();
    }
}
```

13. When configuring the `OkHttpClient` object, the previous source code presented the use of `OkHttpClient.Builder` to create an instance of `OkHttpClient` which is an object that will be in charge of making requests to specified APIs. But this client object still doesn't know how to authenticate the client application against the endpoints. To do so, it's necessary to add some interceptors that we will declare in the next steps.

14. So inside the `client/interceptor` sub-package, add the following class to add bearer tokens as an HTTP header when appropriate (import `Response` class from `okhttp3` package):

```
public class BearerTokenHeaderInterceptor implements Interceptor {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request request = chain.request();
        List<String> headers = request.headers("Authorization");
        if (headers.size() > 0) {
            String accessTokenValue = headers.get(0);
            request = request.newBuilder()
                .removeHeader("Authorization")
```

```
                    .addHeader("Authorization", "Bearer " + accessTokenVal
                    .build();
            }
            return chain.proceed(request);
        }
    }
```

15. Now within the same package, add the `OAuth2ClientAuthenticationInterceptor` class as presented in the following code:

```
    public class OAuth2ClientAuthenticationInterceptor implements Intercep
        @Override
        public Response intercept(Chain chain) throws IOException {
            Request request = chain.request();
            Request authenticatedRequest = request.newBuilder()
                .addHeader("Authorization", getEncodedAuthorization())
                .addHeader("Content-Type", "application/x-www-form-urlenco
                .method(request.method(), request.body())
                .build();
            return chain.proceed(authenticatedRequest);
        }
        private String getEncodedAuthorization() {
            String credentials = "clientapp:123456";
            return "Basic " + Base64.encodeToString(
               credentials.getBytes(), Base64.NO_WRAP);
        }
    }
```

16. And to handle error responses, you can create the following interceptor:

```
    public class ErrorInterceptor implements Interceptor {
        @Override
        public Response intercept(Chain chain) throws IOException {
            Request request = chain.request();
            Response response = chain.proceed(request);
            boolean httpError = (response.code() >= 400);
            if (httpError) {
                throw new HttpException(response.code()
                   + ":" + response.message());
            }
            return response;
        }
        public static class HttpException extends RuntimeException {
            public HttpException(String message) {super(message);}
        }
    }
```

17. To start using all of the previously declared interceptors, open the `RetrofitAPIFactory` class and replace the `createClient` method with the following code:

```
private OkHttpClient createClient(
    OAuth2ClientAuthenticationInterceptor clientAuthentication) {
    OkHttpClient.Builder client = new OkHttpClient.Builder();
    client.addInterceptor(new ErrorInterceptor());
    client.addInterceptor(new BearerTokenHeaderInterceptor());
    if (clientAuthentication != null) {
        client.addInterceptor(clientAuthentication);
    }
    return client.build();
}
```

18. Then create the `ClientAPI` class within the same package as `RetrofitAPIFactory` and add the code presented as follows. Notice that `Retrofit` creates proxies for each interface that defines the external services to be accessed by our application:

```
public class ClientAPI {
    public static final String BASE_URL = "10.0.2.2:8080";
    public static UserProfileAPI userProfile() {
        RetrofitAPIFactory api = new RetrofitAPIFactory(BASE_URL, null
        return api.getRetrofit().create(UserProfileAPI.class);
    }
    public static OAuth2API oauth2() {
        RetrofitAPIFactory api = new RetrofitAPIFactory(BASE_URL,
            new OAuth2ClientAuthenticationInterceptor());
        return api.getRetrofit().create(OAuth2API.class);
    }
}
```

19. Now it's time to create the presenter layer, which is comprised by the activity classes, and their respective layouts. The first activity was created by default and was named `MainActivity`. In my case, `MainActivity` class was moved to `presenter` sub-package for better legibility. Open the respective layout (`activity_main.xml`) and add the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://sch
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.packt.com.authcodeapp.presenter.MainActivit
    <Button
        android:id="@+id/profile_button"
        android:text="Get profile"
        android:layout_width="368dp"
        android:layout_height="wrap_content"
        tools:layout_editor_absoluteY="0dp"
        tools:layout_editor_absoluteX="8dp" />
</android.support.constraint.ConstraintLayout>
```

20. Right-click on the app icon (in the Project pane on left side) and select `New/Activity/Empty Activity`. Then define the name of the activity as `ProfileActivity` and click on the Finish button (make sure you select the sub-package `presenter`).
21. Then open the `activity_profile.xml` layout file and add the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/andr
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="example.packt.com.authcodeapp.presenter.ProfileActi
    <TextView
        android:id="@+id/profile_name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    <TextView
        android:layout_below="@+id/profile_name"
        android:id="@+id/profile_email"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</RelativeLayout>
```

22. After defining the layout for `Profile` activity, add the following source code within the `ProfileActivity` class:

```
public class ProfileActivity extends AppCompatActivity {
    private TextView textName;
    private TextView textEmail;
    private TokenStore tokenStore;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_profile);
        tokenStore = new TokenStore(this);
        textName = (TextView) findViewById(R.id.profile_name);
        textEmail = (TextView) findViewById(R.id.profile_email);
        Call<UserProfile> call = ClientAPI
                .userProfile().token(tokenStore.getToken().getValue())
        call.enqueue(new Callback<UserProfile>() {
            @Override
            public void onResponse(Call<UserProfile> call,
                Response<UserProfile> response) {
                UserProfile userProfile = response.body();
                textName.setText(userProfile.getName());
                textEmail.setText(userProfile.getEmail());
            }
            @Override
            public void onFailure(Call<UserProfile> call, Throwable t)
```

```
                    Log.e("ProfileActivity", "Error trying to retrieve use
                }
            });
        }
    }
```

23. The `ProfileActivity` class is in charge of interacting with the profile endpoint which is an OAuth 2.0 protected API. Notice that we are assuming that there is a valid access token stored in the `TokenStore` class (you should validate the access token before using it; you can do it as an exercise for this recipe).
24. As we are using the Authorization Code grant type, our application needs some means of receiving the Authorization Code after the Resource Owner authorizes our application. To allow it to work as the URL callback used for web applications, we will manually create an activity and configure a URL scheme, that when requested, will launch the activity defined through the `AuthorizationCodeActivity` class. Create the `AuthorizationCodeActivity` classinside the `presenter` sub-package and add the following source code:

```
public class AuthorizationCodeActivity extends AppCompatActivity {
    private String code;
    private String state;
    private TokenStore tokenStore;
    private OAuth2StateManager manager;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_profile);
        tokenStore = new TokenStore(this);
        manager = new OAuth2StateManager(this);

        Uri callbackUri = Uri.parse(getIntent().getDataString());
        code = callbackUri.getQueryParameter("code");
        state = callbackUri.getQueryParameter("state");
        // validates state
        if (!manager.isValidState(state)) {
            Toast.makeText(this, "CSRF Attack detected", Toast.LENGTH_
            return;
        }
        Call<AccessToken> accessTokenCall = ClientAPI
                .oauth2()
                .requestToken(AccessTokenRequest.fromCode(code));
        accessTokenCall.enqueue(new Callback<AccessToken>() {
            @Override
            public void onResponse(Call<AccessToken> call, Response<Ac
                AccessToken token = response.body();
                tokenStore.save(token);
                Intent intent = new Intent(AuthorizationCodeActivity.t
                        ProfileActivity.class);
```

```
                    startActivity(intent);
                    finish();
                }
                @Override
                public void onFailure(Call<AccessToken> call, Throwable t)
                    Log.e("AuthorizationCode", "Error retrieving access tc
                }
            });
        }
    }
```

25. The activity we created in the previous step doesn't need a layout. But we still need to register it within the `AndroidManifest.xml` file. Add the following `activity` definition inside the `application` tag within the `AndroidManifest.xml` file:

```
<activity android:name=".presenter.AuthorizationCodeActivity">
<intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:scheme="oauth2"
        android:host="profile"
        android:path="/callback"/>
</intent-filter>
</activity>
```

26. Still in the `AndroidManifest.xml` file, add the following declaration as the first element within the root tag, which is the `manifest` tag:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

27. Finally, make sure that `MainActivity` looks like follows:

```
public class MainActivity extends AppCompatActivity
    implements View.OnClickListener {
    private Button profileButton;
    private TokenStore tokenStore;
    private OAuth2StateManager oauth2StateManager;
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tokenStore = new TokenStore(this);
        oauth2StateManager = new OAuth2StateManager(MainActivity.this)
        profileButton = (Button) findViewById(R.id.profile_button);
        profileButton.setOnClickListener(this);
    }
    public void onClick(View view) {
        AccessToken accessToken = tokenStore.getToken();
        if (accessToken != null && !accessToken.isExpired()) {
            Intent intent = new Intent(this, ProfileActivity.class);
```

```
                    startActivity(intent);
                    return;
                }
                String state = UUID.randomUUID().toString();
                oauth2StateManager.saveState(state);
                Uri authorizationUri = AuthorizationRequest
                        .createAuthorizationUri(state);
                Intent authorizationIntent = new Intent(Intent.ACTION_VIEW);
                authorizationIntent.setFlags(Intent.FLAG_ACTIVITY_NO_HISTORY);
                authorizationIntent.setData(authorizationUri);
                startActivity(authorizationIntent);
            }
        }
```

28. Now our app is ready to run (just make sure that the `server` application is running on the `8080` port).

How it works...

Make sure you have the server running and start the application we have built throughout this recipe. When you start the application, you should see a simple screen with the Get profile button. This button is rendered by the `MainActivity` class that is in charge of opening another activity that the system browser will request from the authorization endpoint of our server application. After the user authorizes our application to access her protected resources, she will be redirected back to `oauth2://profile/callback`, which is a URI scheme.

By defining an `intent-filter` binding the URI scheme to `AuthorizationCodeActivity`, in case of a request for `oauth2://profile/callback`, such registered activity will be launched and it will get the chance to extract the URI parameters which must contain the Authorization Code and the state parameter (if sent to the authorization endpoint).

When receiving a valid Authorization Code and a valid state parameter in the `AuthorizationCodeActivity`, we make a request for the token endpoint and if everything runs well, we save the issued access tokens using the `TokenStore` class, and proceed to `ProfileActivity` so the profile can be presented to the user. You might have noticed that we aren't validating if the access token has not expired on `ProfileActivity`, although it's recommended to validate it before using the access token.

There's more...

Notice that we are configuring the client credentials directly on the source code, as you can see inside `OAuth2ClientAuthenticationInterceptor`. This must be avoided through another strategy, such as using the **key chain** feature or by dynamically registering the client storing the credentials in the memory instead of using local storage. This recipe won't get into the details of how to use a key chain as it is too intrinsic to each mobile platform. Even so, you can explore more about dynamically registering the client in the *Using dynamic client registration with mobile applications* recipe.

A good practice that is worth mentioning here is that we are using system browsers, which is highly recommended as a safer approach when redirecting users to the OAuth 2.0 Provider. As per RFC 8252, using a system browser is better than the web using views, because applications hosting embedded user agents can capture user credentials as well as session cookies. This will be bad because this app would be recognized by the server as the user by itself and the app could do anything in the name of the user. Another advantage achieved by using the system browser approach is that, once the user is logged in to the Authorization Server, she will not need to log in again until her session is valid.

See also

- Preparing an Android development environment

Creating an Android OAuth 2.0 client using the Implicit grant type with the system browser

The specification for OAuth 2.0, which is RFC 6749, address native mobile applications with just a small section. It does not states which grant type must be used or not, although it mentions the usage of Authorization Code and Implicit grant type. The only concern when using Implicit grant type is about that a refresh token is not returned requiring the authorization processes once the access token expires. Even though, the most recent specification, *OAuth 2.0 for native apps* (RFC 8252) states that implicit flow isn't recommended for native apps, basically because by using this grant type the client application will not be able to use PKCE, which avoids interception attacks (we will see more about PKCE in the *Protecting an Android client with PKCE* recipe).

Despite these considerations, this recipe still presents you with how to use the Implicit grant type, because depending on your scenario, you might have an application that can use an API which does not compromise user data. **Just make sure you aren't exposing the user's sensitive data**. Bear in mind that this grant type is NOT recommended for most scenarios when implementing native mobile applications.

Getting ready

To run this recipe you need the `server` application running on you environment. This is provided together with this book's source code examples at GitHub, within the `Chapter07`directory. Make sure that this project is running, and to do so, you can just go to the `Chapter07/server` directory and run the `mvn spring-boot:run` command. In addition, you will need Android Studio to create the recipe.

How to do it...

Create a new project in Android Studio which can be named `ImplicitApp`, and perform the following steps (when creating the application, make sure to automatically create an empty activity named `MainActivity`). The complete source code for this recipe can be download at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter07/ImplicitApp.

1. As we need to interact with an API through HTTP requests, let's add `retrofit2` as a dependency for our project. The `retrofit2` is a great library that helps interacting with REST APIs. To add `retrofit2` to our project, open the application `build.gradle` file and add the following content within the `dependencies` declaration:

```
compile 'com.squareup.retrofit2:retrofit:2.3.0'
compile 'com.squareup.retrofit2:converter-jackson:2.3.0'
compile 'com.squareup.okhttp3:logging-interceptor:3.9.0'
```

2. In the same `build.gradle` file, at the end of android object declaration, add the following content:

```
packagingOptions {
    exclude 'META-INF/LICENSE'
}
```

3. Now let's create the package structure for our project so we can have a well organized application. Within the project's base package, create `client` and `presenter` packages. Within the `client` package, create `interceptor`, `oauth2`, `utils`, and `profile` sub-packages (we will refer to these sub-packages later on, in the next steps).
4. Before creating the activities for our application, let's create all the infrastructure needed to interact with the server APIs. Within `client/oauth2`, create the `AccessToken` and `TokenStore` classes with the same content that was presented in the *Android OAuth 2.0 client using an Authorization Code with the System browser* recipe.

5. Within the same package, create the `AuthorizationRequest` class with the following content:

```
public class AuthorizationRequest {
    public static final String REDIRECT_URI
            = "oauth2://profile/callback";
    public static Uri createAuthorizationURI(String state) {
        return new Uri.Builder()
           .scheme("http")
           .encodedAuthority(ClientAPI.BASE_URL)
           .path("/oauth/authorize")
           .appendQueryParameter("client_id", "clientapp")
           .appendQueryParameter("response_type", "token")
           .appendQueryParameter("redirect_uri", REDIRECT_URI)
           .appendQueryParameter("scope", "read_profile")
           .appendQueryParameter("state", state)
           .build();
    }
}
```

6. And to avoid CSRF attacks, create the `OAuth2StateManager` class to manage the `state` parameter, inside the `client/oauth2` sub-package:

```
public class OAuth2StateManager {
    private final SharedPreferences prefs;
    public OAuth2StateManager(Context context) {
        prefs = PreferenceManager.getDefaultSharedPreferences(context)
    }
    public void saveState(String state) {
        SharedPreferences.Editor editor = prefs.edit();
        editor.putString("state", state);
        editor.commit();
    }
    public String getState() {
        return prefs.getString("state", "");
    }
    public boolean isValidState(String state) {
        return this.getState().equals(state);
    }
}
```

7. Now, let's create the interceptors as explained in the *Android OAuth 2.0 client using an Authorization Code with the System browser* recipe. Create the `BearerTokenHeaderInterceptor` class which is in charge of adding the authorization header with access tokens to access OAuth 2.0 protected resources. This class should be created inside the `client/interceptor` sub-package (remember to import both `Request` and `Response` classes from `okhttp3` package):

```
public class BearerTokenHeaderInterceptor implements Interceptor {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request request = chain.request();
        List<String> headers = request.headers("Authorization");
```

```
            if (headers.size() > 0) {
                String accessTokenValue = headers.get(0);
                request = request.newBuilder()
                    .removeHeader("Authorization")
                    .addHeader("Authorization", "Bearer " + accessTokenVal
                    .build();
            }
            return chain.proceed(request);
        }
    }
```

8. And in the same package, create the `ErrorInterceptor` class as follows:

```
    public class ErrorInterceptor implements Interceptor {
        @Override
        public Response intercept(Chain chain) throws IOException {
            Request request = chain.request();
            Response response = chain.proceed(request);
            boolean httpError = (response.code() >= 400);
            if (httpError) {
                throw new HttpException();
            }
            return response;
        }
        public static class HttpException extends RuntimeException {}
    }
```

9. Inside the `client/profile` sub-package, create `UserProfile` and `UserProfileAPI` classes with the same source code from *Android OAuth 2.0 client using an Authorization Code with the System browser* recipe.

10. Inside the `client/utils` sub-package, create the `URIUtils` class to help us extract the parameters from the URI fragment, which need to be handled when using the Implicit grant type:

```
    public class URIUtils {
        @NonNull
        public static Map<String, String> getQueryParameters(String fragme
            String[] queryParams = fragment.split("&");
            Map<String, String> parameters = new HashMap<>();
            for (String keyValue : queryParams) {
                String[] parameter = keyValue.split("=");
                String key = parameter[0];
                String value = parameter[1];
                parameters.put(key, value);
            }
            return parameters;
        }
    }
```

11. Now create the following simplified version of the `ClientAPI` class within

the `client` sub-package:

```
public class ClientAPI {
    public static final String BASE_URL = "10.0.2.2:8080";
    private final Retrofit retrofit;
    private ClientAPI() {
        HttpLoggingInterceptor logging = new HttpLoggingInterceptor();
        logging.setLevel(HttpLoggingInterceptor.Level.BODY);

        OkHttpClient.Builder client = new OkHttpClient.Builder();
        client.addInterceptor(logging);
        client.addInterceptor(new BearerTokenHeaderInterceptor());
        client.addInterceptor(new ErrorInterceptor());
        retrofit = new Retrofit.Builder()
                .baseUrl("http://" + BASE_URL)
                .addConverterFactory(JacksonConverterFactory.create())
                .client(client.build())
                .build();
    }
    public static UserProfileAPI userProfile() {
        ClientAPI api = new ClientAPI();
        return api.retrofit.create(UserProfileAPI.class);
    }
}
```

12. For the `MainActivity` that might be created as the default activity, add the following source code (it won't compile until we finish the whole recipe, so don't worry with any compilation error for now):

```
public class MainActivity extends AppCompatActivity implements View.Or
    private TokenStore tokenStore;
    private OAuth2StateManager oAuth2StateManager;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tokenStore = new TokenStore(this);
        oAuth2StateManager = new OAuth2StateManager(this);
        Button profileButton = (Button) findViewById(R.id.profile_butt
        profileButton.setOnClickListener(this);
    }
    @Override
    public void onClick(View view) {
        AccessToken accessToken = tokenStore.getToken();
        Intent intent;
        if (accessToken != null && !accessToken.isExpired()) {
            intent = new Intent(this, ProfileActivity.class);
        } else {
            String state = UUID.randomUUID().toString();
            oAuth2StateManager.saveState(state);
            Uri authorizationUri = AuthorizationRequest.createAuthoriz
            intent = new Intent(Intent.ACTION_VIEW);
            intent.setFlags(Intent.FLAG_ACTIVITY_NO_HISTORY);
            intent.setData(authorizationUri);
        }
        startActivity(intent);
    }
}
```

13. The application needs to receive the access token implicitly through the URI fragment in the same way this happens for web applications. To handle this redirection callback, create the following class, which will be registered as an Activity and will be bound with a URI scheme:

```
public class RedirectCallbackActivity  extends AppCompatActivity {
    private TokenStore tokenStore;
    private OAuth2StateManager stateManager;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        tokenStore = new TokenStore(this);
        stateManager = new OAuth2StateManager(this);
        Uri callbackUri = Uri.parse(getIntent().getDataString());
        Map<String, String> parameters = URIUtils.getQueryParameters(c
        if (parameters.containsKey("error")) {
            Toast.makeText(this, parameters.get("error_description"),
            return;
        }
        String state = parameters.get("state");
        if (!stateManager.isValidState(state)) {
            Toast.makeText(this, "CSRF Attack detected", Toast.LENGTH_
            return;
        }
        AccessToken accessToken = new AccessToken();
        accessToken.setValue(parameters.get("access_token"));
        accessToken.setExpiresIn(Long.parseLong(parameters.get("expire
        accessToken.setTokenType("bearer");
        tokenStore.save(accessToken);

        Intent intentProfile = new Intent(this, ProfileActivity.class)
        startActivity(intentProfile);
        finish();
    }
}
```

14. The previous class, when receiving an implicit valid access token, saves the access token through the use of the `TokenStore` class and redirects the user to the `ProfileActivity`, which will use the saved access token to request the user's profile and present it on screen. For the `RedirectCallbackActivity` class to be recognized as an activity and to handle the URI scheme registered as a redirection URI at the server side, add the following content into `AndroidManifest.xml` within the `application` tag file:

```
        <activity android:name=".presenter.RedirectCallbackActivity">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.BROWSABLE" />
                <data android:host="profile"
                    android:path="/callback"
                    android:scheme="oauth2" />
            </intent-filter>
        </activity>
```

15. Make sure to add the following declaration within the root tag of the `AndroidManifest.xml` file to allow our application to interact with external APIs over the network.

```
        <uses-permission android:name="android.permission.INTERNET" />
```

16. Now right-click on the app icon in the Project view (present on the left panel) and go to New | Activity | Empty Activity to create the `ProfileActivity` within the sub-package `presenter`.
17. Open the `ProfileActivity` class and add the following code:

```
        private TokenStore tokenStore;
        private TextView textName;
        private TextView textEmail;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_profile);
            tokenStore = new TokenStore(this);
            textName = (TextView) findViewById(R.id.profile_name);
            textEmail = (TextView) findViewById(R.id.profile_email);
            AccessToken accessToken = tokenStore.getToken();

            Call<UserProfile> getUserProfile = ClientAPI.userProfile().token(a
            getUserProfile.enqueue(new Callback<UserProfile>() {
                @Override
                public void onResponse(Call<UserProfile> call, Response<UserPr
                    UserProfile userProfile = response.body();
                    textName.setText(userProfile.getName());
                    textEmail.setText(userProfile.getEmail());
                }
                @Override
                public void onFailure(Call<UserProfile> call, Throwable t) {
                    Log.e("ProfileActivity", "Error trying to retrieve user pr
                }
            });
        }
```

18. And finally, let's create the layouts needed for our application.

19. Replace the content of `activity_main.xml` with the code presented in the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://sch
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.packt.com.implicitapp.presenter.MainActivit
    <Button
        android:id="@+id/profile_button"
        android:text="Get profile"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</android.support.constraint.ConstraintLayout>
```

20. And replace all the source code for `activity_profile.xml` with the following content.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/andr
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="example.packt.com.implicitapp.presenter.ProfileActi
    <TextView
        android:id="@+id/profile_name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    <TextView
        android:layout_below="@+id/profile_name"
        android:id="@+id/profile_email"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</RelativeLayout>
```

How it works...

In the same way we used the URI scheme for an activity to receive an Authorization Code, we also created an activity named `RedirectCallbackActivity` that was in charge of receiving the access token within the URI fragment. The access token, as well as the `state` and `expires_in` parameter, was extracted from the URI fragment by the `URIUtils` class that transforms one single string into a map of strings.

The process starts with the `MainActivity` that provides a button that, when clicked, will try to retrieve an access token from the `TokenStore`, and if there is one valid token, it will send the user directly to `ProfileActivity`. Otherwise, it will start the system browser as presented in the following code, using the authorization URI created by the `AuthorizationRequest` class:

```
Uri authorizationUri = AuthorizationRequest.createAuthorizationURI(state);
intent = new Intent(Intent.ACTION_VIEW);
intent.setFlags(Intent.FLAG_ACTIVITY_NO_HISTORY);
intent.setData(authorizationUri);
```

Notice that although our client is registered with a redirect URI and we are using the state parameter to prevent CSRF attacks, the URI scheme to receive the access token, any other application could be bound to the same URI scheme and start intercepting access tokens from our OAuth 2.0 Provider. When using the Authorization Code grant type we can rely on PKCE, but when using the Implicit grant type we can't use the same technique. So just to emphasize, this grant type should be avoided for most applications.

See also

- Preparing the Android development environment
- Creating an Android OAuth 2.0 client using an Authorization Code with the system browser
- Native client applications section from RFC 6749, available at https://tools.ietf.org/html/rfc6749#section-9
- OAuth Implicit Grant Authorization Flow from RFC 8252, available at https://tools.ietf.org/html/rfc8252#section-8.2

Creating an Android OAuth 2.0 client using the embedded browser

Although there's a lot of vulnerabilities regarding the use of embedded browsers, they are still widely used and sometimes required in specific use cases. Before using embedded browsers, just be aware of the vulnerabilities described in RFC 8252 and make sure that you can't use in-app browser tabs as an alternative. For Android, you can use the Custom tabs feature that is described by the official documentation at https://developer.chrome.com/multidevice/android/customtabs. This recipe presents you with how to use embedded browsers (Android WebView) using the Implicit grant type just for brevity purposes.

As an advice, use `WebView`*judiciously because of issues mentioned in OAuth 2.0 for native apps specification (RFC 8252).*

Remember to use the Authorization Code grant type if you are developing a production application.

Getting ready

To run this recipe you need the `server` application running on you environment. This is provided together with the book's source code examples at GitHub, within the `Chapter07` directory. Make sure that this project is running, and to do so, you can just go to the `Chapter07/server` directory and run the `mvn spring-boot:run` command. In addition, you will need Android Studio to create the recipe and have to use minimum SDK version by 21.

How to do it...

Create a new project in Android Studio which can be named `EmbeddedApp`, and perform the following steps (when creating the application, make sure to automatically create an empty activity named `MainActivity`). The complete source code for this recipe can be download at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter07/EmbeddedApp.

1. As we need to interact with an API through HTTP requests, let's add `retrofit2` as a dependency for our project. To add `retrofit2` to our project, open the application `build.gradle` file and add the following content within the dependencies declaration:

```
compile 'com.squareup.retrofit2:retrofit:2.3.0'
compile 'com.squareup.retrofit2:converter-jackson:2.3.0'
compile 'com.squareup.okhttp3:logging-interceptor:3.9.0'
```

2. In the same `build.gradle` file, at the end of android object declaration, add the following content:

```
packagingOptions {
     exclude 'META-INF/LICENSE'
 }
```

3. Now let's create the package structure for our project so we can have a well organized application. Within the project's base package, create `client` and `presenter` packages. Within the `client` package, create `interceptor`, `oauth2`, and `profile` sub-packages (we will refer to these sub-packages later on in the next steps). I recommend you to move `MainActivity` class to `presenter` package (use refactor tools from Android Studio).
4. Before creating the activities for our application, let's create all the infrastructure needed to interact with the server APIs. Within `client/oauth2`, create the `AccessToken` and `TokenStore` classes with the same content that was presented in the *Creating an Android OAuth 2.0 client using an Authorization Code with the System browser* recipe.
5. Within the same package, create the `AuthorizationRequest` class with the

same content presented in the source code in the *Creating an Android OAuth 2.0 client using the Implicit grant type with a system browser* recipe (but use `response_type` as the token because both are using the Implicit grant type).

6. Inside the `client/oauth2` sub-package, also copy the `URIUtils` and `OAuth2StateManager` classes from the *Android OAuth 2.0 client using an Authorization Code with the system browser* recipe.
7. Create the interceptors `BearerTokenHeaderInterceptor` and `ErrorInterceptor` inside the `client/interceptor` sub-package and add the same content from the *Android OAuth 2.0 client using an Authorization Code with the system browser* recipe.
8. To illustrate the use of the external API, copy the classes `UserProfile` and `UserProfileAPI` from the `AuthCodeApp` project created in the *Android OAuth 2.0 client using an Authorization Code with the system browser* recipe, into the `client/profile` sub-package.
9. And in the client sub-package, create the `ClientAPI` class as follows:

```
public class ClientAPI {
     private final Retrofit retrofit;
     public static final String BASE_URL = "10.0.2.2:8080";
     private ClientAPI() {
         HttpLoggingInterceptor logging = new HttpLoggingInterceptor()
         logging.setLevel(HttpLoggingInterceptor.Level.BODY);
         OkHttpClient.Builder client = new OkHttpClient.Builder();
         client.addInterceptor(logging);
         client.addInterceptor(new ErrorInterceptor());
         client.addInterceptor(new BearerTokenHeaderInterceptor());
         retrofit = new Retrofit.Builder()
                 .baseUrl("http://" + BASE_URL)
                 .addConverterFactory(JacksonConverterFactory.create()
                 .client(client.build())
                 .build();
     }
     public static UserProfileAPI userProfile() {
         ClientAPI api = new ClientAPI();
         return api.retrofit.create(UserProfileAPI.class);
     }
 }
```

10. Now, it's time to create the `Activities` and their corresponding layouts when needed. Open the `MainActivity` class and replace the content with what follows:

```
public class MainActivity extends AppCompatActivity
     implements View.OnClickListener {
     private TokenStore tokenStore;
     private Button mainButton;
     @Override
     protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_main);
         tokenStore = new TokenStore(this);
         mainButton = (Button) findViewById(R.id.main_button);
         mainButton.setOnClickListener(this);
     }
     @Override
     public void onClick(View view) {
         if (view == mainButton) {
             Intent intent;
             AccessToken accessToken = tokenStore.getToken();
             if (accessToken != null && !accessToken.isExpired()) {
                 intent = new Intent(this, ProfileActivity.class);
             } else {
                 intent = new Intent(this, AuthorizationActivity.class
             }
             startActivity(intent);
         }
     }
 }
```

11. Open the `activity_main.xml` layout and replace the content with the XML presented in the following code:

```
    <?xml version="1.0" encoding="utf-8"?>
     <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/anc
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context="example.packt.com.embeddedapp.presenter.MainActivi
        <Button
            android:id="@+id/main_button"
            android:text="Authorize to read profile"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
     </RelativeLayout>
```

12. The `MainActivity` class, after checking for an existing and valid access token, directs the user to the `ProfileActivity` or the `AuthorizationActivity`. If there isn't a valid access token, the user will be sent to `AuthorizationActivity`, which will present a `WebView` pointing to the OAuth Provider's authorization endpoint. But before creating the `AuthorizationActivity`, let's create the activity that will be registered with the URI scheme intent, to allow our application to receive the access token in response to user's authorization. Create the `RedirectUriActivity`

class inside the `presenter` sub-package as follows:

```
public class RedirectUriActivity extends AppCompatActivity {
     private TokenStore tokenStore;
     private OAuth2StateManager stateManager;
     @Override
     protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         tokenStore = new TokenStore(this);
         stateManager = new OAuth2StateManager(this);

         Uri callbackUri = Uri.parse(getIntent().getDataString());
         Map<String, String> parameters = URIUtils.getQueryParameters(
            callbackUri.getFragment());
         if (parameters.containsKey("error")) {
             Toast.makeText(this, parameters.get("error_description"),
                Toast.LENGTH_SHORT).show();
             return;
         }

         String state = parameters.get("state");
         if (!stateManager.isValidState(state)) {
             Toast.makeText(this, "CSRF Attack detected", Toast.LENGTH
             return;
         }
         AccessToken accessToken = new AccessToken();
         accessToken.setValue(parameters.get("access_token"));
         accessToken.setExpiresIn(Long.parseLong(parameters.get("expir
         accessToken.setScope(parameters.get("scope"));
         accessToken.setTokenType("bearer");
         tokenStore.save(accessToken);

         Intent intentProfile = new Intent(this, ProfileActivity.class
         startActivity(intentProfile);
         finish();
     }

 }
```

13. Go to File | New | Activity | Empty Activity and create the `AuthorizationActivity` within the `presenter` sub-package.
14. Then, open the `activity_authorization.xml` file and replace the content with what follows (check the base package of your project, which might be different from `example.packt.com.embeddedapp`):

```
<?xml version="1.0" encoding="utf-8"?>
 <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/and
     xmlns:app="http://schemas.android.com/apk/res-auto"
     xmlns:tools="http://schemas.android.com/tools"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     tools:context="example.packt.com.embeddedapp.presenter.Authorizat
     <WebView
```

```
            android:id="@+id/authorization_webview"
            android:layout_width="match_parent"
            android:layout_height="match_parent">
        </WebView>
    </RelativeLayout>
```

15. Now open the `AuthorizationActivity` class and replace the whole content with the following code:

```
public class AuthorizationActivity extends AppCompatActivity {
    private WebView webView;
    private OAuth2StateManager oauth2StateManager;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_authorization);
        webView = (WebView) findViewById(R.id.authorization_webview);
        oauth2StateManager = new OAuth2StateManager(this);
        String state = UUID.randomUUID().toString();
        Uri authorizationUri = AuthorizationRequest.createAuthorizatic
        oauth2StateManager.saveState(state);
        webView.setWebViewClient(new WebViewClient() {
            public boolean shouldOverrideUrlLoading(WebView view, Stri
                return urlLoading(view, url);
            }
            public boolean shouldOverrideUrlLoading(WebView view, WebF
                String url = request.getUrl().toString();
                return urlLoading(view, url);
            }
            private boolean urlLoading(WebView view, String url) {
                if (url.contains("oauth2://profile/callback")) {
                    Intent intent = new Intent(
                            AuthorizationActivity.this, RedirectUriAct
                    intent.setData(Uri.parse(url));
                    startActivity(intent);
                    finish();
                }
                return false;
            }
        });
        webView.loadUrl(authorizationUri.toString());
    }
}
```

16. Then, to create the profile view, go to File | New | Activity | Empty Activity and create `ProfileActivity` within the same package as `AuthorizationActivity`.

17. Open the `activity_profile.xml` file and replace everything with the following XML content:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/anc
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.packt.com.embeddedapp.presenter.ProfileAct
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <TextView
            android:id="@+id/profile_username"
            android:text="user name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
        <TextView
            android:id="@+id/profile_email"
            android:text="email"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</RelativeLayout>
```

18. Then open the `ProfileActivity` class and add the source code presented in the following code:

```
public class ProfileActivity extends AppCompatActivity {
     private TextView usernameText;
     private TextView emailText;
     private TokenStore tokenStore;
     @Override
     protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_profile);
         tokenStore = new TokenStore(this);
         AccessToken accessToken = tokenStore.getToken();
         usernameText = (TextView) findViewById(R.id.profile_username)
         emailText = (TextView) findViewById(R.id.profile_email);
         Call<UserProfile> profileCallback = ClientAPI.userProfile().t
         profileCallback.enqueue(new Callback<UserProfile>() {
             @Override
             public void onResponse(Call<UserProfile> call, Response<U
                 UserProfile userProfile = response.body();
                 usernameText.setText(userProfile.getName());
                 emailText.setText(userProfile.getEmail());
             }
             @Override
             public void onFailure(Call<UserProfile> call, Throwable t
                 Toast.makeText(ProfileActivity.this, "Error retrievir
                     Toast.LENGTH_SHORT).show();
             }
         });
     }
 }
```

19. Now open the `AndroidManifest.xml` file and add the following permission:

```
        <uses-permission android:name="android.permission.INTERNET" />
```

20. Register the URI scheme for `RedirectUriActivity`, by adding the following `activity` section inside the application tag:

```
        <activity android:name=".presenter.RedirectUriActivity">
         <intent-filter>
             <action android:name="android.intent.action.VIEW"/>
             <category android:name="android.intent.category.DEFAULT"/>
             <category android:name="android.intent.category.BROWSABLE"/>
             <data android:scheme="oauth2"
                 android:host="profile"
                 android:path="/callback"/>
         </intent-filter>
         </activity>
```

How it works...

This recipe looks pretty similar to the *Creating an Android OAuth 2.0 client using the Implicit grant type with the system browser* recipe. But rather than using a system browser, this recipe presents you with how you can use the web view component from Android so that the user can interact with the Authorization Server to grant permissions to our application.

Although it presents the user with a seamless navigation, it also allows for some vulnerabilities that might compromise the user's credentials and session cookies. Despite the advantage of a seamless navigation, a `WebView` cannot save the user's session. So whenever the user is redirected to the `WebView`, she will need to authenticate herself again, thereby compromising user experience.

See also

- Preparing the Android development environment
- Creating an Android OAuth 2.0 client using an Authorization Code with the system browser
- Creating an Android OAuth 2.0 client using the Implicit grant type with the system browser

Using the Password grant type for client apps provided by the OAuth 2 server

Sometimes, we are developing native mobile applications that belong to the same solution provided by the server application. Regarding this scenario, users will present their credentials in any case, and these credentials will be the same as those used to authenticate the user at the server side. When faced with such a scenario, instead of storing the user's credentials, our application can exchange them for an access token at the server side (which must be an OAuth 2.0 Provider). Although the access token is also sensitive, it can be maintained in memory or by using some kind of strategy such as a key chain. Even so, access tokens are easily manageable and can have a short life. This recipe presents you with how you can use the Resource Owner Password Credentials grant type to allow the application to exchange user credentials for an OAuth 2.0 access token.

Getting ready

To run this recipe you need the `server` application running on your environment. This is provided together with the book's source code examples at GitHub, within the `Chapter07` directory. Make sure that this project is running, and to do so, you can just go to the `Chapter07/server` directory and run the `mvn spring-boot:run` command. In addition, you will need Android Studio to create the recipe.

How to do it...

Create a new project in Android Studio which can be named `ResourceOwnerPassword` and perform the following steps (when creating the application, make sure to automatically create an empty activity named `MainActivity`). The complete source code for this recipe can be download at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter07/ResourceOwnerPassword.

1. As we need to interact with an API through HTTP requests, let's add `retrofit2` as a dependency for our project. To add `retrofit2` to our project, open the application `build.gradle` file and add the following content within the `dependencies` declaration:

```
compile 'com.squareup.retrofit2:retrofit:2.3.0'
compile 'com.squareup.retrofit2:converter-jackson:2.3.0'
compile 'com.squareup.okhttp3:logging-interceptor:3.9.0'
```

2. In the same `build.gradle` file, at the end of android object declaration, add the following content:

```
packagingOptions {
    exclude 'META-INF/LICENSE'
}
```

3. Now let's create the package structure for our project so we can have a well organized application. Within the project's base package, create `client`, `login`, and `presenter` packages. Within the `client` package, create `interceptor`, `oauth2`, and `profile` sub-packages (we will refer to these sub-packages later, in the next steps).
4. Before creating the activities for our application, let's create all the infrastructure needed to interact with the server APIs. Within `client/oauth2` , create the classes `AccessToken` and `TokenStore` with the same content that was presented in the *Creating an Android OAuth 2.0 client using an Authorization Code with the system browser* recipe.
5. Within the same sub-package, create the class `PasswordAccessTokenRequest`, with the following content:

```
public class PasswordAccessTokenRequest {
    public static Map<String, String> from(
        String username, String password) {
        Map<String, String> map = new HashMap<>();
        map.put("scope", "read_profile");
        map.put("grant_type", "password");
        map.put("username", username);
        map.put("password", password);
        return map;
    }
}
```

6. Then declare the `OAuth2API` interface as presented in the following code (preferably in the same package as `PasswordAccessTokenRequest`):

```
public interface OAuth2API {
    @FormUrlEncoded @POST("oauth/token")
    Call<AccessToken> token(@FieldMap Map<String, String> tokenRequest
}
```

7. Inside the `profile` sub-package, create the following classes (create the respective getters for the `UserProfile` class):

```
public class UserProfile {
    private String name;
    private String email;
    // getters omitted for brevity
}
```

8. And create the `UserProfileAPI` interface as follows:

```
public interface UserProfileAPI {
    @GET("api/profile")
    Call<UserProfile> token(@Header("Authorization") String accessToke
}
```

9. Inside the `interceptor` sub-package, copy `BearerTokenHeaderInterceptor` and `ErrorInterceptor` from the `AuthCodeApp` project, as presented in the *Creating an Android OAuth 2.0 client using an Authorization Code with the system browser* recipe.
10. Then, within the same package, create the `OAuth2ClientAuthenticationInterceptor` class as follows:

```
public class OAuth2ClientAuthenticationInterceptor implements Intercep
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request request = chain.request();
        Request authenticatedRequest = request.newBuilder()
            .addHeader("Authorization", getEncodedAuthorization())
```

```
                .addHeader("Content-Type", "application/x-www-form-urlencc
                .method(request.method(), request.body())
                .build();
            return chain.proceed(authenticatedRequest);
        }
        private String getEncodedAuthorization() {
            return "Basic " + Base64.encodeToString(
              "clientapp:123456".getBytes(), Base64.NO_WRAP);
        }
    }
```

11. To finish the REST client configuration, create the `ClientAPI` class within the sub-package `client` as presented in the following code:

```
    public class ClientAPI {
        public static final String BASE_URL = "10.0.2.2:8080";
        private final Retrofit retrofit;
        public static UserProfileAPI userProfile() {
            ClientAPI api = new ClientAPI(null);
            return api.retrofit.create(UserProfileAPI.class);
        }
        public static OAuth2API oauth2() {
            ClientAPI api = new ClientAPI(new OAuth2ClientAuthenticationIr
            return api.retrofit.create(OAuth2API.class);
        }
        private ClientAPI(OAuth2ClientAuthenticationInterceptor basicAuthe
            HttpLoggingInterceptor logging = new HttpLoggingInterceptor();
            logging.setLevel(HttpLoggingInterceptor.Level.BODY);
            OkHttpClient.Builder client = new OkHttpClient.Builder();
            client.addInterceptor(logging);
            client.addInterceptor(new ErrorInterceptor());
            client.addInterceptor(new BearerTokenHeaderInterceptor());
            if (basicAuthentication != null) {
                client.addInterceptor(basicAuthentication);
            }
            retrofit = new Retrofit.Builder()
                .baseUrl("http://" + BASE_URL)
                .addConverterFactory(JacksonConverterFactory.create())
                .client(client.build())
                .build();
        }
    }
```

12. This recipe is presenting an application that relies on user authentication. To manage user authentication, we will create some classes to emulate an authentication process as well as to identify if the user has logged in through the `AuthenticationManager` class. Create the following class inside the `login` sub-package:

```
    public class AuthenticationManager {
        private final SharedPreferences sharedPreferences;
        public AuthenticationManager(Context context) {
            sharedPreferences = PreferenceManager
                .getDefaultSharedPreferences(context.getApplicationContext
        }
        public void authenticate() {
            SharedPreferences.Editor editor = sharedPreferences.edit();
            editor.putBoolean("authenticated", true);
            editor.commit();
        }
        public boolean isAuthenticated() {
            return sharedPreferences.getBoolean("authenticated", false);
        }
    }
```

13. Then, to emulate the user credentials validation, create the `LoginService` class as presented in the following code (this should be created inside the `login` sub-package):

```
    public class LoginService {
        public void loadUser(String login, String password, Callback callb
            if ("adolfo".equals(login) && "123".equals(password)) {
                User user = new User(login, password);
                callback.onSuccess(user);
            } else {
                callback.onFailed("user or password invalid");
            }
        }
        public interface Callback {
            void onSuccess(User user);
            void onFailed(String message);
        }
    }
```

14. And to represent the logged user, create the following `User` class inside the same package as `LoginService`:

```
    public class User {
        private String username;
        private String password;
        public User(String username, String password) {
            this.username = username;
            this.password = password;
        }
        public String getUsername() {return username;}
        public String getPassword() {return password;}
    }
```

15. Now, as our application will use the network to send requests to our `server` application, add the following permissions within the `AndroidManifest.xml` file. The following declaration must be added as the first element under the `manifest` section:

```
<uses-permission android:name="android.permission.INTERNET" />
```

16. As a native mobile application that presents a user's profile, this project needs respective views to allow the user to interact with the application. At this moment, let's first create the activity that will present the user's profile data (the main activity should have been created at the moment of the project's creation). Go to File | New | Activity | Empty Activity and define the name of the activity as `DashboardActivity`.
17. Now, open the `activity_dashboard.xml` file and replace the whole content with what follows:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/andr
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="example.packt.com.resourceownerpassword.presenter.D
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <ListView
            android:id="@+id/dashboard_entries"
            android:layout_width="match_parent"
            android:layout_height="match_parent">
        </ListView>
        <Button
            android:id="@+id/profile_button"
            android:text="Get user profile"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
        <TextView
            android:id="@+id/profile_username"
            android:text="name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
        <TextView
            android:id="@+id/profile_email"
            android:text="email"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</RelativeLayout>
```

18. Then open the `DashboardActivity` class and add the following attributes:

```
private TokenStore tokenStore;
private TextView usernameText, emailText;
```

19. Start implementing the interface `View.OnClickListener`, and replace the

content of the `onCreate` method, with the following snippet of code:

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_dashboard);
usernameText = (TextView) findViewById(R.id.profile_username);
emailText = (TextView) findViewById(R.id.profile_email);
tokenStore = new TokenStore(this);

if (new AuthenticationManager(this).isAuthenticated()) {
    ListView listView = (ListView) findViewById(R.id.dashboard_entries
    listView.setAdapter(new ArrayAdapter<>(
        this, android.R.layout.simple_list_item_1,
        new String[] {"Entry 1"}));

    Button profileButton = (Button) findViewById(R.id.profile_button);
    profileButton.setOnClickListener(this);
} else {
    Intent loginIntent = new Intent(this, MainActivity.class);
    startActivity(loginIntent);
    finish();
}
```

20. Then add the snippet code presented as follows, within the `onClick` method implemented for the `View.OnClickListener` interface:

```
AccessToken accessToken = tokenStore.getToken();
if (accessToken != null && !accessToken.isExpired()) {
    Call<UserProfile> call = ClientAPI.userProfile()
        .token(accessToken.getValue());
    call.enqueue(new Callback<UserProfile>() {
        @Override
        public void onResponse(Call<UserProfile> call, Response<UserPr
            UserProfile profile = response.body();
            usernameText.setText(profile.getName());
            emailText.setText(profile.getEmail());
        }
        @Override
        public void onFailure(Call<UserProfile> call, Throwable t) {
            Log.e("DashboardActivity", "Error reading user profile dat
        }
    });
}
```

21. To perform the authentication mechanism, this recipe recommends the creation of mediator activity which we will call `AuthorizationActivity`. Create it as a simple class extending `AppCompatActivity` as presented in the following code (remember to create this class inside the `presenter` sub-package to improve the readability of our project):

```
public class AuthorizationActivity extends AppCompatActivity {
    private TokenStore tokenStore;
```

```
        private AuthenticationManager authenticationManager;
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            tokenStore = new TokenStore(this);
            authenticationManager = new AuthenticationManager(this);
            String username = getIntent().getStringExtra("username");
            String password = getIntent().getStringExtra("password");
            if (authenticationManager.isAuthenticated()) {
                Call<AccessToken> call = ClientAPI.oauth2().token(
                    PasswordAccessTokenRequest.from(username, password));
                call.enqueue(new Callback<AccessToken>() {
                    @Override
                    public void onResponse(Call<AccessToken> call,
                      Response<AccessToken> response) {
                        AccessToken accessToken = response.body();
                        tokenStore.save(accessToken);
                        Intent intent = new Intent(
                            AuthorizationActivity.this, DashboardActivity.
                        startActivity(intent);
                        finish();
                    }
                    @Override
                    public void onFailure(Call<AccessToken> call, Throwabl
                        Log.e("AuthorizationActivity", "could not retrieve
                    }
                });
            }
        }
    }
```

22. The previous declared class doesn't need a layout. But it still needs to be declared inside the `AndroidManifest.xml` file. Add the following declaration inside the `application` section of the `AndroidManifest.xml` file:

```
    <activity android:name=".presenter.AuthorizationActivity"></activity>
```

23. The main activity of the `ResourceOwnerPassword` application was created by default and named `MainActivity`. Open the respective layout file (`activity_main.xml`) and replace the whole content with what follows (pay attention to the activity package defined in the following `tools:context` attribute).

```
    <?xml version="1.0" encoding="utf-8"?>
    <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/andr
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context="example.packt.com.resourceownerpassword.presenter.M
        <LinearLayout
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true"
            android:orientation="vertical"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
            <EditText
                android:id="@+id/main_username"
                android:hint="Username"
                android:layout_width="match_parent"
                android:layout_height="wrap_content" />
            <EditText
                android:inputType="textPassword"
                android:id="@+id/main_password"
                android:hint="Password"
                android:layout_width="match_parent"
                android:layout_height="wrap_content" />
            <Button
                android:id="@+id/main_login_button"
                android:text="Login"
                android:layout_width="match_parent"
                android:layout_height="wrap_content" />
        </LinearLayout>
    </RelativeLayout>
```

24. Then open the `MainActivity` class and make sure that it looks like the source code presented:

```
public class MainActivity extends AppCompatActivity implements View.Or
    private LoginService loginService;
    private TokenStore tokenStore;
    private AuthenticationManager authenticationManager;
    private TextView usernameText;
    private TextView passwordText;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        loginService = new LoginService();
        tokenStore = new TokenStore(this);
        authenticationManager = new AuthenticationManager(this);
        Button loginButton = (Button) findViewById(R.id.main_login_but
        loginButton.setOnClickListener(this);
    }
    @Override
    public void onClick(View view) {
    }
}
```

25. Now add the following snippet of code within the `onClick` method from `MainActivity` so the user can be authenticated by the application:

```
usernameText = (TextView) findViewById(R.id.main_username);
passwordText = (TextView) findViewById(R.id.main_password);
String username = usernameText.getText().toString();
```

```
        String password = passwordText.getText().toString();

        loginService.loadUser(username, password, new LoginService.Callback()
            @Override
            public void onSuccess(User user) {
                authenticationManager.authenticate();
                AccessToken accessToken = tokenStore.getToken();
                Intent intent;
                if (accessToken != null && !accessToken.isExpired()) {
                    intent = new Intent(MainActivity.this, DashboardActivity.c
                } else {
                    intent = new Intent(MainActivity.this, AuthorizationActivi
                    intent.putExtra("username", user.getUsername());
                    intent.putExtra("password", user.getPassword());
                }
                startActivity(intent);
            }
            @Override
            public void onFailed(String message) {
                Toast.makeText(MainActivity.this, message, Toast.LENGTH_SHORT)
            }
        });
```

How it works...

This recipe creates an application that asks for user credentials at the first moment the user launches the application. Once the user provides a valid username and password they will be redirected to the `AuthorizationActivity` so the application can start requesting an access token through the use of the Resource Owner Password Credentials grant type. Notice that the user credentials are validated by the `LoginService` that should perform the authentication remotely instead of checking for hard coded credentials.

Instead of storing the user's credentials, we exchange them for an access token, thus protecting the username and password which would be harder to rotate. By using an access token, it's easy for our server application to just invalidate the access token by itself and it will not affect the final user (who is the Resource Owner).

There's more...

There's an interesting point on `DashboardActivity` related to the user's profile retrieval. When the user clicks on the Get User Profile button, the application validates if the access token is valid before sending a request to the profile endpoint. But when the access token isn't valid we simply do nothing. It would be nice if our client application could request a new access token by using a refresh token. Instead of a refresh token, we could simply send the user to the `MainActivity` that requires the user to log in again (you can do it as an exercise).

The problem that we have here is where to store the refresh token or even the access token. If the access token is stolen, it will be valid for a defined period of time, but the problem with refresh tokens is that they are created to live longer than access tokens. If you are considering using a refresh token, try first to tighten the expiration time for the refresh token, or even better, try using something similar to key chain storage.

See also

- Preparing the Android development environment
- Creating an Android OAuth 2.0 client using an Authorization Code with the system browser

Protecting an Android client with PKCE

When implementing OAuth 2.0 native mobile applications, it's required that you handle the redirection URI when using the Authorization Code or the Implicit grant types. Handing the callback started by the OAuth 2.0 Provider can be achieved by registering a URI scheme strategy. But how can we protect the Authorization Code to be delivered to the right client application? If another client application (a malicious one) registers an activity to listen to the same URI scheme registered for our application, the operational system (in this case Android), will prompt the user with the means to select which application to use. If the user selects the bad one, the Authorization Code will be delivered to the bad application which can request an access token improperly. The PKCE is defined by RFC 7636 just to address this kind of problem and this recipe will help you implement a native application that relies on PKCE to be protected against Authorization Code interception attacks.

Getting ready

To run this recipe you need the `server` application running on you environment. This is provided together with the book's source code examples at GitHub, within the `Chapter07` directory. Make sure that this project is running, and to do so, you can just go to the `Chapter07/server` directory and run the `mvn spring-boot:run` command.

For this recipe you also need to have created a simple application using the Authorization Code grant type, so we can add just what's needed for the PKCE feature. Although, instead of creating the project from scratch, you can do one of the following options. The first option is to copy all the source code (application `build.gradle` file, classes, layouts and `AndroidManifest.xml`) from the `AuthCodeApp` created in the *Creating an Android OAuth 2.0 client using an Authorization Code with the system browser* recipe and create a new Android project with these copied files. The last and second option, is to just change the required classes that will be presented throughout this recipe directly in the `AuthCodeApp` project (I personally recommend the last option).

How to do it...

Once you have a new project created or are working with the `AuthCodeApp` project that was created in the *Creating an Android OAuth 2.0 client using an Authorization Code with the system browser* recipe, you just have to perform the following steps to start adding support for PKCE to avoid Authorization Code interception. The complete source code for this recipe can also be download at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter07/.

1. First of all, create the class `PkceManager` that will be in charge of generating the code challenge and to store the code verifier which must be bound to the code challenge. This class should be created inside the sub-package `client/oauth2`:

```
public class PkceManager {
    private final SharedPreferences preferences;
    public PkceManager(Context context) {
        preferences = PreferenceManager.getDefaultSharedPreferences(cc
    }
    public String createChallenge() {
        String codeVerifier = UUID.randomUUID().toString();
        try {
            MessageDigest messageDigest = MessageDigest.getInstance("S
            byte[] signedContent = messageDigest.digest(codeVerifier.g
            StringBuilder challenge = new StringBuilder();
            for (byte signedByte : signedContent) {
                challenge.append(String.format("%02X", signedByte));
            }
            SharedPreferences.Editor editor = preferences.edit();
            editor.putString("code_verifier", codeVerifier);
            editor.commit();
            return challenge.toString().toLowerCase();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
    public String getCodeVerifier() {
        String codeVerifier = preferences.getString("code_verifier", "
        SharedPreferences.Editor editor = preferences.edit();
        editor.clear();
        return codeVerifier;
    }
}
```

2. As this application is using the Authorization Code grant type, we need to create the authorization URI which must convey the `code_challenge` and `code_challenge_method` parameters. This will be stored by the authorization server, to validate it later against the `code_verifier` when requesting for an access token. Open the class `AuthorizationRequest` and make sure that this class looks as follows:

```
public class AuthorizationRequest {
    public static final String REDIRECT_URI
            = "oauth2://profile/callback";
    private final PkceManager pixyManager;
    public AuthorizationRequest(PkceManager pixyManager) {
        this.pixyManager = pixyManager;
    }
    public Uri createAuthorizationUri(String state) {
        return new Uri.Builder()
          .scheme("http")
          .encodedAuthority(ClientAPI.BASE_URL)
          .path("/oauth/authorize")
          .appendQueryParameter("client_id", "clientapp")
          .appendQueryParameter("response_type", "code")
          .appendQueryParameter("redirect_uri", REDIRECT_URI)
          .appendQueryParameter("scope", "read_profile")
          .appendQueryParameter("state", state)
          .appendQueryParameter("code_challenge", pixyManager.createCh
          .appendQueryParameter("code_challenge_method", "S256")
          .build();
    }
}
```

3. As explained in the previous step, when requesting an access token against the Authorization Server, the client is required to send the `code_verifier` from the parameter which is related to the `code_challenge` sent before in the authorization phase. To do so, open the `AccessTokenRequest` class and make sure it looks like the following:

```
public class AccessTokenRequest {
    private final PkceManager pixyManager;
    public AccessTokenRequest(PkceManager pixyManager) {
        this.pixyManager = pixyManager;
    }
    public Map<String, String> from(String code) {
        Map<String, String> map = new HashMap<>();
        map.put("code", code);
        map.put("code_verifier", pixyManager.getCodeVerifier());
        map.put("scope", "read_profile");
        map.put("grant_type", "authorization_code");
        map.put("redirect_uri", "oauth2://profile/callback");
        return map;
    }
    }
```

4. The authorization request is created by the MainActivity. So to start using the new version of the `AuthorizationRequest` class, open the `MainActivity` and add the following class attribute:

```
    private AuthorizationRequest authorizationRequest;
```

5. Then, inside the `onCreate` method, create an instance of `AuthorizationRequest` using an instance of `PkceManager` as presented in the following code:

```
    authorizationRequest = new AuthorizationRequest(new PkceManager(this))
```

6. Now, within the `onClick` method, look at the source code that creates the `authorizationUri` variable, and instead of creating this `Uri` by using a static factory method, replace it with what is presented in the following code:

```
    Uri authorizationUri = authorizationRequest.createAuthorizationUri(sta
```

7. At the moment, the application is able to start the authorization flow by using the PKCE **code challenge** parameter. The authorization server still needs to receive the code verifier to check if it comes from the same application that started the OAuth 2.0 flow to receive the access token. For our client application to work, it needs to send the code verifier at the moment of the access token request. Open the `AuthorizationCodeActivity` class and add the following class attribute:

```
    private AccessTokenRequest accessTokenRequest;
```

8. After declaring the previous attribute, initialize it inside the onCreate method by creating an instance of `AccessTokenRequest` by injecting an instance of `PkceManager` as follows (you can create the instance of `AccessTokenRequest` just below the `OAuth2StateManager` definition):

```
    ...
    tokenStore = new TokenStore(this);
    manager = new OAuth2StateManager(this);
    accessTokenRequest = new AccessTokenRequest(new PkceManager(this));
    ...
```

9. And finally, replace the access token request call, to use the `from`

instance method instead of `fromCode` static method, as presented in the following code:

```
Call<AccessToken> accessTokenCall = ClientAPI
        .oauth2()
        .requestToken(accessTokenRequest.from(code));
```

How it works...

To start supporting PKCE, our client application basically sends a new parameter to the authorization request and another new parameter to the token endpoint. These parameters, `code_challenge` and `code_verifier`, are bound to each other, such that the `code_challenge` parameter is derived from the `code_verifier` value. The `code_challenge` is just SHA256 signed content from the `code_verifier` value. So our client application sends a signed version of a randomly created string to the Authorization Server and it uses the `code_verifier` later to check if the Authorization Code is being used to create a new access token that was sent by the same client that requested user authorization before.

Our client application must work, because the server application which is provided for this recipe adds support for PKCE too. When using the Authorization Code grant type, RFC 8252 states that the OAuth 2.0 Provider is required to protect the Authorization Code grants using PKCE (you can check for more at https://tools.ietf.org/html/rfc8252).

See also

- Preparing the Android development environment
- Creating an Android OAuth 2.0 client using an Authorization Code with the system browser
- Proof Key for Code Exchange by OAuth public clients specification at https://tools.ietf.org/html/rfc7636

Using dynamic client registration with mobile applications

As stated by RFC 8252, native apps are stated as public clients except when using a mechanism like dynamic client registration. By using dynamic client registration (which also has it's own specification defined by RFC 7591 and can be read at https://tools.ietf.org/html/rfc7591), we create the possibility of having separate credentials for each client installation. The main advantage achieved by this approach is that it compromises just one client application instead of all client applications that may share the same credentials. It also creates the possibility to store the credentials in the memory rather than using local storage. This recipe presents you with how you can create an Android app that registers itself against the OAuth 2.0 Provider to improve the safeness of the application.

Issuing an a client secret for a native client, when using dynamic client registration is not a problem regarding the Security Considerations section from OAuth 2.0 specification. Inside this section, there is a sub-section called ***Client Authentication*** *which states that the Authorization Server, may issue a client secret for a specific installation of a native application client running on specific device (which is our case now).*

Getting ready

To run this recipe you need the `server` application running on your environment. This is provided together with the book's source code examples at GitHub, within the `Chapter07` directory. Make sure that this project is running, and to do so, you can just go to the `Chapter07/server` directory and run the `mvn spring-boot:run` command.

For this recipe, you also need to use the project `AuthCodeApp` created for the *Creating an Android OAuth 2.0 client using an Authorization Code with the system browser* recipe. If you still don't have this project, you can either create it by reading the respective recipe or by downloading the `AuthCodeApp` directly from GitHub. In any case, the complete source code for this recipe is also available at GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter07/DynamicRegisterApp.

How to do it...

Once you have the `AuthCodeApp` imported to Android Studio, perform the following steps to add support for dynamic client registration:

1. As the application has to handle it's own registration, create the sub-package `client/registration`.
2. Then, within the registration package, create the following class to represent client credentials (which is `clientId` and `clientSecret`). Don't forget to create the respective getters and setters for each attribute:

```
public class ClientCredentials {
    private String clientId;
    private String clientSecret;
    // getters and setters omitted for brevity
}
```

3. Now, to store an instance of `ClientCredentials` in memory, create the class `ClientCredentialsStore` which will be a singleton instance as presented in the following code (by using a singleton instance we are preventing the client credentials from being saved on local storage, which must be avoided):

```
public final class ClientCredentialsStore {
    private static ClientCredentialsStore instance;
    private ClientCredentials credentials = null;
    private ClientCredentialsStore() {}
    public static ClientCredentialsStore getInstance() {
        if (instance == null) {
            instance = new ClientCredentialsStore();
        }
        return instance;
    }
    public void save(ClientCredentials credentials) {
        this.credentials = credentials;
    }
    public ClientCredentials get() {
        return this.credentials;
    }
}
```

4. To allow the client application to send requests to register itself against the running OAuth 2.0 server provider, we also need to create data

structures that represent both the request and the response. First of all, create the request data structure by defining the following class inside the `registration` sub-package. Also create all respective getters and setters for each attribute:

```
public class ClientRegistrationRequest {
    @JsonProperty("client_name")
    private String clientName;
    @JsonProperty("client_uri")
    private String clientUri;
    @JsonProperty("scope")
    private String scope;
    @JsonProperty("software_id")
    private String softwareId;
    @JsonProperty("redirect_uris")
    private Set<String> redirectUris = new HashSet<>();
    @JsonProperty("grant_types")
    private Set<String> grantTypes = new HashSet<>();
    // getters and setters omitted
}
```

5. Then create the response data structure by defining the following class with its respective getter and setter methods that were omitted for brevity:

```
public class ClientRegistrationResponse {
    @JsonProperty("redirect_uris")
    private Set<String> redirectUris = new HashSet<>();
    @JsonProperty("token_endpoint_auth_method")
    private String tokenEndpointAuthMethod;
    @JsonProperty("grant_types")
    private Set<String> grantTypes = new HashSet<>();
    @JsonProperty("response_types")
    private Set<String> responseTypes = new HashSet<>();
    @JsonProperty("client_name")
    private String clientName;
    @JsonProperty("client_uri")
    private String clientUri;
    @JsonProperty("scope")
    private String scope;
    @JsonProperty("software_id")
    private String softwareId;
    @JsonProperty("client_id")
    private String clientId;
    @JsonProperty("client_secret")
    private String clientSecret;
    @JsonProperty("client_secret_expires_at")
    private long clientSecretExpiresAt;
    // getters and setters hidden
}
```

6. Now we are ready to declare the following interface which will be used

by `retrofit2` to generate the appropriate proxy to allow our application to make HTTP requests to the registration endpoint of the OAuth 2.0 Provider:

```
public interface ClientRegistrationAPI {
    @POST("/register")
    Call<ClientRegistrationResponse> register(
        @Body ClientRegistrationRequest request);
}
```

7. All the previous recipes that need to authenticate the client against the Authorization Server, use hard coded client credentials within the interceptor `OAuth2ClientAuthenticationInterceptor`. But as we are creating an application that self registers against the Authorization Server, these client credentials can't be hard coded because we don't know the client ID and client secret at designing time. It needs to be dynamically configured, so let's change the source code for `OAuth2ClientAuthenticationInterceptor` as presented in the following code:

```
public class OAuth2ClientAuthenticationInterceptor implements Intercep
    private String username;
    private String password;
    public OAuth2ClientAuthenticationInterceptor(String username, Stri
        this.username = username;
        this.password = password;
    }
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request request = chain.request();
        Request authenticatedRequest = request.newBuilder()
            .addHeader("Authorization", getEncodedAuthorization())
            .addHeader("Content-Type", "application/x-www-form-urlenco
            .method(request.method(), request.body())
            .build();
        return chain.proceed(authenticatedRequest);
    }
    private String getEncodedAuthorization() {
        String credentials = username + ":" + password;
        return "Basic " + Base64.encodeToString(
         credentials.getBytes(), Base64.NO_WRAP);
    }
}
```

8. Once we have changed the constructor for the`OAuth2ClientAuthenticationInterceptor` class, it will generate a compiler error for the `ClientAPI` class inside the `oauth2` method. To fix this issue, just change the `oauth2` method signature to accept a reference for the

`ClientCredentials` class and propagate the client ID and client secret as arguments for the `OAuth2ClientAuthenticationInterceptor` constructor as presented in the following code:

```
public static OAuth2API oauth2(ClientCredentials clientCredentials) {
    RetrofitAPIFactory api = new RetrofitAPIFactory(BASE_URL,
        new OAuth2ClientAuthenticationInterceptor(
            clientCredentials.getClientId(), clientCredentials.getClie
    return api.getRetrofit().create(OAuth2API.class);
}
```

9. Still within the `ClientAPI` class, add the following method to provide an instance of the required API for client registration:

```
public static ClientRegistrationAPI registration() {
    RetrofitAPIFactory api = new RetrofitAPIFactory(BASE_URL, null);
    return api.getRetrofit().create(ClientRegistrationAPI.class);
}
```

10. Now, within the `client` sub-package, create the following interface to help our application perform actions after the client is properly registered against the OAuth 2.0 Provider (or to properly handle an error in the registration process):

```
public interface OnClientRegistrationResult {
    void onSuccessfulClientRegistration(ClientCredentials credentials)
    void onFailedClientRegistration(String s, Throwable t);
}
```

11. Instead of leading our activities handle the client registration process directly by using the `ClientRegistrationAPI` interface, it will be better to isolate this code within a service class, because it's required to create an instance of the `ClientRegistrationRequest` as well as handling the `ClientRegistrationResponse`. By isolating this process within the following class we allow for cleaner `Activities`:

```
public class ClientRegistrationService {
    public void registerClient(final OnClientRegistrationResult regist
        ClientRegistrationRequest request = new ClientRegistrationRequ
        request.setScope("read_profile");
        request.setClientName("android-app");
        request.setClientUri("http://adolfoeloy.com.br/en");
        request.setSoftwareId("android-packt");
        request.getGrantTypes().add("authorization_code");
        request.getRedirectUris().add(AuthorizationRequest.REDIRECT_UF
        Call<ClientRegistrationResponse> call = ClientAPI.registratior
        call.enqueue(new Callback<ClientRegistrationResponse>() {
            @Override
            public void onResponse(Call<ClientRegistrationResponse> ca
                                   Response<ClientRegistrationResponse
                ClientRegistrationResponse credentialsResponse = respo
                registrationResult.onSuccessfulClientRegistration(
                        createClientCredentials(credentialsResponse));
            }
            @Override
            public void onFailure(Call<ClientRegistrationResponse> cal
                registrationResult.onFailedClientRegistration(
                        "Failed on trying to register client", t);
            }
        });
    }
    @NonNull
    private ClientCredentials createClientCredentials(
            ClientRegistrationResponse credentialsResponse) {
        ClientCredentials credentials = new ClientCredentials();
        credentials.setClientId(credentialsResponse.getClientId());
        credentials.setClientSecret(credentialsResponse.getClientSecre
        return credentials;
    }
}
```

12. Now add the following attribute inside `MainActivity`:

```
    private ClientRegistrationService clientRegistrationService;
```

13. And create an instance of `ClientRegistrationService` within the `onCreate` method as presented in the following code:

```
    clientRegistrationService = new ClientRegistrationService();
```

14. With the main `Activity`, we have to deal with client registration and proceed with the existing flow if there is a registered client. To avoid code duplication, add the following private method at the end of the `MainActivity` class:

```
    private void proceed(ClientCredentials clientCredentials) {
        String state = UUID.randomUUID().toString();
        oauth2StateManager.saveState(state);

        Uri authorizationUri = AuthorizationRequest.createAuthorizationUri
        AccessToken token = tokenStore.getToken();
        final Intent intent;
        if (token != null && !token.isExpired()) {
            intent = new Intent(this, ProfileActivity.class);
        } else {
            intent = new Intent(Intent.ACTION_VIEW);
            intent.setFlags(Intent.FLAG_ACTIVITY_NO_HISTORY);
            intent.setData(authorizationUri);
        }
        startActivity(intent);
    }
```

15. Now replace the code within the `onClick` method, with the following snippet of code:

```
    ClientCredentialsStore credentialsStore = ClientCredentialsStore.getIr
    ClientCredentials clientCredentials = credentialsStore.get();
    if (clientCredentials != null) {
        proceed(clientCredentials);
    } else {
        clientRegistrationService.registerClient(this);
    }
```

16. As you might notice, we are passing a reference for the `MainActivity` to the `registerClient` method from the `ClientRegistrationService`. To fix the current compilation error, make sure that `MainActivity` implements the `OnClientRegistrationResult` interface as follows:

```
    public class MainActivity extends AppCompatActivity
        implements View.OnClickListener, OnClientRegistrationResult {
        ....
    }
```

17. Then implement the required methods as presented in the following code:

```
    @Override
    public void onSuccessfulClientRegistration(ClientCredentials credentia
        ClientCredentialsStore store = ClientCredentialsStore.getInstance(
        store.save(credentials);
        proceed(credentials);
    }
    @Override
    public void onFailedClientRegistration(String s, Throwable t) {
        Log.e("MainActivity", "Error trying to register client credentials
    }
```

18. The source code for the `MainActivity` class doesn't compile yet. That's because the `createAuthorizationUri` method now expects one more parameter instead of just the state parameter. Now we need to send the `clientCredentials` instance so the authorization URI can be built using the dynamically registered client. Open the `AuthorizationRequest` class and change the `createAuthorizationUri` method signature to what follows:

```
    public static Uri createAuthorizationUri(String state,
            ClientCredentials clientCredentials) {
            ....
        }
```

19. Besides changing the method signature, replace the `client_id` attribute definition which is hard coded by the following dynamic definition:

```
        .appendQueryParameter("client_id", clientCredentials.getClientId())
```

20. Now there is just one more class that isn't compiling. That's the `AuthorizationCodeActivity`. Open this class and replace the `ClientAPI.oauth().requestToken` invocation to look like the snippet of code presented:

```
        ClientCredentials credentials = ClientCredentialsStore.getInstance().g
        Call<AccessToken> accessTokenCall = ClientAPI.oauth2(credentials)
                .requestToken(AccessTokenRequest.fromCode(code));
```

21. Uninstall the `AuthCodeApp` present in your current device (or virtual device) and install it again by running the application from Android Studio.

How it works...

When running our application for the first time, when the user clicks on the Get User Profile button, the application will start registering itself against the Authorization Server, holding the client credentials in memory to avoid data from getting easily leaked. The way we have implemented our application, every time the user shuts down the device, the client credentials will be lost and it will require the client to self-register again, thus decreasing the user experience of our application.

To avoid this kind of problem, a better way to handle this scenario would be to check for an existent access token beforehand. Then, whenever an access token is valid, we don't need to register the application again. We could just check for the client registration if there is any valid access token. As an exercise, change the application to switch the way it validates access tokens and client credentials.

See also

- Preparing the Android development environment
- Creating an Android OAuth 2.0 client using an Authorization Code with the system browser

Avoiding Common Vulnerabilities

This chapter will cover the following recipes:

- Validating the Resource Server audience
- Protecting Resource Server with scope validation
- Binding scopes with user roles to protect user's resources
- Protecting the client against Authorization Code injection
- Protecting the Authorization Server from invalid redirection

Introduction

This chapter will present you with the means to better protect applications that interact with the OAuth 2.0 ecosystem. That's to protect all the main components of OAuth 2.0 against common issues. The components that will be covered are the client and the OAuth Provider, which is composed by the Authorization Server and the Resource Server.

Although this chapter presents you some best practices to avoid common vulnerabilities when working with OAuth 2.0, be aware to always protect connections with SSL/TLS when running these recipes in production.

Validating the Resource Server audience

Although a bearer access token can be used by anyone to access OAuth 2.0 protected resources, you can reduce the scope of access token usage just for specified resources; that's to set up the audience for an access token. If an access token has the audience for Resource Server A, it cannot be used to access resources protected by Resource Server B. This chapter will cover this important feature, so that you can use it when you have an Authorization Server serving multiple Resource Servers.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. Because of the architecture of this solution, this recipe requires that you create three applications: one for the Authorization Server and two for the Resource Servers. The Authorization Server was created with the name `authorization-server` and the resources servers were created as `resource-server-a` and `resource-server-b`. The source code for these projects are all available on GitHub at https://github.com/PacktPublishing/OAuth-2.0-Cookbook/tree/master/Chapter08/validate-audience. To ease the project creation step, you can use Spring Initializr which can be accessed at http://start.spring.io/ . Define the dependencies as `Web` and `Security` (that will declare properly all the spring boot starters needed for this recipe). Do not forget to set up the Artifact and Group names.

How to do it...

Create the projects `authorization-server`, `resource-server-a`, and `resource-server-b` and import them as a Maven project into your IDE. Then perform the following steps:

1. Add the following extra dependency for Spring Security OAuth2 within the `pom.xml` file for each project created in this recipe:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.2.0.RELEASE</version><!--$NO-MVN-MAN-VER$-->
 </dependency>
```

2. Add the following content to the `application.properties` file from the `authorization-server` project:

```
security.user.name=adolfo
security.user.password=123
```

3. Add the following content within `application.properties` from the `resource-server-a` project:

```
server.port=9000
security.oauth2.client.client-id=client-a
security.oauth2.client.client-secret=123
security.oauth2.resource.token-info-uri=http://localhost:8080/oauth/ch
```

4. Add the following content within the `application.properties` file from the `resource-server-b` project:

```
server.port=9001
security.oauth2.client.client-id=client-b
security.oauth2.client.client-secret=123
security.oauth2.resource.token-info-uri=http://localhost:8080/oauth/ch
```

5. Now in the project `authorization-server`, declare the class `OAuth2AuthorizationServer` within the package `com.packt.example.authorizationserver.oauth` with the following content:

```
@Configuration @EnableAuthorizationServer
```

```
public class OAuth2AuthorizationServer
    extends AuthorizationServerConfigurerAdapter {
    @Autowired
    private AuthenticationManager authenticationManager;
    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpc
        throws Exception {
        endpoints.authenticationManager(authenticationManager);
    }
    @Override
    public void configure(AuthorizationServerSecurityConfigurer securi
        throws Exception {
        security.checkTokenAccess("hasAuthority('introspection')");
    }
    @Override
    public void configure(ClientDetailsServiceConfigurer clients)
       throws Exception {
       clients.inMemory()
          .withClient("client-a").secret("123")
          .authorizedGrantTypes("password")
          .scopes("read_profile", "read_contacts")
          .authorities("introspection").resourceIds("resource-a")
      .and()
          .withClient("client-b").secret("123")
          .authorizedGrantTypes("password")
          .scopes("read_profile").authorities("introspection")
          .resourceIds("resource-b");
     }
 }
```

6. The Authorization Server is ready to go. Now, let's configure Resource Server A by creating the class `ResourceA` inside the `resource-server-a` project with the following content:

```
@Configuration @Controller @EnableResourceServer
public class ResourceA
   extends ResourceServerConfigurerAdapter {
   @Override
   public void configure(ResourceServerSecurityConfigurer resources)
       throws Exception {
       resources.resourceId("resource-a");
   }
   @Override
   public void configure(HttpSecurity http) throws Exception {
       http.authorizeRequests().anyRequest().authenticated().and()
           .requestMatchers().antMatchers("/res-a");
   }
   @RequestMapping("/res-a")
   public ResponseEntity<String> resourceA() {
       return ResponseEntity.ok("resource A with success");
   }
}
```

7. Now configure Resource Server B by adding the following class into the

`resource-server-b` project:

```
@Configuration @Controller @EnableResourceServer
public class ResourceB
    extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(ResourceServerSecurityConfigurer resources)
        throws Exception {
        resources.resourceId("resource-b");
    }
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/res-b");
    }
    @RequestMapping("/res-b")
    public ResponseEntity<String> resourceB() {
        return ResponseEntity.ok("resource B with success");
    }
}
```

How it works...

Within the Authorization Server, we have declared two client configurations with `client-a` and `client-b` credentials. Each one is bound to one resource ID, which makes reference to respective resource servers declared through separate projects. As both Resource Servers are configured as separate projects, we also had to enable remote token validation by defining the authority to check the token access endpoint.

Each Resource Server was configured in a really simple manner, taking advantage of Spring Boot properties. As you might notice, we don't have to declare the `RemoteTokenServices` bean if we declare the following properties:

```
security.oauth2.client.client-id=client-id
security.oauth2.client.client-secret=secret
security.oauth2.resource.token-info-uri=http://localhost:8080/oauth/check_tok
```

Both Resource Servers provide an endpoint, called `/res-a` or `/res-b` , and define the resource ID that maps to Resource Server A or B. If we run all the applications, and request an access token for `client-a`, this token can only access the endpoint `/res-a` from project `resource-server-a`.

To better understand how the token audience is validated, start all three applications and create an access token for `client-a` by running the following command:

```
curl -X POST --user client-a:123 -d "grant_type=password&username=adolfo&pass
```

Then check the access token returned by the previous command by running the following command:

```
curl -X GET --user client-a:123 "http://localhost:8080/oauth/check_token?toke
```

You should see something similar to what follows (notice the `aud` attribute):

```
{
 "aud": ["resource-a"],
 "user_name": "adolfo",
 "scope": ["read_profile"],
 "active": true,
 "exp": 1506302223,
 "authorities": ["ROLE_USER"],
 "client_id": "client-a"
}
```

Then if you try to access the `res-a` endpoint by running the following command, you should receive a successful response:

```
curl -X GET http://localhost:9000/res-a -H 'authorization: Bearer 35a35b85-2c
```

But if you try to access the endpoint `http://localhost:9001/res-b`, you should receive the following response:

```
{
 "error": "access_denied",
 "error_description": "Invalid token does not contain resource id (resource-b
}
```

So if `client-a` needs to access `Resource B`, it needs to have the corresponding resource ID bound to its client details.

Protecting Resource Server with scope validation

At the Authorization Server, we have been declaring the available scopes that the user can approve when authorizing some third-party applications to use resources on her behalf. But actually, until now we aren't protecting resources based on approved scopes. It makes sense to start validating scopes for different features on the Resource Server. This recipe presents you with how you can take advantage of Spring Security OAuth2 and Spring Security to start validating scopes and to add a fine grained protection to the user's resources.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies as `Web` and `Security` (that will declare properly all the spring boot starters needed for this recipe). Do not forget to set up the Artifact and Group names.

How to do it...

This recipe creates the project `scope-validation` which is available on GitHub in the `Chapter08` folder. Import the generated project as a Maven project into your IDE and follow the following steps:

1. Open the `pom.xml` file and add the following extra dependency for *Spring Security OAuth2*. I am considering that you have already imported the starters for `Web` and `Security`:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.2.0.RELEASE</version>
</dependency>
```

2. Open `application.properties` and add the following content:

```
security.user.name=adolfo
security.user.password=123
```

3. The resources that we will use to protect this recipe are simply endpoints that are OAuth 2.0 protected. As these endpoints are mapped to methods declared within controllers, we can enable method security validation so that we can annotate them with `@PreAuthorize`, `@PreFilter`, `@PostAuthorize`, or `@PostFilter`. To enable method validation, annotate the main class in the project with `@EnableGlobalMethodSecurity` as follows (the main class is supposed to be annotated with `@SpringBootApplication`):

```
@SpringBootApplication
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ScopeValidationApplication {
   public static void main(String[] args) {
      SpringApplication.run(ScopeValidationApplication.class, args);
   }
}
```

4. Then, create the following controller, which will declare some endpoints to be available just for specified scopes (note that to access `/api/x` the access token must be scoped to `read_x`, and to access `/api/y`, `read_y`, scope

is required):

```
@Controller
public class ApiController {

    @PreAuthorize("#oauth2.hasScope('read_x')")
    @RequestMapping("/api/x")
    public ResponseEntity<String> resourceX() {
        return ResponseEntity.ok("resource X");
    }

    @PreAuthorize("#oauth2.hasScope('read_y')")
    @RequestMapping("/api/y")
    public ResponseEntity<String> resourceY() {
        return ResponseEntity.ok("resource Y");
    }

}
```

5. Finally, let's create the Authorization Server and the Resource Server by defining the endpoints to be protected and the scopes allowed for the user to authorize:

```
@Configuration
public class OAuthConfiguration {

    @EnableAuthorizationServer
    public static class AuthorizationServer
        extends AuthorizationServerConfigurerAdapter {
        public void configure(ClientDetailsServiceConfigurer clients)
            clients.inMemory()
                .withClient("clientapp").secret("123")
                .scopes("read_x", "read_y")
                .authorizedGrantTypes("authorization_code");
        }
    }
    @EnableResourceServer
    public static class ResourceServer
        extends ResourceServerConfigurerAdapter {
        public void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests().anyRequest().authenticated().and(
                .requestMatchers().antMatchers("/api/**");
        }
    }
}
```

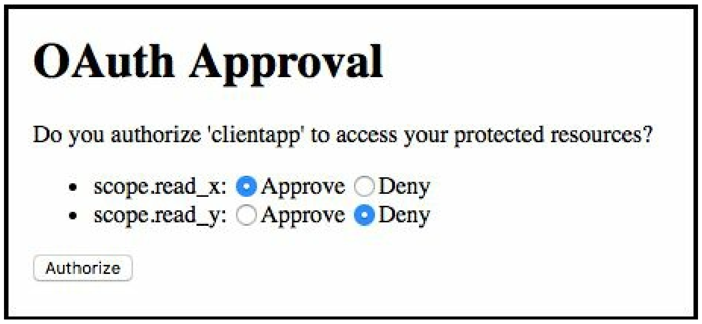
How it works...

Now the protected resources are bound to their respective scopes. As you can see, the client details declaration defines `read_x` and `read_y` scopes as available for the user to approve. The client application can ask for both scopes, but if the Resource Owner authorizes just `read_x` or `read_y`, only endpoints which are bound to scopes chosen by the Resource Owner will be allowed.

To better understand what's going on here, start the application, open your browser and go to the following authorization URL:

```
http://localhost:8080/oauth/authorize?client_id=clientapp&redirect_uri=http:/
```

Authenticate using `adolfo` and `123` as the username and password respectively. After being authenticated, the Authorization Server will present you the following screen where you can choose the scopes to authorize. Select just `read_x` so we can explore all scope validation results:


OAuth Approval

Do you authorize 'clientapp' to access your protected resources?

- scope.read_x: [✓ Approve] Deny
- scope.read_y: Approve [✓ Deny]

Then click on Authorize and retrieve the Authorization Code that will be present on the browser address bar (the parameter is `code` as you might already know). Request an access token using this Authorization Code as follows:

```
curl -X POST "http://localhost:8080/oauth/token" --user clientapp:123 -H "cor
```

The result of the previous command will present you with a new access token and the scope that approved that must be just `read_x`. Copy the access token and try to access the endpoint `api/x` as follows:

```
curl -X GET http://localhost:8080/api/x -H "authorization: Bearer 1770ebcb01e
```

It must return the string `resource X`, but if you try to access the endpoint `/api/y` you should receive the following response:

```
{
 "error": "access_denied",
 "error_description": "Access denied"
}
```

Binding scopes with user roles to protect user's resources

By using scopes we can add fine-grained protection to the user's resources. Spring Security also provides the concept of roles which can be interchangeable with scopes defined by Spring Security OAuth2. As explained by Dave Syer (leader of the Spring Securit OAuth2 project), roles and scopes are just arbitrary strings that might be considered as the same thing (you can read more about this explanation at https://stackoverflow.com/questions/22417780/using-scopes-as-roles-in-spring-security-oauth2-provider). This recipe presents you with how to bind scopes that are available for some client details with the roles that a Resource Owner might have. In this case, the Resource Owner would be able to approve scopes that are related to her roles.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies as `Web` and `Security` (that will declare properly all the spring boot starters needed for this recipe). Do not forget to set up the Artifact and Group names.

How to do it...

This recipe creates the project `scope-binding` which is available on GitHub in the `Chapter08` folder. Import the generated project as a Maven project into your IDE and follow the following steps:

1. Open the `pom.xml` file and add the following extra dependency for *Spring Security OAuth2*. I am considering that you have already imported the starters for `Web` and `Security`:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.2.0.RELEASE</version>
</dependency>
```

2. Instead of defining the default user configuration within the `application.properties` file, let's create implementations for `UserDetails` and `UserDetailsService`. As we have to define the authorities related to allowed scopes for the Resource Owner, we also need to define the role `ROLE_USER`, which is required because of validations performed by `AccessDecisionManager` (this is an internal Spring Security class that relies on instances of `AccessDecisionVoter`). So create the class `CustomUser` as follows and make sure to return true for all Boolean methods:

```
public class CustomUser implements UserDetails {
    private final String username;
    private final String password;
    private final List<String> authorities;
    public CustomUser(String username, String password,
        List<String> authorities) {
        this.username = username;
        this.password = password;
        this.authorities = authorities;
    }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities.stream()
           .map((item) -> new SimpleGrantedAuthority(item))
            .collect(Collectors.toList());
    }
    @Override public String getPassword() { return password; }
    @Override public String getUsername() { return username; }
    // all boolean methods omitted
}
```

3. Then create the class `CustomUserDetailsService` as follows:

```
@Service
public class CustomUserDetailsService
    implements UserDetailsService, InitializingBean {
    private Map<String, CustomUser> users = new HashMap<>();
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        return users.get(username);
    }
    public void afterPropertiesSet() throws Exception {
        users.put("adolfo", new CustomUser(
           "adolfo", "123", Arrays.asList("ROLE_USER", "read_x")));
    }
}
```

4. As we did for `scope-validation` project in the *Protect Resource Server with Scope validation* recipe, annotate the main class with `@EnableGlobalMethodSecurity` as presented in the following code:

```
@SpringBootApplication
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ScopeBindingApplication {
   public static void main(String[] args) {
      SpringApplication.run(ScopeBindingApplication.class, args);
   }
}
```

5. As we are configuring an OAuth Provider that provides both Authorization Server and Resource Server functionalities, declare both responsibilities within the same class (as internal classes):

```
@Configuration
public class OAuthConfiguration {
    @EnableAuthorizationServer
    public static class AuthorizationServer
        extends AuthorizationServerConfigurerAdapter {
        @Autowired
        private ClientDetailsService clientDetailsService;
        @Override
        public void configure(AuthorizationServerEndpointsConfigurer e
            throws Exception {
            DefaultOAuth2RequestFactory factory =
                new DefaultOAuth2RequestFactory(clientDetailsService);
            factory.setCheckUserScopes(true);
            endpoints.requestFactory(factory);
        }
        public void configure(ClientDetailsServiceConfigurer clients)
            throws Exception {
                clients.inMemory()
                  .withClient("clientapp").secret("123")
                  .scopes("read_x", "read_y")
                  .authorities("read_x", "read_y")
                  .authorizedGrantTypes("authorization_code");
            }
        }
        @EnableResourceServer
        public static class ResourceServer
            extends ResourceServerConfigurerAdapter {
            public void configure(HttpSecurity http) throws Exception {
                http.authorizeRequests().anyRequest().authenticated().and(
                    .requestMatchers().antMatchers("/api/**");
            }
        }
    }
```

6. Finally, declare the controller which provides the APIs protected by OAuth 2.0 scopes:

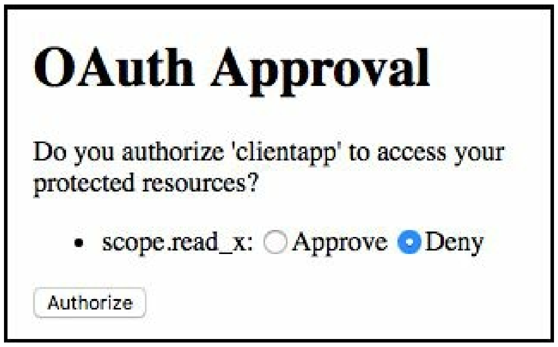
```
    @Controller
    public class ApiController {
        @PreAuthorize("#oauth2.hasScope('read_x')")
        @RequestMapping("/api/x")
        public ResponseEntity<String> resourceX() {
            return ResponseEntity.ok("resource X");
        }
        @PreAuthorize("#oauth2.hasScope('read_y')")
        @RequestMapping("/api/y")
        public ResponseEntity<String> resourceY() {
            return ResponseEntity.ok("resource Y");
        }
    }
```

7. Now, the application is ready to be started.

How it works...

Besides configuring authorities for in-memory registered client through client details, now we can also configure authorities for the Resource Owner so he can authorize just the scopes that are related to his roles. This is connected by the usage of the `checkUserScopes` method from `DefaultOAuth2RequestFactory` class.

Once you start the application, and start the authorization flow, the user consent page will present you only scopes that the Resource Owner has configured as authorities or roles (role is just a special case of authority, where the prefix ROLE is placed in front of an authority string). Although we have declared `read_x` and `read_y` scopes for the `clientapp` client, the user will be actually able to choose just `read_x` in the consent page, as presented in the following screenshot:



OAuth Approval

Do you authorize 'clientapp' to access your protected resources?

- scope.read_x: Approve [✓ Deny]

Authorize the `read_x` scope, copy the Authorization Code returned within the browser's address bar and request a new access token at the `/oauth/token` endpoint. You should receive something similar as follows (notice that this access token just has the `read_x` scope attached):

```
{
 "access_token": "8d5dd151-bffc-4698-b8a8-380a8825b25d",
 "token_type": "bearer",
 "expires_in": 43199,
 "scope": "read_x"
}
```

Try to use the returned access token to access the `http://localhost:8080/api/x` endpoint and you must receive a successful response. Then try to access the endpoint `http://localhost:8080/api/y` using the same access token and the response must be as follows:

```
{
 "error": "access_denied",
 "error_description": "Access denied"
}
```

See also

- Protecting Resource Server with scope validation

Protecting the client against Authorization Code injection

This recipe talks about a security flaw that is extremely easy to mitigate, but unfortunately, there are many companies that still do not pay attention to this problem. That's about **Cross-Site Request Forgery** (**CSRF**) attacks, which allow anybody to inject a forged authorization code to compromise the Resource Owner's resources. This recipe shows the importance of state parameters when using the Authorization Code grant type (although it's also important when using the Implicit grant type).

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. If you want to run the examples to explore how to simulate an attack, you have to install Firefox and the `NoRedirect` add-on. To ease the project creation step, use Spring Initializr at http://start.spring.io/.

How to do it...

This recipe creates the project `oauth2-provider-state` for OAuth Provider and `client-state` for the client application. Both applications are available on GitHub in the `state-param/Chapter08` folder:

1. Create the project `oauth2-provider-state` on Spring Initializr, and define the dependencies as `Web`, `Security`, `JPA`, and `H2`.
2. Download and import the generated project as a Maven project into your IDE.
3. Add the following extra dependency within `pom.xml` from the `oauth2-provider-state` project. I am considering that you have already imported the starters for `Web` and `Security`:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.2.0.RELEASE</version>
</dependency>
```

4. Create the following class to represent the Resource Owner on the Authorization Server (return true to all Boolean getters that you have to implement because of the `UserDetails` interface):

```
@Entity @Table(name = "custom_user")
public class CustomUser implements UserDetails {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"));
    }
    @Override
    public String getPassword() { return password; }
    @Override
    public String getUsername() { return username; }
    // boolean methods omitted for brevity
}
```

5. Create the following Repository interface for the `CustomUser` class:

```
public interface CustomUserRepository
    extends JpaRepository<CustomUser, Long> {
    Optional<CustomUser> findByUsername(String username);
}
```

6. Create the `CustomUserDetailsService` class as presented in the following code to allow for `CustomUser` authentication:

```
@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Autowired
    private CustomUserRepository repository;
    @Override
    public UserDetails loadUserByUsername(String username) throws User
        Optional<CustomUser> user =
            repository.findByUsername(username);
        return user.orElseThrow(() ->
            new UsernameNotFoundException("user does not exists"));
    }
}
```

7. Inside the `oauth2-provider-state` project, configure a simple OAuth 2.0 Provider by creating the class `OAuth2Provider` as presented in the following code. The following class declares both the Authorization Server and the Resource Server:

```
@Configuration
public class OAuth2Provider {

    @EnableAuthorizationServer
    public static class AuthorizationServer
        extends AuthorizationServerConfigurerAdapter {

        public void configure(ClientDetailsServiceConfigurer clients)
            clients.inMemory()
                .withClient("clientapp").secret("123")
                .scopes("read", "write")
                .authorizedGrantTypes("authorization_code");
        }
    }

    @EnableResourceServer
    public static class ResourceServer
        extends ResourceServerConfigurerAdapter {
        public void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests().anyRequest().authenticated().and(
                    .requestMatchers().antMatchers("/api/**");
        }
    }
}
```

8. Still in the `oauth2-provider-state` project, create a simple API by defining

the following class. Such an API will be protected by OAuth 2.0:

```
@Controller
public class ApiController {

    @GetMapping("/api/read")
    public ResponseEntity<String> read() {
        CustomUser user = (CustomUser) SecurityContextHolder
            .getContext().getAuthentication().getPrincipal();
        return ResponseEntity.ok("success " + user.getUsername());
    }

}
```

9. Before you start creating the client application, create the file `data.sql` inside the `src/main/resources` directory from the `oauth2-provider-state` project with the following content:

```
insert into custom_user (username, password) values ('victim', '123');
insert into custom_user (username, password) values ('attacker', '123'
```

10. That's enough for the OAuth 2.0 Provider. Now create the project `client-state` on Spring Initializr and define the dependencies as `Web`, `Thymeleaf`, and `Security`.
11. Download and import the generated project as a Maven project into your IDE.
12. Add the following extra dependency within the `pom.xml` file from the `client-state` project. I am considering that you have already imported the starters for `Web` and `Security`:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.2.0.RELEASE</version>
</dependency>
```

13. Open `application.properties` from the `client-state` project and add the following content:

```
server.port=9000
server.session.cookie.name=client_session
security.user.name=adolfo
security.user.password=123
```

14. Create the following class to configure OAuth 2.0 client:

```
@Configuration @EnableOAuth2Client
public class OAuth2Configuration {
    @Bean
    public OAuth2ProtectedResourceDetails authorizationCode() {
        AuthorizationCodeResourceDetails details =
            new AuthorizationCodeResourceDetails();
        details.setId("oauth2server");
        details.setClientId("clientapp");
        details.setClientSecret("123");
        details.setUseCurrentUri(true);
        details.setUserAuthorizationUri("http://localhost:8080/oauth/a
        details.setAccessTokenUri("http://localhost:8080/oauth/token")
        return details;
    }
    @Bean
    public OAuth2RestTemplate restTemplate(OAuth2ClientContext context
        return new OAuth2RestTemplate(authorizationCode(), context);
    }
}
```

15. Then create the following controller so the user can interact with the client application:

```
@Controller
public class HomeController {
    @Autowired
    private OAuth2RestTemplate restTemplate;
    @GetMapping("/")
    public ModelAndView home() {
        User user = (User) SecurityContextHolder
            .getContext().getAuthentication().getPrincipal();
        ModelAndView mv = new ModelAndView("home");
        mv.addObject("username", user.getUsername());
        return mv;
    }
    @GetMapping("/resource")
    public ModelAndView resource() {
        String result = restTemplate
                .getForObject("http://localhost:8080/api/read", String
        ModelAndView mv = new ModelAndView("resource");
        mv.addObject("result", result);
        return mv;
    }
}
```

16. Now we have to create the template views for home and the page that allows interaction with the OAuth 2.0 Provider. Create the file `home.html` inside `src/main/resources/templates` with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:th="http://www.thymeleaf.org">
```

```
<body>
<div style="border: 3px solid black; width: 30%; padding: 10px">
    <h1>Hello</h1>
    <span th:text="${username}"></span>
    <div>
        <a th:href="@{/resource}">Get resource</a>
    </div>
</div>
</body>
</html>
```

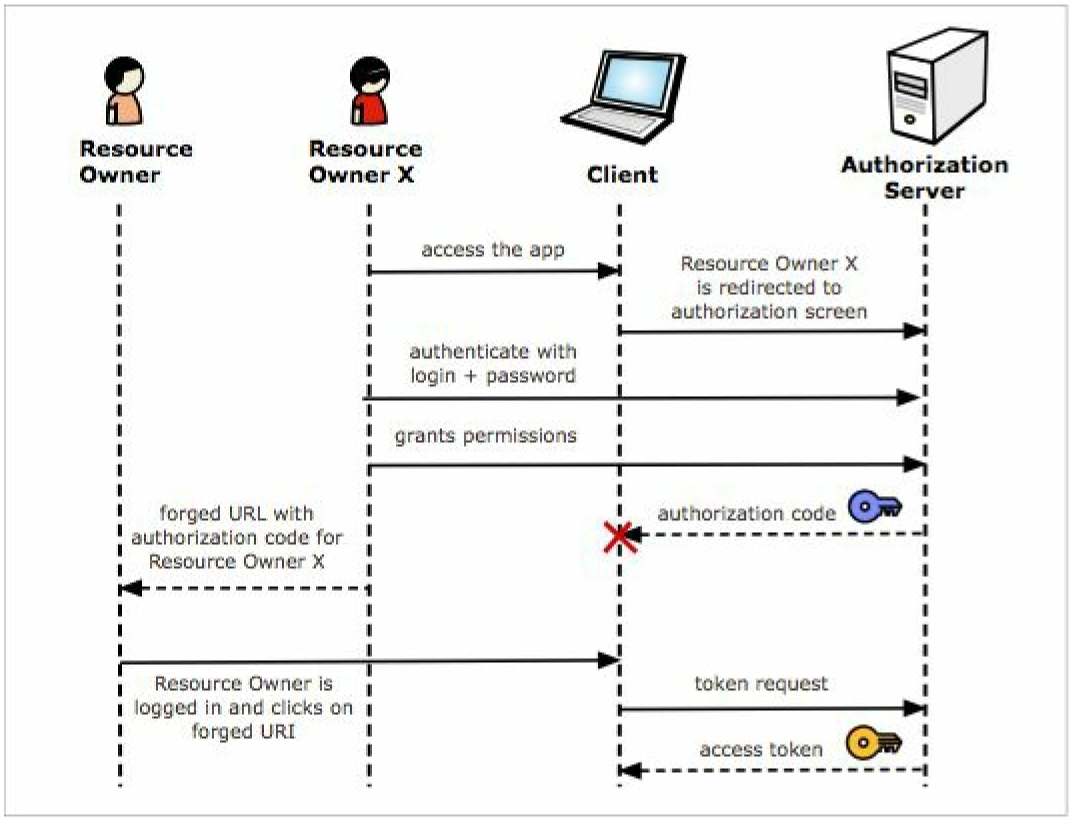
17. And finally, create the `resource.html` file within the same directory as `home.html` with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:th="http://www.thymeleaf.org">
<body>
<div style="border: 3px solid black; width: 30%; padding: 10px">
    <h1>That's the result</h1>
    <p>result:<span th:text="${result}"></span></p>
</div>
</body>
</html>
```

18. Before starting the application and trying to simulate the CSRF attack, install Firefox (if you don't have it), then install the `NoRedirect` add-on. After installing this add-on, you have to open the add-ons manager and click on Preferences for the `NoRedirect` add-on.
19. Add a rule containing the pattern `http://localhost:8080/*` and make sure to mark just the Source checkbox.
20. Move the created rule to the top of the Rule list. This will intercept every redirection requested by `localhost:8080`.

How it works...

As you might have noticed, we didn't use anything different to deal with the state parameter at the OAuth 2.0 Provider. That's because Spring Security OAuth2 already provides the state parameter support, which means that once the client sends a state parameter to `/oauth/authorize` endpoint, it will be returned after the authorization phase. So, it's totally up to the client to generate a state parameter to send to the Authorization Server, and to validate if it has the same value after the user has granted permission to access their resources. To understand how a CSRF attack can be applied to forge a URL with a malicious Authorization Code, look at the following diagram:


Resource Owner
Resource Owner X
Client
Authorization Server
access the app
Resource Owner X is redirected to authorization screen
authenticate with login + password
grants permissions
forged URL with authorization code for Resource Owner X
authorization code
Resource Owner is logged in and clicks on forged URI
token request
access token

Basically, to forge an authorization link with a valid Authorization Code, a malicious user (say Resource Owner X), can start the authorization process until she is redirected back to the client application with an Authorization Code. The user can stop the automatic redirection performed by the web browser by using a plugin such as the `NoRedirect` add-on for Firefox.

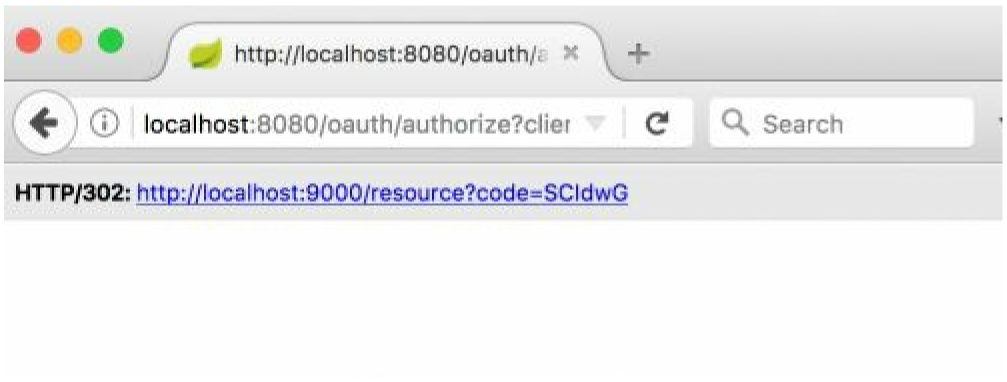
> *The `NoRedirect` add-on for Firefox allows users to set up URLs that can't be automatically redirected. The URI that comes on the Location HTTP Header response is rendered in the browser so the user can decide to keep going forward or not.*

Once a malicious user has access to the redirection URI with the Authorization Code, she can just copy the location URI presented on the web browser and send it to another logged user. By taking advantage of the victim's session on client application, if the victim clicks on the link, the client will request an access token and will bind it with the victim's account on the client application. By doing it, everything the client does on behalf of the victim will apply to Resource Owner X (the attacker). If it is about transferring money to the victim's account, the money will instead be transferred to the attacker.

Fortunately, even when using the `OAuth2RestTemplate` on the client application, Spring Security OAuth2 also automatically generates a state parameter and validates it properly. Let's try to perform a CSRF attack: start both applications, open Firefox and try to send the following request:

```
http://localhost:8080/oauth/authorize?client_id=clientapp&redirect_uri=http:/
```

Authenticate yourself using `attacker` and `123` as the username and password respectively. If you are using `NoRedirect` add-on correctly, the browser will render a redirect URI link as follows:


http://localhost:8080/oauth/a
localhost:8080/oauth/authorize?clier
Search
HTTP/302: http://localhost:9000/resource?code=SCldwG

Copy the link rendered on Firefox, go to another browser with a clean session and access `http://localhost:9000`. Authenticate using the default user account that we have defined for client application, which is `adolfo` and `123` for the username and password respectively. Then, try to replace and access the URL `http://localhost:9000/resource?code=SCIdwG` (the code attribute will be a different one for you) and you should receive an error page as follows:

```
Possible CSRF detected - state parameter was required but no state could be f
```

If Spring Security OAuth2 didn't provide the state parameter validation, the user `adolfo` would have the access token required by the attacker bound to his account. If you aren't using Spring Security on your projects, it's extremely recommended to generate and validate for state parameters.

Protecting the Authorization Server from invalid redirection

This recipe will present you with the importance of defining a redirection URI when registering client applications. By defining a redirection URI, the client always needs to send the redirection URI on the authorization request, which must be validated by the Authorization Server. Redirect URI validation is extremely important when using the Implicit grant type, because the client can't authenticate when redirecting the user so she grant permissions to her resources.

Getting ready

To run this recipe, you will need Java 8, Maven, Spring Web, and Spring Security. To ease the project creation step, use Spring Initializr at http://start.spring.io/ and define the dependencies as `Web` and `Security` (that will declare properly all the spring boot starters needed for this recipe). Do not forget to set up the Artifact and Group names.

How to do it...

This recipe creates the project `uri-validation`, which is available on GitHub in the `Chapter08` folder. Import the generated project as a Maven project into your IDE and follow the following steps:

1. Open the `pom.xml` file and add the following extra dependency for *Spring Security OAuth2*. I am considering that you have already imported the starters for `Web` and `Security`:

```
<dependency>
 <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.2.0.RELEASE</version>
</dependency>
```

2. Open the `application.properties` and add the following content:

```
security.user.name=adolfo
security.user.password=123
```

3. Create the `OAuth2Provider` class as follows (notice that we are configuring the Implicit grant type and registering a redirection URI):

```
@Configuration
public class OAuth2Provider {
    @EnableAuthorizationServer
    public static class AuthorizationServer
            extends AuthorizationServerConfigurerAdapter {
        public void configure(ClientDetailsServiceConfigurer clients)
            throws Exception {
            clients.inMemory()
                .withClient("clientapp").secret("123")
                .scopes("read", "write")
                .redirectUris("http://localhost:9000/callback")
                .authorizedGrantTypes("implicit");
        }
    }
    @EnableResourceServer
    public static class ResourceServer
            extends ResourceServerConfigurerAdapter {
        public void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests().anyRequest().authenticated().and(
                    .requestMatchers().antMatchers("/api/**");
        }
    }
}
```

4. Create the class `ApiController` to simulate an API to be OAuth 2.0 protected:

```
@Controller
public class ApiController {
    @GetMapping("/api/read")
    public ResponseEntity<String> read() {
        return ResponseEntity.ok("success");
    }
}
```

How it works...

It's possible that someone could create an ill-intentioned web application and register it against the Authorization Server to take advantage of unprotected OAuth Providers and leak user data. As an example, if an OAuth Provider does not require the user to register a redirection URI when registering the application, and the client is able to use the Implicit grant type, we might experience catastrophic results.

If you forge an authorization URI with an unregistered redirect URI, after the Resource Owner grants permission to the client application, she will be redirected back to a malicious endpoint, delivering the access token to an unauthorized client (remember that we are considering the Implicit grant type that does not require the client to authenticate against the Authorization Server).

As we registered the redirect URI `http://localhost:8080/callback` for the `clientapp` client, if you try to access the following URL through your web browser, you won't be able to receive an access token. Try to access the following URL and authenticate it using the default user credentials (`adolfo` and `123` as username and password):

```
http://localhost:8080/oauth/authorize?client_id=clientapp&redirect_uri=http:/
```

You should receive the following error response:

```
error="invalid_grant",
error_description="Invalid redirect: http://localhost:9000/malicious does not
```

As an exercise, try to remove the redirect URI configuration using the following configuration for the `AuthorizationServer` inner class, and try to access the same authorization URL presented earlier:

```
@EnableAuthorizationServer
    public static class AuthorizationServer
            extends AuthorizationServerConfigurerAdapter {
        public void configure(ClientDetailsServiceConfigurer clients)
            throws Exception {
            clients.inMemory()
```

```
                    .withClient("clientapp").secret("123")
                    .scopes("read", "write")
                    .authorizedGrantTypes("implicit");
        }
    }
```

The access token will be generated and delivered to `http://localhost:9000/mailicious` URI.