Felipe Gutierrez, *Pro Spring Boot 2*
https://doi.org/10.1007/978-1-4842-3676-5_8

8. Security with Spring Boot

Felipe Gutierrez [1]

(1) Albuquerque, NM, USA

This chapter shows you how to use security in your Spring Boot applications to secure your web application. You learn everything from using basic security to using OAuth. Security has become a primary and important factor for desktop, web, and mobile applications in the last decade. But security is a little hard to implement because you need to think about everything—cross-site scripting, authorization, and authentication, secure sessions, identification, encryption, and a lot more. There is still a lot to do to implement simple security in your applications.

The Spring security team has worked hard to make it easier for developers to bring security to their applications, from securing service methods to entire web applications. Spring security is centered around `AuthenticationProvider`, `AuthenticationManager`, and specialized `UserDetailsService`; it also provides integration with identity provider systems, such as LDAP, Active Directory, Kerberos, PAM, OAuth, and so on. You are going to review a few of them in the examples in this chapter.

Spring Security

Spring Security is highly customizable and powerful framework that helps with authentication and authorization (or access control); it is the default module for securing Spring applications. The following are some of the important features.

- Servlet API integration
- Integration with Spring Web MVC and WebFlux
- Protection against attacks such as session fixation, clickjacking, CSRF (cross-site request forgery), CORS (cross-origin resource sharing), and so forth
- Extensible and comprehensive support for both authentication and authorization
- Integration with these technologies: HTTP Basic, HTTP Digest, X.509, LDAP, Form-based, OpenID, CAS, RMI, Kerberos, JAAS, Java EE, and more
- Integration with third-party technologies: AppFuse, DWR, Grails, Tapestry, JOSSO, AndroMDA, Roller, and many more

Spring Security has become the de facto way to use security on many Java and Spring projects because it integrates and customizes with minimal effort, creating robust and secure apps.

Security with Spring Boot

Spring Boot uses the power of the Spring Security Framework to secure applications. To use Spring Security it is necessary to add the `spring-boot-starter-security` dependency. This dependency provides all the `spring-security` core jars and it auto-configures the strategy to determine whether to use `httpBasic` or `formLogin` authentication mechanisms. It defaults to `UserDetailService` with a single user. This username is `user` and the pass-

word is printed (RANDOM string) as a log with INFO level when the application starts.

In other words, by adding the `spring-boot-starter-security` dependency, your application is already secured.

ToDo App with Basic Security

Let's start with the ToDo app. Here, you use the same code as the JPA REST project; but I'll review the class once again. So let's begin. Starting from scratch, go to your browser and open Spring Initializr (https://start.spring.io). Add the following values to the fields:

- Group: `com.apress.todo`
- Artifact: `todo-simple-security`
- Name: `todo-simple-security`
- Package Name: `com.apress.todo`
- Dependencies: `Web, Security, Lombok, JPA, REST Repositories, H2, MySQL, Mustache`

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button; this downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 8-1).

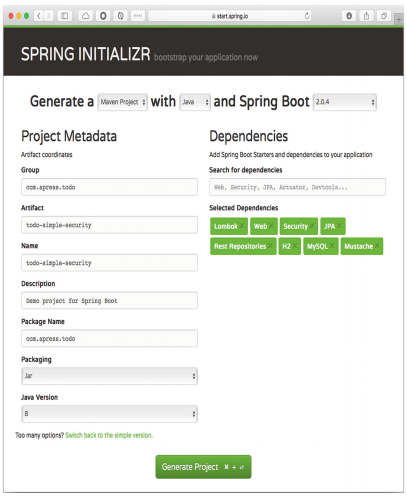

Figure 8-1 Spring Initializr

This project now has the Security module and a template engine, Mustache. Very soon you see how to use it.

Let's start with the ToDo domain class (see Listing 8-1).

```
package com.apress.todo.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy =
"uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){}
    public ToDo(String description){
        this.description = description;
    }

    @PrePersist
    void onCreate() {
```

```
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

Listing 8-1 com.apress.todo.domain.ToDo.java

Listing 8-1 shows the `ToDo` domain class. You already know about it. It's marked with `@Entity` and it's using `@Id` for a primary key. This class is from the *todo-rest* project.

Next, let's review the `ToDoRepository` interface (see Listing 8-2).

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import
org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends
CrudRepository<ToDo,String> {

}
```

Listing 8-2 com.apress.todo.repository.ToDoRepository.java

Listing 8-2 shows the `ToDoRepository`, and of course, you already know about it. Defining the interface that extends from the `CrudRepository<T,ID>` that has not only the CRUD methods, but also the Spring Data REST, creates all the necessary REST APIs to support the domain class.

Next, let's review the `application.properties` and see what is new (see Listing 8-3).

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# H2-Console: http://localhost:8080/h2-console
# jdbc:h2:mem:testdb
spring.h2.console.enabled=true

# REST API
spring.data.rest.base-path=/api
```

Listing 8-3 src/main/resources/application.properties

Listing 8-3 shows you the `application.properties` file. You've seen already some of the properties, except for the last one, right? The `spring.data.rest.base-path` tells the RestController (of the Spring Data REST configuration) that uses the `/api` as the root to expose all the REST API endpoints. So if we want to get ToDo's, we need to access the endpoint at `http://localhost:8080/api/toDos`.

Before running the app, let's add the endpoint in the form of a script. Create the `src/main/resources/data.sql` file with the following SQL statements.

```
insert into to_do
(id,description,created,modified,completed)
values ('8a8080a365481fb00165481fbca90000', 'Read a
Book','2018-08-17 07:42:44.136','2018-08-17
07:42:44.137',true);

insert into to_do
(id,description,created,modified,completed)
values ('ebcf1850563c4de3b56813a52a95e930', 'Buy Movie
Tickets','2018-08-17 09:50:10.126','2018-08-17
09:50:10.126',false);

insert into to_do
(id,description,created,modified,completed)
values ('78269087206d472c894f3075031d8d6b', 'Clean my
Room','2018-08-17 07:42:44.136','2018-08-17
07:42:44.137',false);
```

Now, if you run your application, you should see in the logs this output:

```
Using generated security password: 2a569843-122a-4559-
a245-60f5ab2b6c51
```

This is your password. You can now go to your browser and open `https://localhost:8080/api/toDos`. When you hit Enter to access that URL, you get something similar to Figure 8-2.

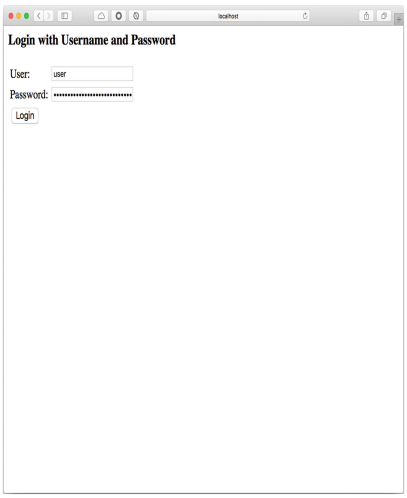

Figure 8-2 ToDo App: *http://localhost:8080/login* page

Figure 8-2 shows a login page, which is the default behavior when you add the `spring-boot-starter-security` dependency. By default, Security is on—so simple!! So, what is the user and password? Well, I mentioned this earlier, the user is `user`, and the password is the random one that was printed in the logs (in this example, `2a569843-122a-4559-a245-60f5ab2b6c51`). So, go ahead and enter the username and password; then you should get the ToDo's list (see Figure 8-3).

Figure 8-3 *http://localhost:80808/api/toDos*

If you want to try using the command line, you can execute the following command in a terminal window.

```
$ curl localhost:8080/api/toDos
{"timestamp":"2018-08-
19T21:25:47.224+0000","status":401,"error":"Unauthoriz
ed","message":"Unauthorized","path":"/api/toDos"}
```

As you can see from the output, you are not authorized to get into that endpoint. Authentication is needed, right? You can execute the following command.

```
$ curl localhost:8080/api/toDos -u user:2a569843-122a-
4559-a245-60f5ab2b6c51
{
  "_embedded" : {
    "toDos" : [ {
      "description" : "Read a Book",
      "created" : "2018-08-17T07:42:44.136",
      "modified" : "2018-08-17T07:42:44.137",
      "completed" : true,
...
}
```

As you can see now, you are passing the username and the random password, and you are getting the response with the list of ToDo's.

As probably you already know, every time you restart this app, the security auto-configuration generates another random password, and that's not optimal; maybe just for development.

OVERRIDING SIMPLE SECURITY

Random passwords don't do the trick in a production environment. Spring Boot Security allows you to override the defaults in multiple ways. The simplest is to override it with the `application.properties` file by adding the following `spring.security.*` properties .

```
spring.security.user.name=apress
spring.security.user.password=springboot2
spring.security.user.roles=ADMIN,USER
```

If you run the app again, the username is `apress` and the password is `spring-boot2` (the same as in a command line). Also notice that in the logs, the random password is no longer printed.

Another way is to provide authentication programmatically. Create a `ToDoSecurityConfig` class that extends from `WebSecurityConfigureAdapter`. Take a look at Listing 8-4.

```
package com.apress.todo.config;

import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authent
ication.builders.AuthenticationManagerBuilder;
import
org.springframework.security.config.annotation.web.con
figuration.WebSecurityConfigurerAdapter;
import
org.springframework.security.crypto.bcrypt.BCryptPassw
ordEncoder;

@Configuration
public class ToDoSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(
      AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()
                        .passwordEncoder(passwordEncoder
())
                        .withUser("apress")
                        .password(passwordEncoder().enco
de("springboot2"))
                        .roles("ADMIN","USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Listing 8-4 com.apress.todo.config.ToDoSecurityConfig.java

Listing 8-4 shows the necessary configuration for programmatically building the security, in this case, with one user (you can add more, of course). Let's analyze the code.

- `WebSecurityConfigurerAdapter`. Extending this class is one way to override security because it allows you to override the methods that you really need. In this case, the code overrides the `configure(AuthenticationManagerBuilder)` signature.
- `AuthenticationManagerBuilder`. This class creates an `AuthenticationManager` that allows you to easily build in memory, LDAP, JDBC authentications, `UserDetailsService` and add `AutheticationProvider`s. In this case, you are building an in-memory authentication. It's necessary to add a `PasswordEncoder` and a new and more secure way to use and encrypt/decrypt the password.
- `BCryptPasswordEncoder`. In this code you are using the `BCryptPasswordEncoder` (returns a `PasswordEncoder` implementation) that uses the BCrypt strong hashing function. You can use also `Pbkdf2PasswordEncoder` (uses PBKDF2 with a configurable number of iterations and a random 8-byte random salt value), or `SCryptPasswordEncoder` (uses the SCrypt hashing function). Even better, use `DelegatingPasswordEncoder`, which supports password upgrades.

Before you run the application, comment out the `spring.security.*` properties that you added to the `application.properties` file. If you run the app, it should work as expected. You need to provide the username, `apress`, and the password, `springboot2`.

OVERRIDING THE DEFAULT LOGIN PAGE

Spring Security allows you to override the default login page in several ways. One way is to configure `HttpSecurity`. The `HttpSecurity` class allows you to configure web-based security for specific HTTP requests. By default, it is applied to all requests, but can be restricted using `requestMatcher(RequestMatcher)` or similar methods.

Let's look at a modification of the `ToDoSecurityConfig` class (see Listing 8-5).

```
package com.apress.todo.config;

import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authent
ication.builders.AuthenticationManagerBuilder;
import
org.springframework.security.config.annotation.web.bui
lders.HttpSecurity;
import
org.springframework.security.config.annotation.web.con
figuration.WebSecurityConfigurerAdapter;
import
org.springframework.security.crypto.bcrypt.BCryptPassw
ordEncoder;

@Configuration
public class ToDoSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void
configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()
                .passwordEncoder(passwordEncoder())
                .withUser("apress")
                .password(passwordEncoder().encode("sp
ringboot2"))
                .roles("ADMIN","USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.authorizeRequests()
                .anyRequest().fullyAuthenticated()
                .and()
                .httpBasic();

    }
}
```

Listing 8-5 com.apress.todo.config.ToDoSecurityConfig.java – v2

Listing 8-5 shows version 2 of the `ToDoSecurityConfig` class. If you run the app and go to the browser (`http://localhost:8080/api/toDos`), you now get a pop-up for the basic authentication (see Figure 8-4).

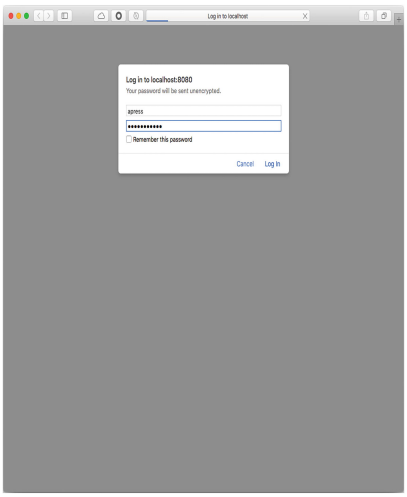

Figure 8-4 *http://localhost:8080/api/toDos*—Http Basic Authentication

You can use the username and password that you already know, and you should get the ToDo's list. It is the same for the command line. You need to authenticate

```
$ curl localhost:8080/api/toDos -u apress:springboot2
```

CUSTOM LOGIN PAGE

Normally in applications, you never see a page like that; typically, there is a very nice and well-designed login page, right? Spring Security allows you to create and customize your login page.

Let's prepare the ToDo app with a login page. First, we are going to add some CSS and the well-known jQuery library. Nowadays in a Spring Boot app, we can use WebJars dependencies. This new way avoids manually downloading the files; instead, you can use them as resources. Spring Boot web auto-configuration creates the necessary access for them.

If you are using Maven, open `pom.xml` and add the following dependencies.

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.3.7</version>
</dependency>

<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>3.2.1</version>
</dependency>
```

If you are using Gradle, open your `build.gradle` file and add the following dependencies.

```
compile ('org.webjars:bootstrap:3.3.7')
compile ('org.webjars:jquery:3.2.1')
```

Next, let's create the login page, which has the `.mustache` extension (`login.mustache`). It must be created in the `src/main/resources/templates` folder (see Listing 8-6).

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible"
content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1">
    <title>ToDo's API Login Page</title>
    <link
href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"
rel="stylesheet">
    <link href="css/signin.css" rel="stylesheet">
  </head>

  <body>

    <div class="container">
      <form class="form-signin" action="/login"
method="POST">
        <h2 class="form-signin-heading">Please sign
in</h2>

        <label for="username" class="sr-
only">Username</label>
        <input type="text"     name="username"
class="form-control" placeholder="Username" required
autofocus>

        <label for="inputPassword" class="sr-
only">Password</label>
        <input type="password" name="password"
class="form-control" placeholder="Password" required>

        <button class="btn btn-lg btn-primary btn-
block" id="login" type="submit">Sign in</button>
        <input type="hidden" name="_csrf" value="
{{_csrf.token}}" />
      </form>
    </div>
  </body>
</html>
```

Listing 8-6 src/main/resources/templates/login.mustache

Listing 8-6 shows the HTML login page. This page is using CSS from Bootstrap (`https://getbootstrap.com`) through the WebJars (`www.webjars.org` (http://www.webjars.org)) dependencies. These files are taken as file resources from those jars. HTML-FORM is using *username* and *password* as names (a must for Spring Security). We need to include the CSRF token to avoid any attacks. The Mustache engine provides this with the `{{_csrf.token}}` value. Spring Security uses the *synchronizer token pattern* to avoid any attacks in requests. Later on, we are going to see how we get this value.

Next, let's create an index page that lets you see the homepage and log out. Create the `index.mustache` page in the `src/main/resources/templates` folder (see Listing 8-7).

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible"
content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1">
    <title>ToDo's API</title>
```

```
    <link
href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"
rel="stylesheet">
    <script src="webjars/jquery/3.2.1/jquery.min.js">
</script>

</head>

<body>
<div class="container">
    <div class="header clearfix">
        <nav>
            <a href="#" id="logoutLink">Logout</a>
        </nav>
    </div>

    <div class="jumbotron">
        <h1>ToDo's Rest API</h1>
        <p class="lead">Welcome to the ToDo App. A
Spring Boot application!</p>
    </div>
</div>

<form id="logout" action="/logout" method="POST">
    <input type="hidden" name="_csrf" value="
{{_csrf.token}}" />
</form>
<script>
    $(function(){
        $('#logoutLink').click(function(){
            $('#logout').submit();
        });
    });
</script>
</body>
</html>
```

Listing 8-7 src/main/resources/templates/index.mustache

Listing 8-7 shows the index page. We are still using Bootstrap and the jQuery resources, and the most important part, the `{{_csrf.token}}`, for logout.

Next, let's start with the configuration. First, it is necessary to modify the `ToDoSecurityConfig` class (see Listing 8-8).

```
package com.apress.todo.config;

import
org.springframework.boot.autoconfigure.security.servle
t.PathRequest;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authent
ication.builders.AuthenticationManagerBuilder;
import
org.springframework.security.config.annotation.web.bui
lders.HttpSecurity;
import
org.springframework.security.config.annotation.web.con
figuration.WebSecurityConfigurerAdapter;
import
org.springframework.security.crypto.bcrypt.BCryptPassw
ordEncoder;
import
org.springframework.security.web.util.matcher.AntPathR
equestMatcher;

@Configuration
public class ToDoSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void
configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()
                .passwordEncoder(passwordEncoder())
                .withUser("apress")
                .password(passwordEncoder().encode("sp
ringboot2"))
                .roles("ADMIN","USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http
            .authorizeRequests()
                .requestMatchers(
                        PathRequest
```

```
                                .toStaticResources()
                                .atCommonLocations()).pe
rmitAll()
                .anyRequest().fullyAuthenticated()
                .and()
                .formLogin().loginPage("/login").permi
tAll()
                .and()
                .logout()
                    .logoutRequestMatcher(
                        new
AntPathRequestMatcher("/logout"))
                    .logoutSuccessUrl("/login");
    }

}
```

Listing 8-8 com.apress.todo.config.ToDoSecurityConfig.java – v3

Listing 8-8 shows version 3 of the `ToDoSecurityConfig` class. The new modification show how `HttpSecurity` is being configured. First, its adding `requestMatchers`, which point to common locations, such as the static resources (`static/*`). This is where CSS, JS, or any other simple HTML can live and doesn't need any security. Then it uses `anyRequest`, which should be `fullyAuthenticated`. this means that the `/api/*` will be. Then, it uses `formLogin` to specify with `loginPage("/login")` that it is the endpoint for finding the login page. Next, declare the logout and its endpoint (`"/logout"`); if the logout is successful, it redirects to the `"/login"` endpoint/page.

Now it is necessary to tell Spring MVC how to locate the login page. Create the `ToDoWebConfig` class (see Listing 8-9).

```
package com.apress.todo.config;

import
org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.View
ControllerRegistry;
import
org.springframework.web.servlet.config.annotation.WebM
vcConfigurer;

@Configuration
public class ToDoWebConfig implements WebMvcConfigurer
{

    @Override
    public void
addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewNa
me("login");
    }
}
```

Listing 8-9 com.apress.todo.config.ToDoWebConfig.java

Listing 8-9 shows a different way of configuring a web controller in Spring MVC. You can still use a class annotated with `@Controller` and create the mapping for the login page; but this is the JavaConfig way.

Here the class is implementing the `WebMvcConfigure` interface. It's implementing the `addViewControllers` method and registering the `/login` endpoint by telling the controller where the view is. This locates the `templates/login.mustache` page.

Finally, it is necessary to update the `application.properties` file by adding the following property.

```
spring.mustache.expose-request-attributes=true
```

Remember the `{{_csrf.token}}`? This is how it gets its value—by adding the `spring.mustache.expose-request-attributes` property.

Now, you can run the application. If you go to `http://localhost:8080`, you get something similar to Figure 8-5.

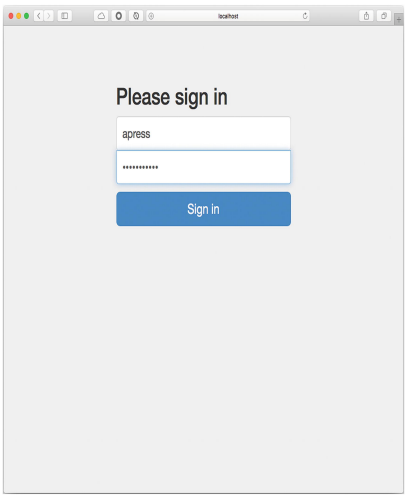

Figure 8-5 *http://localhost:8080/login*

You get the custom login page. Perfect!! Now you can enter the credentials, and it returns the index page (see Figure 8-6).

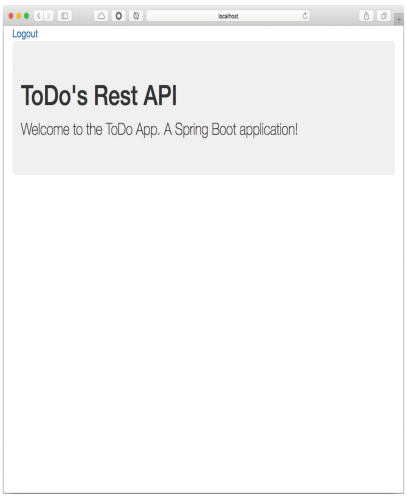

Figure 8-6 *http://localhost:8080* after login

Once you have the homepage, you can visit `http://localhost:8080/api/toDos`. You should be fully authenticated, and you can go back to the ToDo's list. You can go back to the homepage and press the Logout link, which redirects you to the `/login` endpoint again.

Now, what happens if you try to execute the following command line in a terminal window?

```
$ curl localhost:8080/api/toDos -u apress:springboot2
```

It won't return anything. It is an empty line. If you use the `-i` flag, it tells you that you are being redirected to `http://localhost:8080/login`. But there is no way to interact from the command line, right? So what can we do to fix this? In reality, there are clients that never use web interfaces. Most of the clients are apps and programmatically need to use the REST API, but with this solution, there is no way to do authentication to interact with a form.

Open the `ToDoSecurityConfig` class and modify the `configure(HttpSecurity)` method. It should look like the following snippet.

```
@Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.authorizeRequests()
                .requestMatchers(PathRequest.toStaticR
esources().atCommonLocations()).permitAll()
                .anyRequest().fullyAuthenticated()
                .and()
                .formLogin().loginPage("/login").permi
tAll()
                .and()
                .logout()
                    .logoutRequestMatcher(
                            new
AntPathRequestMatcher("/logout"))
                    .logoutSuccessUrl("/login")
                .and()
```

```
                .httpBasic();
    }
```

The last two lines of the method add the `httpBasic` call, which allows clients (like cURL) to use the basic authentication mechanisms. You can re-run the ToDo app and see that the executing the command line works now.

Using Security with JDBC

Imagine for a moment that your company already has an employee database, and you want to reuse it for authentication and authorization for the ToDo app. It is nice to integrate something like that, right?

Spring Security allows you to use AuthenticationManager with in-memory, LDAP and JDBC mechanisms. In this section, we are going to modify the ToDo app to run with JDBC.

DIRECTORY APP WITH JDBC SECURITY

In this section, you create a new app—a directory application where all the personnel are. The Directory app is integrated with the ToDo app to do the authentication and authorization. So, if a client needs to add a new ToDo, it needs to be authenticated with a `USER` role.

So let's begin. Starting from scratch, go to your browser and open Spring Initializr. Add the following values to the fields.

- Group: `com.apress.directory`
- Artifact: `directory`
- Name: `directory`
- Package Name: `com.apress.directory`
- Dependencies: `Web`, `Security`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 8-7).

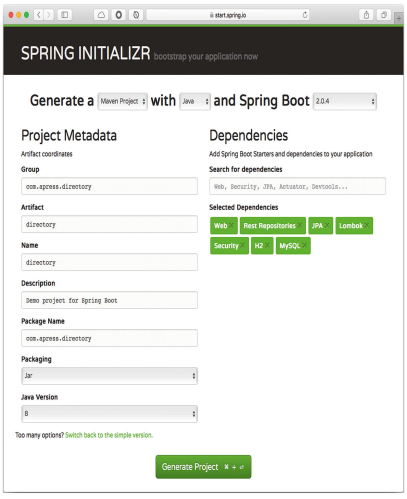

Figure 8-7 Spring Initializr

As you can see, the dependencies are very similar to other projects. We are going to use the power of Spring Data, Security, and REST. Let's start by adding a new class that holds a person's information. Create the `Person` class (see Listing 8-10).

```
package com.apress.directory.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
@Data
@Entity
public class Person {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy =
"uuid")
    private String id;
```

```
    @Column(unique = true)
    private String email;
    private String name;
    private String password;
    private String role = "USER";
    private boolean enabled = true;
    private LocalDate birthday;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;

    public Person() {
    }

    public Person(String email, String name, String
password, String birthday) {
        this.email = email;
        this.name = name;
        this.password = password;
        this.birthday = LocalDate.parse(birthday,
DateTimeFormatter.ofPattern("yyyy-MM-dd"));

    }

    public Person(String email, String name, String
password, LocalDate birthday) {
        this.email = email;
        this.name = name;
        this.password = password;
        this.birthday = birthday;
    }

    public Person(String email, String name, String
password, String birthday, String role, boolean
enabled) {
        this(email, name, password, birthday);
        this.role = role;
        this.enabled = enabled;
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

Listing 8-10 com.apress.directory.domain.Person.java

Listing 8-10 shows the `Person` class; very simple. It holds enough information about a person. Next, let's create the repository—the `PersonRepository` interface (see Listing 8-11).

```
package com.apress.directory.repository;

import com.apress.directory.domain.Person;
import
org.springframework.data.repository.CrudRepository;
import
org.springframework.data.repository.query.Param;

public interface PersonRepository extends
CrudRepository<Person,String> {
    public Person
findByEmailIgnoreCase(@Param("email") String email);
}
```

Listing 8-11 com.apress.directory.repository.PersonRepository.java

Listing 8-11 shows the `PersonRepository` interface; but what is different from the others? It declared a *query-method* `findByEmailIgnoreCase` with an email as the parameter (annotated by `@Param`). This syntax tells the Spring Data REST that it needs to implement these methods and create the SQL statement accordingly (this is based on the name and the fields in the domain class, in this case, the `email` field).

> **NOTE** If you want to learn more about how to define your own query-method, take a look at the Spring Data JPA Reference at https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.

Next, create the `DirectorySecurityConfig` class that extends from the `WebSecurityConfigurerAdapter` class. Remember that by extending from this class, we can customize the way Spring Security is set for this app (see Listing 8-12).

```
package com.apress.directory.config;

import
com.apress.directory.repository.PersonRepository;
```

```
import
com.apress.directory.security.DirectoryUserDetailsServ
ice;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authent
ication.builders.AuthenticationManagerBuilder;
import
org.springframework.security.config.annotation.web.bui
lders.HttpSecurity;
import
org.springframework.security.config.annotation.web.con
figuration.WebSecurityConfigurerAdapter;

@Configuration
public class DirectorySecurityConfig extends
WebSecurityConfigurerAdapter {

    private PersonRepository personRepository;

    public DirectorySecurityConfig(PersonRepository
personRepository){
        this.personRepository = personRepository;
    }

    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http
                .authorizeRequests()
                .antMatchers("/**").hasRole("ADMIN")
                .and()
                .httpBasic();
    }

    @Override
    public void configure(AuthenticationManagerBuilder
auth) throws Exception {
        auth.userDetailsService(
            new
DirectoryUserDetailsService(this.personRepository));
    }

}
```

Listing 8-12 com.apress.directory.config.DirectorySecurityConfig.java

Listing 8-12 shows the `DirectorySecurityConfig` class. This class is configuring `HttpSecurity` by allowing only users with an `ADMIN` role to any endpoint (`/**`) using basic authentication.

What else is different from other security configs? You are right! The `AuthenticationManager` is configuring a `UserDetailsService` implementation. This is the key to using any other third-party security app and integrating them with Spring Security.

As you can see, the `userDetailsService` method is using the `DirectoryUserDetailsService` class. Let's create it (see Listing 8-13).

```
package com.apress.directory.security;

import com.apress.directory.domain.Person;
import
com.apress.directory.repository.PersonRepository;
import
org.springframework.security.core.userdetails.User;
import
org.springframework.security.core.userdetails.UserDeta
ils;
import
org.springframework.security.core.userdetails.UserDeta
ilsService;
import
org.springframework.security.core.userdetails.Username
NotFoundException;
import
org.springframework.security.crypto.factory.PasswordEn
coderFactories;
import
org.springframework.security.crypto.password.PasswordE
ncoder;

public class DirectoryUserDetailsService implements
UserDetailsService {

    private PersonRepository repo;

    public
DirectoryUserDetailsService(PersonRepository repo) {
        this.repo = repo;
    }

    @Override
    public UserDetails loadUserByUsername(String
username) throws UsernameNotFoundException {
        try {
```

```
            final Person person =
this.repo.findByEmailIgnoreCase(username);

            if (person != null) {
                PasswordEncoder encoder =
PasswordEncoderFactories.createDelegatingPasswordEncod
er();
                String password =
encoder.encode(person.getPassword());

                return
User.withUsername(person.getEmail()).accountLocked(!pe
rson.isEnabled()).password(password).roles(person.getR
ole()).build();
            }
        }catch(Exception ex){
            ex.printStackTrace();
        }

        throw new UsernameNotFoundException(username);
    }
}
```

Listing 8-13 com.apress.directory.security.DirectoryUserDetailsService.java

Listing 8-13 shows the `DirectoryUserDetailsService` class. This class implements the `UserDetailsService` interface and needs to implement `loadUserByUserName` and return a `UserDetails` instance. In this implementation, the code is showing how the `PersonRepository` is being used. In this case, it uses `findByEmailIgnoreCase`; so, if a person is found with the email provided at the time the user wants to access `/**` (any endpoint), it compares the email vs. the password provided, the role, and if the account is locked or not, by creating a `UserDetails` instance.

This is amazing! This app is using JDBC as a mechanism for authentication. Again, you can plug in any other security system/app that can implement `UserDetailService` and return a `UserDetails` instance; that's it.

Next, let's quickly review the `application.properties` file and see its properties.

```
# Server
server.port=${port:8181}

# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# H2
spring.h2.console.enabled=true
```

The only difference is that it has the `server.port` property, which says: *If you provide the variable port (either command line, environment) I will use it; if not, I will use port 8181*. That's the `:`. This is part of the SpEL (Spring Expression Language).

Before running the Directory app, let's add some data. Create the `data.sql` file in the `src/main/resources` folder.

```
insert into person
(id,name,email,password,role,enabled,birthday,created,
modified)
values
('dc952d19ccfc4164b5eb0338d14a6619','Mark','mark@examp
le.com','secret','USER',true,'1960-03-29','2018-08-17
07:42:44.136','2018-08-17 07:42:44.137');

insert into person
(id,name,email,password,role,enabled,birthday,created,
modified)
values
('02288a3b194e49ceb1803f27be5df457','Matt','matt@examp
le.com','secret','USER',true,'1980-07-03','2018-08-17
07:42:44.136','2018-08-17 07:42:44.137');

insert into person
(id,name,email,password,role,enabled,birthday,created,
modified)
values
('4fe22e358d0e4e38b680eab91787f041','Mike','mike@examp
le.com','secret','ADMIN',true,'19820-08-05','2018-08-
17 07:42:44.136','2018-08-17 07:42:44.137');

insert into person
(id,name,email,password,role,enabled,birthday,created,
modified)
values
('84e6c4776dcc42369510c2692f129644','Dan','dan@example
.com','secret','ADMIN',false,'1976-10-11','2018-08-17
07:42:44.136','2018-08-17 07:42:44.137');

insert into person
(id,name,email,password,role,enabled,birthday,created,
modified)
values
('03a0c396acee4f6cb52e3964c0274495','Administrator','a
dmin@example.com','admin','ADMIN',true,'1978-12-
```

```
22','2018-08-17 07:42:44.136','2018-08-17
07:42:44.137');
```

Now we are ready to use this application as an authentication and authorization mechanism. Run the Directory application. This app starts in port 8181. You can test it using either the browser and/or `cURL` command .

```
$ curl
localhost:8181/persons/search/findByEmailIgnoreCase?
email=mark@example.com  -u admin@example.com:admin
{
  "email" : "mark@example.com",
  "name" : "Mark",
  "password" : "secret",
  "role" : "USER",
  "enabled" : true,
  "birthday" : "1960-03-29",
  "created" : "2018-08-17T07:42:44.136",
  "modified" : "2018-08-17T07:42:44.137",
  "_links" : {
    "self" : {
      "href" :
"http://localhost:8181/persons/dc952d19ccfc4164b5eb033
8d14a6619"
    },
    "person" : {
      "href" :
"http://localhost:8181/persons/dc952d19ccfc4164b5eb033
8d14a6619"
    }
  }
}
```

From the command, you are getting the user, Mark, by providing the username/password of a person with an `ADMIN` role; in this case, using the `-u` admin@example.com`:admin` parameter.

Great! You are using JDBC to look up users by using Spring Data REST and Spring Security! You can leave this project running.

USING THE DIRECTORY APP WITHIN THE TODO APP

It's time to integrate this Directory app with the ToDo app. And it is very easy.

Open your ToDo app and let's create a `Person` class. Yes, we are going to need a `Person` class that holds just enough information for authentication and authorization purposes. There is no need to have birth dates or any other information (see Listing 8-14).

```
package com.apress.todo.directory;

import
com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import lombok.Data;

@Data
@JsonIgnoreProperties(ignoreUnknown = true)
public class Person {

    private String email;
    private String password;
    private String role;
    private boolean enabled;
}
```

Listing 8-14 com.apress.todo.directory.Person.java

Listing 8-14 shows the `Person` class. This class only has the necessary fields for the authentication and authorization process. It is important to mention that calling the Directory app returns a more complete JSON object. It must match to do the deserialization (from JSON to object using the Jackson library), but because there is no need for extra information, this class is using the `@JasonIgnore-Properties(ignoreUnknown=true)` annotation that helps match the fields needed. I think this is a nice way to decouple classes.

> **NOTE** Some serialization tools in Java require the same class in the same package and implementing the `java.io.Serializable`, making it more difficult for developers and clients to manage and extend.

Next, create the `ToDoProperties` class that holds the information about the Directory app, like `Uri` (what is the address and base Uri), `Username`, and `Password` of the person that has the `ADMIN` role and has access to the REST API (see Listing 8-15).

```
package com.apress.todo.config;

import lombok.Data;
import
org.springframework.boot.context.properties.Configurat
ionProperties;

@Data
@ConfigurationProperties(prefix =
"todo.authentication")
```

```
public class ToDoProperties {

    private String findByEmailUri;
    private String username;
    private String password;

}
```

Listing 8-15 com.apress.todo.config.ToDoProperties.java

Listing 8-15 shows the `ToDoProperties` class; note that the prefix is `todo.authentication.*`. Next, modify the `ToDoSecurityConfig` class . You can comment the whole class and copy the code in Listing 8-16.

```
package com.apress.todo.config;

import com.apress.todo.directory.Person;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import
org.springframework.boot.autoconfigure.security.servle
t.PathRequest;
import
org.springframework.boot.context.properties.EnableConf
igurationProperties;
import
org.springframework.boot.web.client.RestTemplateBuilde
r;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.Resource;
import org.springframework.http.HttpStatus;
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.config.annotation.authent
ication.builders.AuthenticationManagerBuilder;
import
org.springframework.security.config.annotation.web.bui
lders.HttpSecurity;
import
org.springframework.security.config.annotation.web.con
figuration.WebSecurityConfigurerAdapter;
import
org.springframework.security.core.userdetails.User;
import
org.springframework.security.core.userdetails.UserDeta
ils;
import
org.springframework.security.core.userdetails.UserDeta
ilsService;
import
org.springframework.security.core.userdetails.Username
NotFoundException;
import
org.springframework.security.crypto.factory.PasswordEn
coderFactories;
import
org.springframework.security.crypto.password.PasswordE
ncoder;
import
org.springframework.security.web.util.matcher.AntPathR
equestMatcher;
import org.springframework.web.client.RestTemplate;
import
org.springframework.web.util.UriComponentsBuilder;
import java.net.URI;

@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoSecurityConfig extends
WebSecurityConfigurerAdapter {

    private final Logger log =
LoggerFactory.getLogger(ToDoSecurityConfig.class);

    //Use this to connect to the Directory App
    private RestTemplate restTemplate;
    private ToDoProperties toDoProperties;
    private UriComponentsBuilder builder;

    public ToDoSecurityConfig(RestTemplateBuilder
restTemplateBuilder, ToDoProperties toDoProperties){
        this.toDoProperties = toDoProperties;
        this.restTemplate =
restTemplateBuilder.basicAuthorization(toDoProperties.
getUsername(),toDoProperties.getPassword()).build();
    }

    @Override
    protected void
configure(AuthenticationManagerBuilder auth) throws
Exception {
```

```
        auth.userDetailsService(new
UserDetailsService(){

            @Override
            public UserDetails
loadUserByUsername(String username) throws
UsernameNotFoundException {

                try {
                    builder = UriComponentsBuilder
                  .fromUriString(toDoProperties.getFin
dByEmailUri())
                            .queryParam("email",
username);

                    log.info("Querying: " +
builder.toUriString());
                    ResponseEntity<Resource<Person>>
responseEntity =
                    restTemplate.exchange(
                                RequestEntity.get(UR
I.create(builder.toUriString()))
                                        .accept(Medi
aTypes.HAL_JSON)
                                        .build()
                            , new
ParameterizedTypeReference<Resource<Person>>() {
                            });

                    if (responseEntity.getStatusCode()
== HttpStatus.OK) {

                        Resource<Person> resource =
responseEntity.getBody();
                        Person person =
resource.getContent();

                        PasswordEncoder encoder =
                 PasswordEncoderFactories.createDelega
tingPasswordEncoder();
                        String password =
encoder.encode(person.getPassword());

                        return User
                  .withUsername(person.getEmail())
                  .password(password)
                  .accountLocked(!person.isEnabled())
                  .roles(person.getRole()).build();
                    }

                }catch(Exception ex) {
                    ex.printStackTrace();
                }
                throw new
UsernameNotFoundException(username);
            }
        });

    }

    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.authorizeRequests()
                .requestMatchers(PathRequest.toStaticR
esources().atCommonLocations()).permitAll()
                .antMatchers("/","/api/**").hasRole("U
SER")
                .and()
                .formLogin().loginPage("/login").permi
tAll()
                .and()
                .logout()
                .logoutRequestMatcher(new
AntPathRequestMatcher("/logout"))
                .logoutSuccessUrl("/login")
                .and()
                .httpBasic();
    }

}
```

Listing 8-16 com.apress.todo.config.ToDoSecurityConfig.java

Listing 8-16 shows the new `ToDoSecurityConfig` class. Let's analyze it.

- `WebSecurityConfigurerAdapter`. This class overrides what we need to customize the security for the app; but you already knew that, right?
- `RestTemplate`. This helper class makes a REST call to the Directory app endpoint, in particular `/persons/search/findByEmailIgnoreCase` Uri.
- `UriComponentsBuilder`. Remember that the `/persons/search/findByEmailIgnoreCase` endpoint needs a parameter (`email`); that's the one provided by the `loadUserByUsername` method (`username`).

- `AuthenticationBuilder`. The authentication provides `userDetailsService`. In this code, there is an anonymous implementation of the `UserDetailsService` and the implementation of the `loadUserByUsername` method. This is where the `RestTemplate` is being used to make a call to the Directory app and the endpoint.
- `ResponseEntity`. Because the Directory app response is `HAL+JSON`, it is necessary to use a `ResponseEntity` that manages all the resources from the protocol. If there is `HttpStatus.OK`, it is easy to get the content as a `Person` instance and use it to create `UserDetails`.
- `antMatchers`. This class is configuring `HttpSecurity` as before, but this time it is including an `antMatchers` method that exposes the endpoints that are accessed by a valid person with a `USER` role.

We are reusing the same technique from the Directory app. `AuthenticationManager` is configured to provide a `UserDetails` instance by calling the directory service using `RestTemplate`. The Directory app responded with a `HAL+JSON` protocol, which is why it is necessary to use `ResponseEntity` to get the person as a resource.

Next, append the following `todo.authentication.*` properties in the `application.properties` file .

```
# ToDo - Directory integration
todo.authentication.find-by-email-
uri=http://localhost:8181/persons/search/findByEmailIg
noreCase
todo.authentication.username=admin@example.com
todo.authentication.password=admin
```

It is necessary to specify the complete Uri that searches for the email endpoint, and the person that has the `ADMIN` role.

Now you are ready to use the ToDo app. You can use the browser or the command line. Make sure that the Directory app is up and running. Run the ToDo app that runs in port 8080.

You can execute the following command in a terminal window.

```
$ curl localhost:8080/api/toDos -u
mark@example.com:secret
{
  "_embedded" : {
    "toDos" : [ {
      "description" : "Read a Book",
      "created" : "2018-08-17T07:42:44.136",
      "modified" : "2018-08-17T07:42:44.137",
      "completed" : true,

...
...

"profile" : {
      "href" :
"http://localhost:8080/api/profile/toDos"
    }
  }
}
```

Now you are authenticating and authorizing with Mark, who has the `USER` role. Congrats!! You integrated your own JDBC service with the ToDo application.

WebFlux Security

To add security to a WebFlux application, nothing changes. You need to add the `spring-boot-starter-security` dependency, and Spring Boot takes care of the rest with its auto-configuration. If you want to customize it as we did before, the only thing you need to do is use `ReactiveUserDetailsService` (instead of `UserDetailsService`) or use `ReactiveAuthenticationManager` (instead of `AuthenticationManager`). Remember that now you are working with Mono and Flux reactive stream types.

ToDo App with OAuth2

With Spring Boot and Spring Security, OAuth2 is easier than ever. In this section of this chapter, we are going to enter directly into the ToDo app with OAuth2. I assume that you know about OAuth2 and all the benefits of using it as a mechanism for authentication with third-party providers—like Google, Facebook, and GitHub—directly into your app.

So let's begin. Starting from scratch, go to your browser and open Spring Initializr. Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-oauth2`
- Name: `todo-oauth2`
- Package Name: `com.apress.todo`

- Dependencies: `Web`, `Security`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button; this downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 8-8).

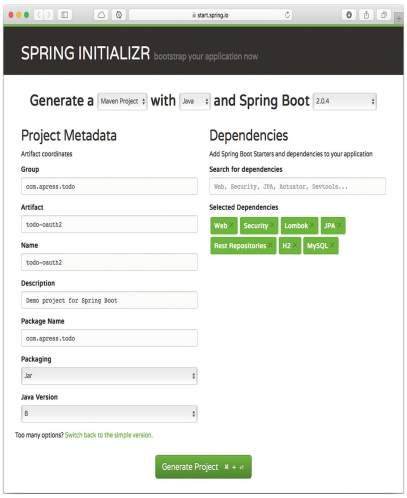

Figure 8-8 Spring Initializr

If you are using Maven, add the following dependencies to your `pom.xml` file .

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

If you are using Gradle, add the following dependencies to your `build.gradle` :

```
compile('org.springframework.security:spring-security-
oauth2-client')
compile('org.springframework.security:spring-security-
oauth2-jose')
```

As you can imagine, when Spring Boot sees the `spring-security-oauth2-client`, it auto-configures all the necessary beans to use the OAuth2 security for the app. It's important to mention the need for the `spring-security-oauth2-jose` that contains the Spring Security's support for JOSE (JavaScript Object Signing and Encryption) framework. The JOSE framework is intended to provide a method to securely transfer claims between parties. It is built from a collection of specifications: JSON Web Token (JWT), JSON Web Signature (JWS), JSON Web Encryption (JWE), and JSON Web Key (JWK).

Next, you can reuse the `ToDo` class and the `ToDoRepository` interface (see Listings 8-17 and 8-18).

```
package com.apress.todo.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy =
"uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;
    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){}
    public ToDo(String description){
        this.description = description;
```

```
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

Listing 8-17 com.apress.todo.domain.ToDo.java

As you can see nothing changed. It remains the same.

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import
org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends
CrudRepository<ToDo,String> { }
```

Listing 8-18 com.apress.todo.repository.ToDoRepository.java

The same for this interface—nothing changed. Let's review the `application.properties`.

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# H2-Console: http://localhost:8080/h2-console
# jdbc:h2:mem:testdb
spring.h2.console.enabled=true
```

Nothing changed. Well, we are going to add more properties very soon.

Now comes the important part. You are going to use GitHub for OAuth2 authentication for the ToDo app.

CREATING THE TODO APP IN GITHUB

I'm assuming that you probably already have a GitHub account; if not, you can open a new one very easily at `https://github.com`. You can log in to your account and then open `https://github.com/settings/applications/new`. That's where you create the application. You can use the following values.

- Application name: todo-app
- Homepage URL: http://localhost:8080
- Application description: ToDo App
- Authorization callback URL: http://localhost:8080/login/oauth2/code/github

It's important to the authorization callback URL because this is how Spring Security's `OAuth2LoginAuthenticationFilter` expects to work with this endpoint pattern: `/login/oauth2/code/*`; of course, it is customizable by using the `redirect-uri-template` property (see Figure 8-9).

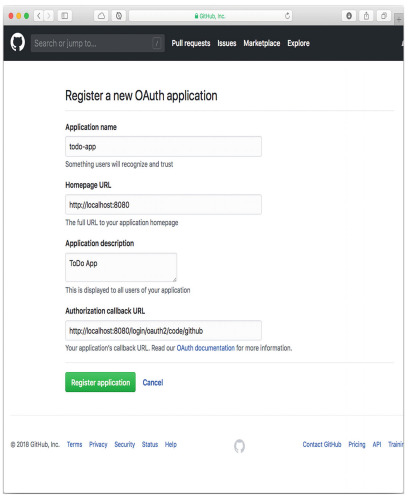

Figure 8-9 GitHub new app: `https://github.com/settings/applications/new`

You can click the Register application button. After this, GitHub creates the keys you need in your application (see Figure 8-10).

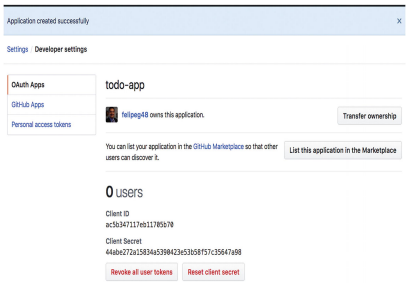

Figure 8-10 Client ID and client secret keys

Once you have this, copy the client id and client secret keys and append them to the `application.properties` with the `spring.security.oauth2.client.registration.*` keys.

```
# OAuth2
spring.security.oauth2.client.registration.todo.client
-id=ac5b347117eb11705b70
spring.security.oauth2.client.registration.todo.client
-secret=44abe272a15834a5390423e53b58f57c35647a98
spring.security.oauth2.client.registration.todo.client
-name=ToDo App with GitHub Authentication
spring.security.oauth2.client.registration.todo.provid
er=github
spring.security.oauth2.client.registration.todo.scope=
user
spring.security.oauth2.client.registration.todo.redire
ct-uri-
template=http://localhost:8080/login/oauth2/code/githu
b
```

The `spring.security.oauth2.client.registration` accepts a map that contains the necessary keys like the `client-id` and `client-secret`.

That's it!! You don't need anything else. You can now run your application. Open the browser and point to http://localhost:8080. You get a link that redirects you to GitHub (see Figure 8-11).

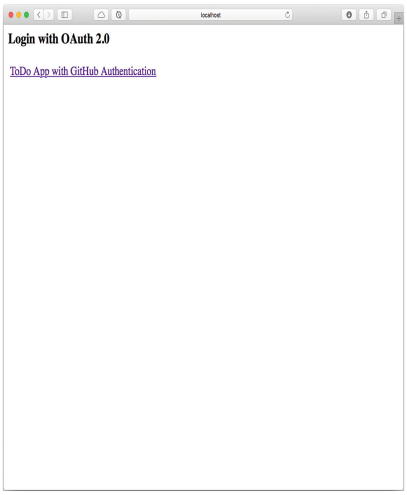

Figure 8-11 *http://localhost:8080*

You can click the link, which gets you through the login process but using a GitHub authentication mechanism (see Figure 8-12).

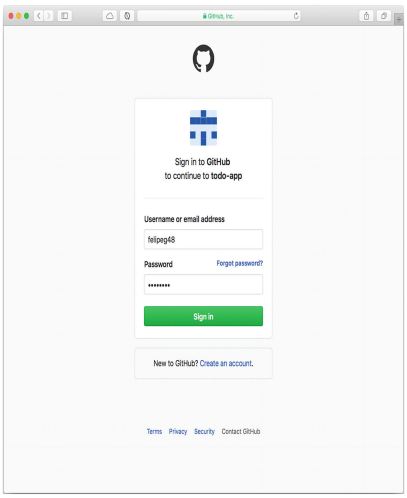

Figure 8-12 GitHub authentication

You can log in now with your credentials. Next, you are redirected to another page where you need to give permissions to the *todo-app* to use contact information (see Figure 8-13).

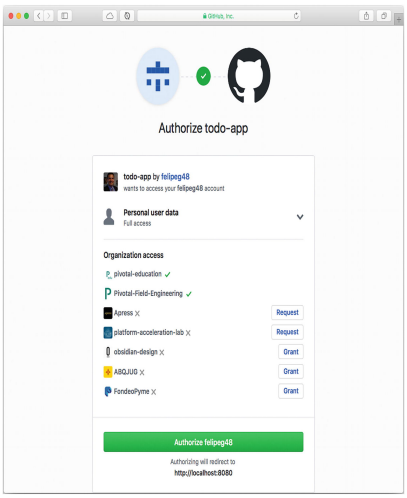

Figure 8-13 GitHub authorization process

You can then click the Authorize button to get back to your app with the ToDo REST API (see Figure 8-14).

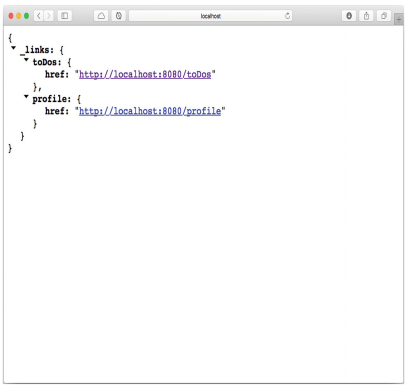

Figure 8-14 After GitHub authorization process

Congratulations!! Now you know how easy it is to integrate OAuth2 with different providers using Spring Boot and Spring Security.

NOTE You can find the solution to this section in the book source code on the Apress website or on GitHub at https://github.com/Apress/pro-spring-boot-2 , or on my personal repository at https://github.com/felipeg48/pro-spring-boot-2nd .

Summary

In this chapter, you learned different ways to do security with Spring Boot. You learned how easy it is to secure an application by adding the `spring-boot-security-starter` dependency.

You also learned that it is easy to customize and override the defaults that Spring Boot offers you with Spring Security. You can use the `spring.security.*` properties or you can customize it with the `WebSecurityConfigurerAdapter` class.

You learned how to use JDBC and connect two applications, one of them acting as a security authority for authentication and authorization.

Lastly, you learned how easy it is to use OAuth2 with third-party authentication and authorization providers like Facebook, Google, GitHub, and more.

In the next chapter, we start working with messaging brokers.

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
7. Testing with Spring Boot

NEXT
9. Messaging with Spring Boot