Chapter 10. Securing Your Spring Boot Application

Understanding the concepts of authentication and authorization are critical to building secure applications, providing the foundations for user verification and access control. Spring Security combines options for authentication and authorization with other mechanisms like the HTTP Firewall, filter chains, extensive use of IETF and the World Wide Web Consortium (W3C) standards and options for exchanges, and more to help lock down applications. Adopting a secure out-of-the-box mindset, Spring Security leverages Boot's powerful autoconfiguration to evaluate developer inputs and available dependencies to deliver maximal security for Spring Boot applications with minimal effort.

This chapter introduces and explains core aspects of security and how they apply to applications. I demonstrate multiple ways to incorporate Spring Security into Spring Boot apps to strengthen an application's security posture, closing dangerous gaps in coverage and reducing attack surface area.

CODE CHECKOUT CHECKUP

Please check out branch *chapter10begin* from the code repository to begin.

Authentication and Authorization

Often used together, the terms *authentication* and *authorization* are related but separate concerns.

authentication

> An act, process, or method of showing something (such as an identity, a piece of art, or a financial transaction) to be real, true, or genuine; the act or process of authenticating something.

authorization

> 1: the act of *authorizing* 2: an instrument that authorizes: SANCTION

The first definition for *authorization* points to *authorizing* for more information:

authorize

> 1: to endorse, empower, justify, or permit by or as if by some recognized or proper authority (such as custom, evidence, personal right, or regulating power) a custom authorized by time 2: to invest especially with legal authority: EMPOWER 3: archaic: JUSTIFY

The definition for *authorize* in turn points to *justify* for more information.

While somewhat interesting, these definitions aren't very clear. Sometimes the dictionary definitions can be less helpful than we might like. My own definitions follow.

authentication

> Proving that someone is who they claim to be

authorization

> Verifying that someone has access to a particular resource or operation

Authentication

Simply put, *authentication* is proving that someone (or something) is who (or what, in the case of a device, application, or service) they claim to be.

The concept of authentication has several concrete examples in the physical world. If you've ever had to show a form of ID like an employee badge, driver's license, or passport to prove your identity, you have been authenticated. Demonstrating that one is who one claims to be is a procedure to which we've all grown accustomed in a variety of situations, and the conceptual differences between authentication at the physical level and to an application are insignificant.

Authentication typically involves one or more of the following:

- Something you are
- Something you know
- Something you have

NOTE

These three *factors* can be used individually or combined to compose Multi-Factor Authentication (MFA).

The manner in which authentication occurs in the physical and virtual worlds is of course different. Rather than a human being eyeing a photo

ID and comparing it to your current physical appearance as often happens in the physical world, authenticating to an application often involves typing a password, inserting a security key, or providing biometric data (iris scan, fingerprint, etc.) that can be more easily evaluated by software than is currently feasible with a comparison of physical appearance with a photo. Nevertheless, a comparison of stored data with provided data is performed in both cases, and a match provides a positive authentication.

Authorization

Once a person is authenticated, they have the possibility of gaining access to resources available and/or operations permitted to one or more individuals.

NOTE

In this context, an individual may (and most likely is) a human being, but the same concept and access considerations apply to applications, services, devices, and more, depending on context.

Once an individual's identity is proven, that individual gains some general level of access to an application. From there, the now-authenticated application user can request access to something. The application then must somehow determine if the user is allowed, i.e., *authorized*, to access that resource. If so, access is granted to the user; if not, the user is notified that their lack of *authority* has resulted in their request being rejected.

Spring Security in a Nutshell

In addition to providing solid options for authentication and authorization, Spring Security provides several other mechanisms to help developers lock down their Spring Boot applications. Thanks to autoconfiguration, Spring Boot applications enable each applicable Spring Security feature with an eye toward maximum possible security with the information provided, or even owing to the lack of more specific guidance. Security capabilities can of course be adjusted or relaxed by developers as necessary to accommodate their organizations' specific requirements.

Spring Security's capabilities are far too numerous to detail exhaustively in this chapter, but there are three key features I consider essential to understanding the Spring Security model and its foundations. They are the HTTP Firewall, security filter chains, and Spring Security's extensive use

of IETF and W3C standards and options for requests and corresponding responses.

The HTTP Firewall

While exact numbers are difficult to obtain, many security compromises begin with a request using a malformed URI and a system's unexpected response to it. This is really an application's first line of defense, and as such, it is the problem that should be solved prior to considering further efforts to secure one's application(s).

Since version 5.0, Spring Security has included a built-in HTTP Firewall that scrutinizes all inbound requests for problematic formatting. If there are any problems with a request, such as bad header values or incorrect formatting, the request is discarded. Unless overridden by the developer, the default implementation used is the aptly named `StrictHttpFirewall`, quickly closing the first and potentially easiest gap to exploit within an application's security profile.

Security Filter Chains

Providing a more specific, next-level filter for inbound requests, Spring Security uses filter chains to process properly formed requests that make it past the HTTP Firewall.

Simply put, for most applications a developer specifies a chain of filter conditions through which an inbound request passes until it matches one. When a request matches a filter, its corresponding conditions are evaluated to determine if the request will be fulfilled. For example, if a request for a particular API endpoint arrives and matches a filter condition in the filter chain, the user who made the request may be checked to verify they have the proper role/authority to access the requested resource. If so, the request is processed; if not, it is rejected, usually with a *403 Forbidden* status code.

If a request passes through all defined filters in the chain without matching any, the request is discarded.

Request and Response Headers

The IETF and W3C have created a number of specifications and standards for HTTP-based exchanges, several of which relate to the secure exchange of information. There are several headers defined for interactions between user agents—command line utilities, web browsers, etc.—and server or cloud-based applications/services. These headers are used to request or signal specific behavior and have defined allowed values and be-

havioral responses, and Spring Security makes extensive use of these header details to strengthen your Spring Boot application's security posture.

Realizing it's true that different user agents may support some or all of these standards and specifications, and even then fully or partially, Spring Security embraces a best-possible-coverage approach by checking for all known header options and applying them across the board, looking for them in requests and supplying them in responses, as applicable.

Implementing Forms-Based Authentication and Authorization with Spring Security

Innumerable applications that use the "something you know" method of authentication are used every day. Whether for apps internal to an organization, web applications provided directly to consumers via the internet, or apps native to a mobile device, typing in a user ID and password is a familiar routine for developers and non-developers alike. And in most of those cases, the security this provides is more than sufficient for the task at hand.

Spring Security provides Spring Boot applications with superb out-of-the-box (OOTB) support for password authentication via autoconfiguration and easy-to-grasp abstractions. This section demonstrates the various stepping-off points by refactoring the `Aircraft Positions` application to incorporate forms-based authentication using Spring Security.

Adding Spring Security Dependencies

When creating a new Spring Boot project, it's a simple matter to add one more dependency via the Spring Initializr—that of *Spring Security*—and to enable a top-tier level of security without additional configuration to a fledgling app, as shown in Figure 10-1.

Figure 10-1. Spring Security dependency within Spring Initializr

Updating an existing application is only slightly less simple. I'll add the same two complementary dependencies that Initializr adds, one for

Spring Security itself and one for testing it, to `Aircraft Positions`'s *pom.xml* Maven build file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

NOTES ABOUT TEST COVERAGE

In order to focus specifically on the key concepts of Spring Security in this chapter, I will somewhat reluctantly set aside the tests that have been and would normally be created when adding additional capabilities. This decision is solely an information-sharing decision to streamline chapter content and is not a development process decision.

To continue with builds in this and subsequent chapters, it may be necessary to add `-DskipTests` if building from the command line or to be sure to select the application's configuration (rather than a test) from a drop-down menu if building from an IDE.

With Spring Security on the classpath and no code or configuration changes to the application, I restart `Aircraft Positions` for a quick functionality check. This provides a great opportunity to see what Spring Security does on a developer's behalf OOTB.

With both PlaneFinder and `Aircraft Positions` running, I return to the terminal and again exercise `Aircraft Positions`'s */aircraft* endpoint, as shown here:

```
mheckler-a01 :: ~ » http :8080/aircraft
HTTP/1.1 401
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Expires: 0
Pragma: no-cache
Set-Cookie: JSESSIONID=347DD039FE008DE50F457B890F2149C0; Path=/; HttpOnly
WWW-Authenticate: Basic realm="Realm"
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

{
```

```
    "error": "Unauthorized",
    "message": "",
    "path": "/aircraft",
    "status": 401,
    "timestamp": "2020-10-10T17:26:31.599+00:00"
}
```

NOTE

Some response headers have been removed for clarity.

As you can see, I no longer have access to the */aircraft* endpoint, receiving a *401 Unauthorized* response to my request. Since the */aircraft* endpoint is currently the only means of accessing information from the `Aircraft Positions` application, this effectively means the application is secured in its entirety from unwanted access. This is great news, but it's important to understand both how this happened and how to restore desired access for valid users.

As I mentioned earlier, Spring Security adopts a mindset of "secure by default" to the extent possible at every level of configuration—even zero configuration—by the developer employing it in a Spring Boot application. When Spring Boot finds Spring Security on the classpath, security is configured using sensible defaults. Even with no user(s) defined or password(s) specified or any other effort made by the developer, the inclusion of Spring Security in the project indicates a goal of a secure application.

As you can imagine, this is very little to go on. But Spring Boot+Security autoconfiguration creates a number of essential beans to implement basic security capabilities based on forms authentication and user authorization using user IDs and passwords. The next questions to quite reasonably flow from that logical assumption are these: What user(s)? What password(s)?

Returning to the startup log for the `Aircraft Positions` application, one can find the answer to one of those questions in this line:

```
Using generated security password: 1ad8a0fc-1a0c-429e-8ed7-ba0e3c3649ef
```

If no user ID(s) and password(s) are specified in the application or no means are provided to access them by other means, a security-enabled Spring Boot app defaults to a single user account `user` with a unique password that is generated anew each time the application is started. Returning to the terminal window, I try to access the application once again, this time with the provided credentials:

```
mheckler-a01 :: ~ » http :8080/aircraft
    --auth user:1ad8a0fc-1a0c-429e-8ed7-ba0e3c3649ef
HTTP/1.1 200
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Expires: 0
Pragma: no-cache
Set-Cookie: JSESSIONID=94B52FD39656A17A015BC64CF6BF7475; Path=/; HttpOnly
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

[
    {
        "altitude": 40000,
        "barometer": 1013.6,
        "bds40_seen_time": "2020-10-10T17:48:02Z",
        "callsign": "SWA2057",
        "category": "A3",
        "flightno": "WN2057",
        "heading": 243,
        "id": 1,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-10T17:48:06Z",
        "lat": 38.600372,
        "lon": -90.42375,
        "polar_bearing": 207.896382,
        "polar_distance": 24.140226,
        "pos_update_time": "2020-10-10T17:48:06Z",
        "reg": "N557WN",
        "route": "IND-DAL-MCO",
        "selected_altitude": 40000,
        "speed": 395,
        "squawk": "2161",
        "type": "B737",
        "vert_rate": -64
    },
    {
        "altitude": 3500,
        "barometer": 0.0,
        "bds40_seen_time": null,
        "callsign": "N6884J",
        "category": "A1",
        "flightno": "",
        "heading": 353,
        "id": 2,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-10T17:47:45Z",
        "lat": 39.062851,
        "lon": -90.084965,
        "polar_bearing": 32.218696,
        "polar_distance": 7.816637,
```

```
        "pos_update_time": "2020-10-10T17:47:45Z",
        "reg": "N6884J",
        "route": "",
        "selected_altitude": 0,
        "speed": 111,
        "squawk": "1200",
        "type": "P28A",
        "vert_rate": -64
    },
    {
        "altitude": 39000,
        "barometer": 0.0,
        "bds40_seen_time": null,
        "callsign": "ATN3425",
        "category": "A5",
        "flightno": "",
        "heading": 53,
        "id": 3,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-10T17:48:06Z",
        "lat": 39.424159,
        "lon": -90.419739,
        "polar_bearing": 337.033437,
        "polar_distance": 30.505314,
        "pos_update_time": "2020-10-10T17:48:06Z",
        "reg": "N419AZ",
        "route": "AFW-ABE",
        "selected_altitude": 0,
        "speed": 524,
        "squawk": "2224",
        "type": "B763",
        "vert_rate": 0
    },
    {
        "altitude": 45000,
        "barometer": 1012.8,
        "bds40_seen_time": "2020-10-10T17:48:06Z",
        "callsign": null,
        "category": "A2",
        "flightno": "",
        "heading": 91,
        "id": 4,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-10T17:48:06Z",
        "lat": 39.433982,
        "lon": -90.50061,
        "polar_bearing": 331.287125,
        "polar_distance": 32.622134,
        "pos_update_time": "2020-10-10T17:48:05Z",
        "reg": "N30GD",
        "route": "",
```

```
            "selected_altitude": 44992,
            "speed": 521,
            "squawk": null,
            "type": "GLF4",
            "vert_rate": 64
        }
    ]
```

NOTE

As before, some response headers have been removed for clarity.

Using the correct default user ID and the generated password, I receive a *200 OK* response and once again have access to the */aircraft* endpoint, and thus the `Aircraft Positions` application.

IMPORTANT INFORMATION ABOUT RESPONSE HEADERS

As I mentioned very briefly earlier, several IETF and W3C standard header options have been formalized and/or recommended for use by browsers and other user agents to improve application security. Spring Security adopts and implements these rigorously in an effort to provide the most complete security coverage possible using every available means.

Spring Security's response header defaults that comply with these standards and recommendations include the following:

Cache Control:: The `Cache-Control` header is set to `no-cache` with a `no-store` directive and a `max-age` of 0 and `must-revalidate` directive; additionally, a `Pragma` header is returned with a `no-cache` directive, and an `Expires` header is given a 0 value. All of these mechanisms are specified to eliminate possible gaps in browser/user agent feature coverage to ensure best-possible control over caching, i.e., to disable it in all cases so that once a user logs out of a site, a hostile actor can't simply click the Back button of the browser and return to the secure site logged in with the victim's credentials. Content Type Options:: The `X-Content-Type-Options` header is set to `nosniff` to disable content sniffing. Browsers can (and often have) attempted to "sniff" the type of content requested and display it accordingly. For example, if a .jpg is requested, a browser might render it as a graphical image. This sounds like it could be a nice feature, and indeed it can be; but if malicious code is embedded within the sniffed content, it can be processed surreptitiously, bypassing otherwise rigorous security measures. Spring Security provides a setting of `nosniff` by default, closing this attack vector. Frame Options:: The `X-Frame-Options` header is set to a value of `DENY` in order to prevent browsers from displaying content within iframes. Dubbed *clickjacking*, the resultant attack can occur when an invisible frame is placed over a displayed control, resulting in the user initiating an undesired action instead of the one intended, thus "hijacking" the user's click. Spring Security disables frame support by default, thus closing the avenue to clickjacking attacks. XSS Protection:: The `X-XSS-Protection` header is set to a value of 1 to enable browser protection against Cross Site Scripting (XSS) attacks. Once enabled however, there are many ways for a browser to respond to perceived attacks; Spring Security defaults to a setting of `mode=block`, the most secure setting, to assure that a browser's well-meaning attempt to modify and process the content safely doesn't leave the user vulnerable. Blocking the content closes that potential vulnerability.

Note that if you use a content delivery network, you may need to adjust the XSS settings to ensure correct handling. That setting, like the others, is fully developer-configurable. In the absence of specific directions from

you, the developer, Spring Security will always strive to adopt a *secure by default* posture to the greatest extent possible given the information available.

Returning to the `Aircraft Positions` application, there are a few concerns with the current state of application security. First among them is that by having only a single defined user, multiple individuals that need to access the application must all use that single account. This is antithetical to the security principle of accountability and even authentication, as no single individual uniquely proves they are who they say they are. Returning to accountability, how can one determine who committed or contributed to a breach if one occurs? Not to mention that if a breach were to occur, locking the only user account would disable access for all users; there is currently no way to avoid it.

A secondary concern with the existing security configuration is the way the single password is handled. With each application startup, a new password is automatically generated, which must then be shared with all users. And while application scaling hasn't yet been discussed, each instance of `Aircraft Positions` that is started will generate a unique password, requiring that particular password from a user attempting to log into that particular app instance. Clearly some improvements can and should be made.

Adding Authentication

Spring Security employs the concept of a `UserDetailsService` as the centerpiece of its authentication capability. `UserDetailsService` is an interface with a single method `loadUserByUsername(String username)` that (when implemented) returns an object that fulfills the `UserDetails` interface, from which can be obtained key information such as the user's name, password, authorities granted to the user, and account status. This flexibility allows for numerous implementations using various technologies; as long as a `UserDetailsService` returns `UserDetails`, the application doesn't need to be aware of underlying implementation details.

To create a `UserDetailsService` bean, I create a configuration class in which to define a bean creation method.

BEAN CREATION

While bean creation methods can be placed within any `@Configuration`-annotated class (as mentioned previously), including those with meta-annotations that include `@Configuration` like the main application class, it is cleaner to create a configuration class for related groups of beans. If the application is small and the number of beans is as well, this could be a single class.

With rare exceptions for the smallest and most disposable of applications, I typically do not place bean creation methods in the `@SpringBootApplication`-annotated main application class. Placing them in separate class(es) simplifies testing by reducing the number of beans that must be created or mocked, and for the occasional circumstance in which a developer may wish to deactivate a group of beans—such as an `@Component` or `@Configuration` class that loads data or performs similar function(s)—removing or commenting out the single class-level annotation disables that functionality while leaving the capability easily accessible. While "comment and keep" should not be used with abandon, it makes a great deal of sense in narrowly defined circumstances.

This separate class will come in handy for the next iteration as well, making it easier and faster to iterate due to the natural separation of concerns.

First, I create a class called `SecurityConfig` and annotate it with `@Configuration` to enable Spring Boot to find and execute bean creation methods within. The bean necessary for authentication is one that implements the `UserDetailsService` interface, so I create a method `authentication()` to create and return that bean. Here is a first, intentionally incomplete pass at the code:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
public class SecurityConfig {
    @Bean
    UserDetailsService authentication() {
        UserDetails peter = User.builder()
                .username("peter")
                .password("ppassword")
```

```
                .roles("USER")
                .build();

        UserDetails jodie = User.builder()
                .username("jodie")
                .password("jpassword")
                .roles("USER", "ADMIN")
                .build();

        System.out.println("   >>> Peter's password: " + peter.getPassword())
        System.out.println("   >>> Jodie's password: " + jodie.getPassword())

        return new InMemoryUserDetailsManager(peter, jodie);
    }
}
```

Within the `UserDetailService` `authentication()` method, I create two application objects that implement the `UserDetails` interface requirements using the `User` class's `builder()` method and specifying username, password, and roles/authorities the user possesses. I then `build()` these users and assign each of them to a local variable.

Next, I display the passwords *for demonstration purposes only*. This helps to demonstrate another concept in this chapter but is *for demonstration purposes only*.

WARNING

Logging passwords is an antipattern of the worst kind. Never log passwords in production applications.

Finally, I create an `InMemoryUserDetailsManager` using the two created `User` objects and return it as a Spring bean. An `InMemoryUserDetailsManager` implements interfaces `UserDetailsManager` and `UserDetailsPasswordService`, enabling user management tasks like determining if a particular user exists; creating, updating, and deleting the user; and changing/updating the user's password. I use `InMemoryUserDetailsManager` for its clarity in demonstrating the concept (due to no external dependencies), but any bean that implements the `UserDetailsService` interface can be provided as the authentication bean.

Restarting `Aircraft Positions`, I attempt to authenticate and retrieve a list of current aircraft positions, with the following results (some headers removed for brevity):

```
mheckler-a01 :: ~ » http :8080/aircraft --auth jodie:jpassword
HTTP/1.1 401
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Length: 0
Expires: 0
Pragma: no-cache
WWW-Authenticate: Basic realm="Realm"
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

This prompts a bit of troubleshooting. Returning to the IDE, there is a helpful bit of information in the stack trace:

```
java.lang.IllegalArgumentException: There is no PasswordEncoder
    mapped for the id "null"
        at org.springframework.security.crypto.password
        .DelegatingPasswordEncoder$UnmappedIdPasswordEncoder
            .matches(DelegatingPasswordEncoder.java:250)
                ~[spring-security-core-5.3.4.RELEASE.jar:5.3.4.RELEASE]
```

This provides a hint at the root of the problem. Examining the logged passwords (kind reminder: logging passwords is *for demonstration purposes only*) provides confirmation:

```
>>> Peter's password: ppassword
>>> Jodie's password: jpassword
```

Clearly these passwords are in plain text, with no encoding whatsoever being performed. The next step toward working and secure authentication is to add a password encoder for use within the `SecurityConfig` class, as shown in the following:

```
private final PasswordEncoder pwEncoder =
        PasswordEncoderFactories.createDelegatingPasswordEncoder();
```

One of the challenges of creating and maintaining secure applications is that, of necessity, security is ever evolving. Recognizing this, Spring Security doesn't simply have a designated encoder it plugs in; rather, it uses a factory with several available encoders and delegates to one for encoding and decoding chores.

Of course, this means that one must serve as a default in the event one isn't specified, as in the previous example. Currently *BCrypt* is the (excellent) default, but the flexible, delegated nature of Spring Security's encoder architecture is such that one encoder can be replaced easily by an-

other as standards evolve and/or requirements change. The elegance of this approach allows for a frictionless migration of credentials from one encoder to another when an application user logs into the application, again reducing tasks that don't directly provide value to an organization but are nevertheless critical to perform in a correct and timely manner.

Now that I have an encoder in place, the next step is to use it to encrypt the user passwords. This is done very simply by plugging in a call to the password encoder's `encode()` method, passing the plain text password and receiving in return the encrypted result.

TIP

Technically speaking, encrypting a value also encodes that value, but all encoders do not encrypt. For example, hashing encodes a value but does not necessarily encrypt it. That said, every encoding algorithm supported by Spring Security also encrypts; to support legacy applications, however, some supported algorithms are far less secure than others. Always choose a current recommended Spring Security encoder or opt for the default one provided by `PasswordEncoderFactories.createDelegatingPasswordEncoder()`.

The revised, authenticating version of the `SecurityConfig` class follows:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
public class SecurityConfig {
    private final PasswordEncoder pwEncoder =
            PasswordEncoderFactories.createDelegatingPasswordEncoder();

    @Bean
    UserDetailsService authentication() {
        UserDetails peter = User.builder()
                .username("peter")
                .password(pwEncoder.encode("ppassword"))
                .roles("USER")
                .build();

        UserDetails jodie = User.builder()
                .username("jodie")
                .password(pwEncoder.encode("jpassword"))
```

```
                .roles("USER", "ADMIN")
                .build();

        System.out.println("   >>> Peter's password: " + peter.getPassword());
        System.out.println("   >>> Jodie's password: " + jodie.getPassword());

        return new InMemoryUserDetailsManager(peter, jodie);
    }
}
```

I restart `Aircraft Positions`, then attempt once again to authenticate and retrieve a list of current aircraft positions with the following results (some headers and results removed for brevity):

```
mheckler-a01 :: ~ » http :8080/aircraft --auth jodie:jpassword
HTTP/1.1 200
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Expires: 0
Pragma: no-cache
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

[
    {
        "altitude": 24250,
        "barometer": 0.0,
        "bds40_seen_time": null,
        "callsign": null,
        "category": "A2",
        "flightno": "",
        "heading": 118,
        "id": 1,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-12T16:13:26Z",
        "lat": 38.325119,
        "lon": -90.154159,
        "polar_bearing": 178.56009,
        "polar_distance": 37.661127,
        "pos_update_time": "2020-10-12T16:13:24Z",
        "reg": "N168ZZ",
        "route": "FMY-SUS",
        "selected_altitude": 0,
        "speed": 404,
        "squawk": null,
        "type": "LJ60",
        "vert_rate": 2880
    }
]
```

These results confirm that authentication is now successful (an intentional failing scenario using an incorrect password is omitted due to space considerations) and valid users can once again access the exposed API.

Returning to examine the logged, and now encoded, passwords, I note values similar to the following in the IDE's output:

```
>>> Peter's password:
    {bcrypt}$2a$10$rLKBzRBvtTtNcV9o8JHzFeaIskJIPXnYgVtCPs5H0GINZtk1WzsBu
>>> Jodie's password: {
    bcrypt}$2a$10$VR33/dlbSsEPPq6nlpnE/.ZQt0M4.bjvO5UYmw0ZW1aptO4G8dEkW
```

The logged values confirm that both example passwords specified in the code have been encoded successfully by the delegated password encoder using *BCrypt*.

A NOTE ABOUT ENCODED PASSWORD FORMAT

Based on the format of the encoded password, the correct password encoder will be selected automatically to encode (for comparison) the password supplied by the user attempting to authenticate. Spring Security prepends the encoded value with a key indicating which algorithm was used for convenience, which sometimes gives developers pause. Is that potentially giving away vital information in the event the (encrypted) password is obtained by a hostile actor? Wouldn't that knowledge make it easier to decrypt?

The short answer is no. The strength lies in the encryption itself, not from any perceived obscurity.

How can I be certain of this?

Most methods of encryption already have "tells" that indicate what was used to encrypt a value. Note the two passwords listed previously. Both encoded values begin with the string of characters `$2a$10$`, and indeed, all *BCrypt*-encrypted values do. While it's possible to have an encryption algorithm that does not signal which mechanism was used in the resultant encoded value, this would be the exception rather than the rule.

Authorization

The `Aircraft Positions` application now successfully authenticates users and allows only said users to access its exposed API. There is a rather large issue with the current security configuration, though: access to any part of the API means access to all of it, regardless of roles/authori-

ty the user possesses—or more accurately, regardless of roles *not* possessed.

As a very simple example of this security flaw, I add another endpoint to `Aircraft Position`'s API by cloning, renaming, and remapping the existing `getCurrentAircraftPositions()` method in the `PositionController` class as a second endpoint. Once complete, `PositionController` appears as follows:

```
import lombok.AllArgsConstructor;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@AllArgsConstructor
@RestController
public class PositionController {
    private final PositionRetriever retriever;

    @GetMapping("/aircraft")
    public Iterable<Aircraft> getCurrentAircraftPositions() {
        return retriever.retrieveAircraftPositions();
    }

    @GetMapping("/aircraftadmin")
    public Iterable<Aircraft> getCurrentAircraftPositionsAdminPrivs() {
        return retriever.retrieveAircraftPositions();
    }
}
```

The goal is to allow only users having the "ADMIN" role access to the second method, `getCurrentAircraftPositionsAdminPrivs()`. While in this version of this example the values returned are identical to those returned by the `getCurrentAircraftPositions()`, this will likely not remain the case as the application expands, and the concept applies regardless.

Restarting the `Aircraft Positions` application and returning to the command line, I login first as user Jodie to verify access to the new endpoint, as expected (first endpoint access confirmed but omitted due to space; some headers and results also omitted for brevity):

```
mheckler-a01 :: ~ » http :8080/aircraftadmin --auth jodie:jpassword
HTTP/1.1 200
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Expires: 0
Pragma: no-cache
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

```
[
    {
        "altitude": 24250,
        "barometer": 0.0,
        "bds40_seen_time": null,
        "callsign": null,
        "category": "A2",
        "flightno": "",
        "heading": 118,
        "id": 1,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-12T16:13:26Z",
        "lat": 38.325119,
        "lon": -90.154159,
        "polar_bearing": 178.56009,
        "polar_distance": 37.661127,
        "pos_update_time": "2020-10-12T16:13:24Z",
        "reg": "N168ZZ",
        "route": "FMY-SUS",
        "selected_altitude": 0,
        "speed": 404,
        "squawk": null,
        "type": "LJ60",
        "vert_rate": 2880
    },
    {
        "altitude": 38000,
        "barometer": 1013.6,
        "bds40_seen_time": "2020-10-12T20:24:48Z",
        "callsign": "SWA1828",
        "category": "A3",
        "flightno": "WN1828",
        "heading": 274,
        "id": 2,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-12T20:24:48Z",
        "lat": 39.348862,
        "lon": -90.751668,
        "polar_bearing": 310.510201,
        "polar_distance": 35.870036,
        "pos_update_time": "2020-10-12T20:24:48Z",
        "reg": "N8567Z",
        "route": "TPA-BWI-OAK",
        "selected_altitude": 38016,
        "speed": 397,
        "squawk": "7050",
        "type": "B738",
        "vert_rate": -128
    }
]
```

Next, I log in as Peter. Peter should not have access to the `getCurrentAircraftPositionsAdminPrivs()` method, mapped to */aircraftadmin*. But that isn't the case; currently Peter—an authenticated user—can access everything:

```
mheckler-a01 :: ~ » http :8080/aircraftadmin --auth peter:ppassword
HTTP/1.1 200
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Expires: 0
Pragma: no-cache
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

[
    {
        "altitude": 24250,
        "barometer": 0.0,
        "bds40_seen_time": null,
        "callsign": null,
        "category": "A2",
        "flightno": "",
        "heading": 118,
        "id": 1,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-12T16:13:26Z",
        "lat": 38.325119,
        "lon": -90.154159,
        "polar_bearing": 178.56009,
        "polar_distance": 37.661127,
        "pos_update_time": "2020-10-12T16:13:24Z",
        "reg": "N168ZZ",
        "route": "FMY-SUS",
        "selected_altitude": 0,
        "speed": 404,
        "squawk": null,
        "type": "LJ60",
        "vert_rate": 2880
    },
    {
        "altitude": 38000,
        "barometer": 1013.6,
        "bds40_seen_time": "2020-10-12T20:24:48Z",
        "callsign": "SWA1828",
        "category": "A3",
        "flightno": "WN1828",
        "heading": 274,
        "id": 2,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-12T20:24:48Z",
```

```
        "lat": 39.348862,
        "lon": -90.751668,
        "polar_bearing": 310.510201,
        "polar_distance": 35.870036,
        "pos_update_time": "2020-10-12T20:24:48Z",
        "reg": "N8567Z",
        "route": "TPA-BWI-OAK",
        "selected_altitude": 38016,
        "speed": 397,
        "squawk": "7050",
        "type": "B738",
        "vert_rate": -128
    }
]
```

To enable the `Aircraft Positions` application to not simply authenticate users but also to check user authorization to access particular resources, I refactor `SecurityConfig` to perform that task.

The first step is to replace the class-level annotation `@Configuration` with `@EnableWebSecurity`. `@EnableWebSecurity` is a meta-annotation that includes the removed `@Configuration`, still allowing for bean creation methods within the annotated class; but it also includes the `@EnableGlobalAuthentication` annotation that enables a great deal more security autoconfiguration to be done by Spring Boot for the application. This positions `Aircraft Positions` well for the next step of defining the authorization mechanism itself.

I refactor the `SecurityConfig` class to extend `WebSecurityConfigurerAdapter`, an abstract class with numerous member variables and methods useful for extending the basic configuration of an application's web security. In particular, `WebSecurityConfigurerAdapter` has a `configure(HttpSecurity http)` method that provides a basic implementation for user authorization:

```
protected void configure(HttpSecurity http) throws Exception {
    // Logging statement omitted

    http
        .authorizeRequests()
            .anyRequest().authenticated()
            .and()
        .formLogin().and()
        .httpBasic();
}
```

In the preceding implementation, the following directives are issued:

- Authorize any request from an authenticated user.
- Simple login and logout forms (overridable ones created by the developer) will be provided.
- HTTP Basic Authentication is enabled for nonbrowser user agents (command line tools, for example).

This provides a reasonable security posture if no authorization specifics are supplied by the developer. The next step is to provide more specifics and thus override this behavior.

I use IntelliJ for Mac's `CTRL+O` keyboard shortcut or click the right mouse button and then Generate to open the Generate menu, then select the Override methods... option to display overridable/implementable methods. Selecting the method with signature `configure(http:HttpSecurity):void` produces the following method:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    super.configure(http);
}
```

I then replace the call to the superclass's method with the following code:

```
// User authorization
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .mvcMatchers("/aircraftadmin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .and()
            .httpBasic();
}
```

This implementation of the `configure(HttpSecurity http)` method performs the following actions:

- Using a `String` pattern matcher, the request path is compared for a match with */aircraftadmin* and all paths below.
- If the match is successful, the user is authorized to make the request if the user has the "ADMIN" role/authority.
- Any other request is fulfilled for any authenticated user
- Simple login and logout forms (overridable ones created by the developer) will be provided

- HTTP Basic Authentication is enabled for nonbrowser user agents (command line tools, etc.).

This minimal authorization mechanism places two filters in the security filter chain: one to check for a path match and admin privileges and one for all other paths and an authenticated user. A tiered approach allows for complex scenarios to be captured in fairly simple, easy-to-reason-about logic.

The final version (for forms-based security) of the `SecurityConfig` class looks like this:

```
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.builders.HttpSecuri
import org.springframework.security.config.annotation.web.configuration
    .EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration
    .WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    private final PasswordEncoder pwEncoder =
            PasswordEncoderFactories.createDelegatingPasswordEncoder();

    @Bean
    UserDetailsService authentication() {
        UserDetails peter = User.builder()
                .username("peter")
                .password(pwEncoder.encode("ppassword"))
                .roles("USER")
                .build();

        UserDetails jodie = User.builder()
                .username("jodie")
                .password(pwEncoder.encode("jpassword"))
                .roles("USER", "ADMIN")
                .build();

        System.out.println("   >>> Peter's password: " + peter.getPassword())
        System.out.println("   >>> Jodie's password: " + jodie.getPassword())

        return new InMemoryUserDetailsManager(peter, jodie);
    }

    @Override
```

```
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .mvcMatchers("/aircraftadmin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
                .and()
                .formLogin()
                .and()
                .httpBasic();
    }
}
```

ADDITIONAL NOTES ON SECURITY

It's absolutely critical that *more-specific criteria are placed ahead of less-specific criteria.*

Each request passes through the security filter chain until it matches a filter, at which point the request is processed in accordance with the specified conditions. If, for example, the `.anyRequest().authenticated()` condition were placed before `.mvcMatchers("/aircraftadmin/**").hasRole("ADMIN")`, any request would match, again providing access to all exposed resources—including all resources under */aircraftadmin*—to all authenticated users.

Placing `.mvcMatchers("/aircraftadmin/**").hasRole("ADMIN")` (and any other more specific criteria) before the criteria for `anyRequest()` returns `anyRequest()` to a catch-all state, a very useful position when done intentionally to allow access to areas of the application that all authenticated users should have, such as a common landing page, menus, etc.

Note also that there are some small differences in annotations, class/bean names, return types, and approaches to authentication and authorization between Spring MVC-based (nonreactive) and Spring WebFlux-based (reactive) Spring Boot apps, but there is a very high degree of overlap. These will be addressed to some extent in the following section.

Now to confirm that all works as intended. I restart the `Aircraft Positions` application and access the */aircraftadmin* endpoint as Jodie from the command line (first endpoint access confirmed but omitted due to space; some headers and results also omitted for brevity):

```
mheckler-a01 :: ~ » http :8080/aircraftadmin --auth jodie:jpassword
HTTP/1.1 200
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Expires: 0
Pragma: no-cache
```

```
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

[
    {
        "altitude": 36000,
        "barometer": 1012.8,
        "bds40_seen_time": "2020-10-13T19:16:10Z",
        "callsign": "UPS2806",
        "category": "A5",
        "flightno": "5X2806",
        "heading": 289,
        "id": 1,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-13T19:16:14Z",
        "lat": 38.791122,
        "lon": -90.21286,
        "polar_bearing": 189.515723,
        "polar_distance": 9.855602,
        "pos_update_time": "2020-10-13T19:16:12Z",
        "reg": "N331UP",
        "route": "SDF-DEN",
        "selected_altitude": 36000,
        "speed": 374,
        "squawk": "6652",
        "type": "B763",
        "vert_rate": 0
    },
    {
        "altitude": 25100,
        "barometer": 1012.8,
        "bds40_seen_time": "2020-10-13T19:16:13Z",
        "callsign": "ASH5937",
        "category": "A3",
        "flightno": "AA5937",
        "heading": 44,
        "id": 2,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-10-13T19:16:13Z",
        "lat": 39.564148,
        "lon": -90.102459,
        "polar_bearing": 5.201331,
        "polar_distance": 36.841422,
        "pos_update_time": "2020-10-13T19:16:13Z",
        "reg": "N905J",
        "route": "DFW-BMI-DFW",
        "selected_altitude": 11008,
        "speed": 476,
        "squawk": "6270",
        "type": "CRJ9",
```

```
        "vert_rate": -2624
    }
]
```

Jodie is able to access the */aircraftadmin* endpoint as expected owing to having the "ADMIN" role. Next, I try using Peter's login. Note that the first endpoint access was confirmed but omitted due to space; some headers were also omitted for brevity:

```
mheckler-a01 :: ~ » http :8080/aircraftadmin --auth peter:ppassword
HTTP/1.1 403
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Expires: 0
Pragma: no-cache
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

{
    "error": "Forbidden",
    "message": "",
    "path": "/aircraftadmin",
    "status": 403,
    "timestamp": "2020-10-13T19:18:10.961+00:00"
}
```

This is exactly what should have occurred, since Peter only has the "USER" role and not "ADMIN." The system works.

CODE CHECKOUT CHECKUP

Please check out branch *chapter10forms* from the code repository for a complete forms-based example.

Implementing OpenID Connect and OAuth2 for Authentication and Authorization

While forms-based authentication and internal authorization is useful for a large number of applications, numerous use cases exist in which "something you know" methods of authentication are less than ideal or even insufficient for the desired or required level of security. Some examples include but are not limited to the following:

- Free services that require authentication but that don't need to know anything about the user (or don't want to know, for legal or other reasons)
- Situations in which single-factor authentication isn't considered secure enough, desiring and/or requiring Multi-Factor Authentication (MFA) support
- Concerns about creating and maintaining secure software infrastructure for managing passwords, roles/authorities, and other necessary mechanisms
- Concerns over liability in the event of compromise

There is no simple answer to any of these concerns or goals, but several companies have built and maintain robust and secure infrastructure assets for authentication and authorization and offer it for general use at low or no cost. Companies like Okta, a leading security vendor, and others whose businesses require proven user validation and permissions verification: Facebook, GitHub, and Google, to name a few. Spring Security supports all of these options and more via OpenID Connect and OAuth2.

OAuth2 was created to provide a means for third-party authorization of users for specified resources, such as cloud-based services, shared storage, and applications. OpenID Connect builds on OAuth2 to add consistent, standardized authentication using one or more factors from the following:

- Something you know, for example, a password
- Something you have, like a hardware key
- Something you are, such as a biometric identifier

Spring Boot and Spring Security support autoconfiguration out of the box for OpenID Connect and OAuth2 implementations offered by Facebook, GitHub, Google, and Okta, with additional providers easily configurable due to the published standards for OpenID Connect and OAuth2 and Spring Security's extensible architecture. I use Okta's libraries and authentication+authorization mechanisms for the examples that follow, but differences between providers are mostly variations on a theme. Feel free to use the security provider that best suits your needs.

VARIOUS APPLICATION/SERVICE ROLES FOR OPENID CONNECT AND OAUTH2

While this section speaks directly to roles fulfilled by various services using OpenID Connect and OAuth2 for authentication and authorization, respectively, it can really apply in whole or part to any type of third-party authentication and authorization mechanism(s).

Applications/services fulfill three primary roles:

- Client
- Authorization server
- Resource server

Typically one or more services are considered to be clients, applications/services with which an end user interacts and that works with one or more security providers to authenticate and obtain authorization (roles/authorities) granted to the user for various resources.

There are one or more authorization servers that handle user authentication and return to the client(s) the authorities a user possesses. Authorization servers handle the issuance of timed authorizations and, optionally, renewals.

Resource servers provide access to protected resources based on authorities presented by clients.

Spring Security enables developers to create all three types of applications/services, but for the purposes of this book, I focus on creating a client and a resource server. The Spring Authorization Server is currently considered an experimental project but is maturing quickly, and it will be extremely useful for a number of use cases; however, for many organizations and for many of the goals listed previously, authorization services provided by a third party continue to make the most sense. As with all decisions, your requirements should determine your path.

In this example, I refactor `Aircraft Positions` to serve as an OpenID Connect and OAuth2 client application, working with Okta's capabilities to validate the user and obtain the user's authorities to access resources exposed by a resource server. I then refactor PlaneFinder to provide its resources—as an OAuth2 resource server—based on credentials supplied with requests from the `Aircraft Positions` (client) application.

Aircraft Positions Client Application

I typically begin with the application farthest back in the stack, but in this case, I believe the opposite approach has more merit due to the flows as-

sociated with a user gaining (or being denied) access to a resource.

A user accesses a client application that uses some mechanism to authenticate them. Once authenticated, user requests for resources are relayed to so-called resource servers that hold and manage said resources. This is a logical flow that most of us follow repeatedly and find very familiar. By enabling security in the same order—client, then resource server—it neatly aligns with our own, expected flow.

Adding OpenID Connect and OAuth2 dependencies to Aircraft Positions

As with forms-based security, it's simple to add additional dependencies via the Spring Initializr when creating a new Spring Boot client project to get started with OpenID Connect and OAuth2 in a greenfield client application, as shown in Figure 10-2.

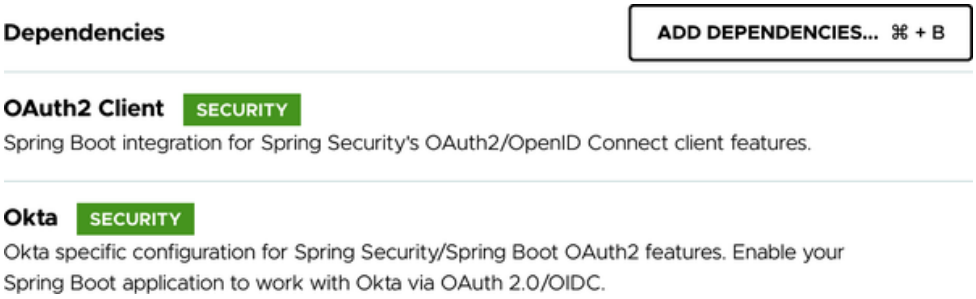

Figure 10-2. Dependencies for OpenID Connect and OAuth2 Client application using Okta within Spring Initializr

Updating an existing application requires only a bit more effort. Since I'm replacing the current forms-based security, I first remove the existing dependency for Spring Security that I added in the previous section. Then I add the same two dependencies that Initializr adds for OpenID Connect and OAuth2, one for the OAuth2 Client (which includes the OpenID Connect authentication piece and other necessary components) and one for Okta, since we'll be using their infrastructure to authenticate and manage authorities, to `Aircraft Positions` 's *pom.xml* Maven build file:

```
<!--    Comment out or remove this      -->
<!--<dependency>-->
<!--    <groupId>org.springframework.boot</groupId>-->
<!--    <artifactId>spring-boot-starter-security</artifactId>-->
<!--</dependency>-->

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

```
        <groupId>com.okta.spring</groupId>
        <artifactId>okta-spring-boot-starter</artifactId>
        <version>1.4.0</version>
    </dependency>
```

NOTE

The current included version of Okta's Spring Boot Starter library is 1.4.0. This is the version that has been tested and verified to work well with the current version of Spring Boot. When adding dependencies to a build file manually, a good practice for developers to make habit is to visit the Spring Initializr, select the current (at that time) version of Boot, add the Okta (or other specifically versioned) dependency, and *Explore* the project to confirm the current recommended version number.

Once I refresh the build, it's time to refactor the code to enable `Aircraft Positions` to authenticate with Okta and obtain user authorities.

Refactoring Aircraft Positions for authentication and authorization

There are really three things required to configure the current `Aircraft Positions` as an OAuth2 client app:

- Remove the forms-based security configuration.
- Add OAuth2 configuration to the created `WebClient` used to access PlaneFinder endpoints.
- Specify OpenID Connect+OAuth2 registered client credentials and a URI for the security provider (in this case, Okta).

I tackle the first two together, beginning by removing the body of the `SecurityConfig` class in its entirety. If access control to resources provided locally by `Aircraft Positions` is still desired or required, `SecurityConfig` can of course remain as it is or with some slight modification; however, for this example, PlaneFinder fulfills the role of resource server and as such should control or deny access to requested resources of value. `Aircraft Positions` acts simply as a user client that works with security infrastructure to enable a user to authenticate, then passes requests for resources to resource server(s).

I replace the `@EnableWebSecurity` annotation with `@Configuration`, as the autoconfiguration for local authentication is no longer needed. Also gone is `extends WebSecurityConfigurerAdapter` from the class header, since this particular iteration of the `Aircraft Positions` application doesn't restrict requests to its endpoints, instead passing the

user's authorities with requests to PlaneFinder so it can compare those authorities against those allowed for each resource and act accordingly.

Next, I create a `WebClient` bean within the `SecurityConfig` class for use throughout the `Aircraft Positions` application. This is not a hard requirement at this point, as I could simply incorporate the OAuth2 configuration into the creation of the `WebClient` assigned to the member variable within `PositionRetriever`, and there are valid arguments for doing so. That said, `PositionRetriever` needs access to a `WebClient`, but configuring the `WebClient` to handle OpenID Connect and OAuth2 configuration runs pretty far afield of the core mission of `PositionRetriever`: to retrieve aircraft positions.

Creating and configuring a `WebClient` for authentication and authorization fits very well within the scope of a class named `SecurityConfig`:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.oauth2.client.registration
    .ClientRegistrationRepository;
import org.springframework.security.oauth2.client.web
    .OAuth2AuthorizedClientRepository;
import org.springframework.security.oauth2.client.web.reactive.function.clien
    .ServletOAuth2AuthorizedClientExchangeFilterFunction;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class SecurityConfig {
    @Bean
    WebClient client(ClientRegistrationRepository regRepo,
                     OAuth2AuthorizedClientRepository cliRepo) {
        ServletOAuth2AuthorizedClientExchangeFilterFunction filter =
                new ServletOAuth2AuthorizedClientExchangeFilterFunction
                    (regRepo, cliRepo);

        filter.setDefaultOAuth2AuthorizedClient(true);

        return WebClient.builder()
                .baseUrl("http://localhost:7634/")
                .apply(filter.oauth2Configuration())
                .build();
    }
}
```

Two beans are autowired into the `client()` bean creation method:

- The `ClientRegistrationRepository`, a list of OAuth2 clients specified for use by the application, usually in a properties file like *application.yml*

- `OAuth2AuthorizedClientRepository`, a list of OAuth2 clients that represent an authenticated user and manage that user's `OAuth2AccessToken`

Within the method to create and configure the `WebClient` bean, I perform the following actions:

1. I initialize a filter function with the two injected repositories.
2. I confirm that the default authorized client should be used. This is typically the case—after all, the authenticated user is typically the resource owner who wishes to gain access to the resource—but optionally, a different authorized client could be desired for use cases involving delegated access. . I specify the URL and apply the filter configured for OAuth2 to the `WebClient` builder and build the `WebClient`, returning it as a Spring bean and adding it to the `ApplicationContext`. The OAuth2-enabled `WebClient` is now available for use throughout the `Aircraft Positions` application.

Since the `WebClient` bean is now created by the application via a bean creation method, I now remove the statement creating and directly assigning a `WebClient` object to a member variable within the `PositionRetriever` class and replace it with a simple member variable declaration. With the Lombok `@AllArgsConstructor` annotation on the class, Lombok automatically adds a `WebClient` parameter to the "all arguments constructor" it generates for the class. Since a `WebClient` bean is available in the `ApplicationContext`, Spring Boot autowires it into `PositionRetriever` where it is assigned to the `WebClient` member variable automatically. The newly refactored `PositionRetriever` class now looks like this:

```
import lombok.AllArgsConstructor;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@AllArgsConstructor
@Component
public class PositionRetriever {
    private final AircraftRepository repository;
    private final WebClient client;

    Iterable<Aircraft> retrieveAircraftPositions() {
        repository.deleteAll();

        client.get()
                .uri("/aircraft")
                .retrieve()
                .bodyToFlux(Aircraft.class)
                .filter(ac -> !ac.getReg().isEmpty())
                .toStream()
```

```
                .forEach(repository::save);

        return repository.findAll();
    }
}
```

Earlier in this section I mentioned the use of a `ClientRegistrationRepository`, a list of OAuth2 clients specified for use by the application. There are many ways to populate this repository, but entries are usually specified as application properties. In this example, I add the following information to `Aircraft Position`'s *application.yml* file (dummy values shown here):

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: <your_assigned_client_id_here>
            client-secret: <your_assigned_client_secret_here>
        provider:
          okta:
            issuer-uri: https://<your_assigned_subdomain_here>
                                .oktapreview.com/oauth2/default
```

OBTAINING CLIENT AND ISSUER DETAILS FROM AN OPENID CONNECT + OAUTH2 PROVIDER

Since this section is focused on how to properly develop Spring Boot applications securely by interacting with security infrastructure provided by trusted third parties, providing detailed steps for creating accounts with those numerous security providers, registering applications, and defining user authorities for various resources falls somewhat outside of the defined scope of this chapter. Fortunately, the procedures required to perform those external actions are covered within the Spring Security OAuth2 sample repository. Follow this link to Configure Okta as an Authentication Provider; similar steps for other supported providers are included in the same document.

With that information in place, the `Aircraft Positions` application's `ClientRegistrationRepository` will have a single entry for Okta that it will use automatically when a user attempts to access the application.

TIP

If multiple entries are defined, a web page will be presented upon first request, prompting the user to choose a provider.

I make one other small change to `Aircraft Positions` (and a small downstream change to `PositionRetriever`), only to better demonstrate successful and unsuccessful user authorization. I replicate the sole endpoint currently defined in the `PositionController` class, rename it, and assign it a mapping implying "admin only" access:

```
import lombok.AllArgsConstructor;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@AllArgsConstructor
@RestController
public class PositionController {
    private final PositionRetriever retriever;

    @GetMapping("/aircraft")
    public Iterable<Aircraft> getCurrentAircraftPositions() {
        return retriever.retrieveAircraftPositions("aircraft");
    }

    @GetMapping("/aircraftadmin")
    public Iterable<Aircraft> getCurrentAircraftPositionsAdminPrivs() {
        return retriever.retrieveAircraftPositions("aircraftadmin");
    }
}
```

To accommodate access to both PlaneFinder endpoints using a single method in `PositionRetriever`, I change its `retrieveAircraftPositions()` method to accept a dynamic path parameter `String endpoint` and use it when building the client request. The updated `PositionRetriever` class looks like this:

```
import lombok.AllArgsConstructor;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@AllArgsConstructor
@Component
public class PositionRetriever {
    private final AircraftRepository repository;
    private final WebClient client;

    Iterable<Aircraft> retrieveAircraftPositions(String endpoint) {
        repository.deleteAll();
```

```
            client.get()
                    .uri((null != endpoint) ? endpoint : "")
                    .retrieve()
                    .bodyToFlux(Aircraft.class)
                    .filter(ac -> !ac.getReg().isEmpty())
                    .toStream()
                    .forEach(repository::save);

            return repository.findAll();
        }
    }
```

`Aircraft Positions` is now a fully configured OpenID Connect and OAuth2 client application. Next, I refactor PlaneFinder to serve as an OAuth2 resource server, providing resources upon request to authorized users.

PlaneFinder Resource Server

With any refactoring involving a change of dependencies, the place to begin is with the build file.

Adding OpenID Connect and OAuth2 Dependencies to Aircraft Positions

As mentioned before, it's easy to simply add another dependency or two via the Spring Initializr when creating a new Spring Boot OAuth2 resource server in a greenfield client application, as shown in Figure 10-3.

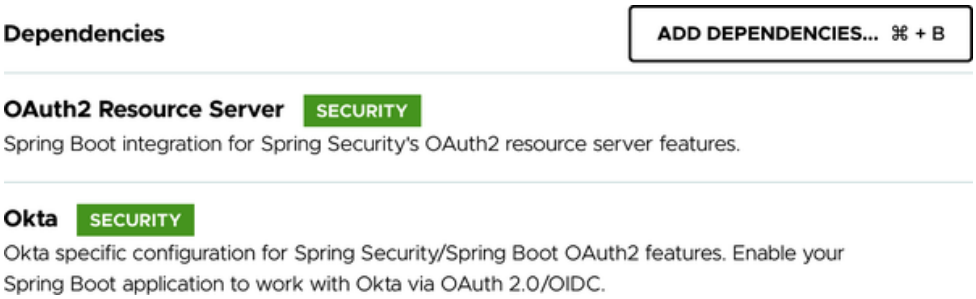

Figure 10-3. Dependencies for OAuth2 Resource Server using Okta within Spring Initializr

INVESTIGATING THOSE DEPENDENCIES

Since I've chosen to use Okta for OpenID Connect and OAuth2 mechanisms (security infrastructure and libraries) in these examples, I could accomplish the same results by adding *only* the Okta Spring Boot starter dependency to the project. The Okta dependency brings along with it all other necessary libraries for both OAuth2 client apps and resource servers:

- Spring Security Config
- Spring Security Core
- Spring Security OAuth2 Client
- Spring Security Core
- Spring Security JOSE
- Spring Security Resource Server
- Spring Security Web

Adding the dependency from the Initializr for the OAuth2 Resource Server doesn't add any extra baggage to a new application because it includes the same dependencies minus the one for OAuth2 Client; the Okta dependencies are a superset. I do it primarily as a visual cue and a good practice to avoid chasing dependencies in the event I later decide to change authentication and authorization providers. That said, I recommend you always inspect your application's dependency tree and remove unnecessary ones.

Updating the existing PlaneFinder application is straightforward enough. I add the same two dependencies that Initializr adds for the OAuth2 Resource Server and for Okta, since we'll be using their infrastructure to verify authorities, to PlaneFinder's *pom.xml* Maven build file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>com.okta.spring</groupId>
    <artifactId>okta-spring-boot-starter</artifactId>
    <version>1.4.0</version>
</dependency>
```

Once I refresh the build, it's time to refactor the code to enable PlaneFinder to verify user authorities provided with inbound requests to verify user permissions and grant (or deny) access to PlaneFinder resources.

Refactoring PlaneFinder for resource authorization

Much of the work to enable OpenID Connect and OAuth2 authentication and authorization using Okta for our distributed system has already been accomplished by this point. Refactoring PlaneFinder to correctly perform the duties of an OAuth2 resource server requires minimal effort:

- Incorporating JWT (JSON Web Token) support
- Comparing the authorities delivered within the JWTs (pronounced "jots") to those required for access to designated resources

Both of these tasks can be accomplished by creating a single `SecurityWebFilterChain` bean that Spring Security will use for retrieving, verifying, and comparing the contents of the inbound request's JWT with required authorities.

Once again I create a `SecurityConfig` class and annotate it with `@Configuration` to provide a distinct place for bean creation methods. Next, I create a `securityWebFilterChain()` method as follows:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

@Configuration
public class SecurityConfig {
    @Bean
    public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity h
        http
                .authorizeExchange()
                .pathMatchers("/aircraft/**").hasAuthority("SCOPE_closedid")
                .pathMatchers("/aircraftadmin/**").hasAuthority("SCOPE_openid
                .and().oauth2ResourceServer().jwt();

        return http.build();
    }
}
```

To create the filter chain, I autowire the existing `ServerHttpSecurity` bean provided by Spring Boot's security autoconfiguration. This bean is used with WebFlux-enabled applications, i.e., when `spring-boot-starter-webflux` is on the classpath.

NOTE

Applications without WebFlux on the classpath would use the `HttpSecurity` bean and its corresponding methods instead, as was done in the forms-based authentication example earlier in this chapter.

Next, I configure the `ServerHttpSecurity` bean's security criteria, specifying how requests should be handled. To do so, I provide two resource paths to match against requests and their required user authorities; I also enable OAuth2 resource server support using JWTs to bear the user information.

NOTE

JWTs are sometimes referred to as *bearer tokens* because they bear the user's authorization for access to resources.

Finally, I build the `SecurityWebFilterChain` from the `ServerHttpSecurity` bean and return it, making it available as a bean throughout the PlaneFinder application.

When a request arrives, the filter chain compares the requested resource's path to paths specified in the chain until a match is found. Once a match is made, the application verifies the token validity with the OAuth2 provider—Okta, in this case—and then compares the contained authorities with those required for access to the mapped resources. If there is a valid match, access is granted; if not, the application returns a *403 Forbidden* status code.

You might have noticed that the second `pathMatcher` specifies a resource path that doesn't (yet) exist in PlaneFinder. I add this path to the `PlaneController` class solely to be able to provide examples of both successful and failed authority checks.

OAuth2 providers may include several default authorities, including *openid*, *email*, *profile*, and more. In the example filter chain, I check a nonexistent (for my provider and OAuth2 authority configuration) authority of *closedid*; consequently, any request for a resource with a path beginning with */aircraft* will fail. As currently written, any inbound request for resources beginning with a path of */aircraftadmin* and bearing a valid token will succeed.

NOTE

Spring Security prepends “SCOPE_” to OAuth2 provider-supplied authorities, mapping Spring Security’s internal concept of scopes 1:1 with OAuth2 authorities. For developers using Spring Security with OAuth2, this is important to be aware of but is a distinction without a practical difference.

To complete the code refactoring, I now add the */aircraftadmin* endpoint mapping referenced in the previous path matcher to PlaneFinder’s `PlaneController` class, simply copying the functionality of the existing */aircraft* endpoint in order to demonstrate two endpoints with different access criteria:

```
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import reactor.core.publisher.Flux;

import java.io.IOException;
import java.time.Duration;

@Controller
public class PlaneController {
    private final PlaneFinderService pfService;

    public PlaneController(PlaneFinderService pfService) {
        this.pfService = pfService;
    }

    @ResponseBody
    @GetMapping("/aircraft")
    public Flux<Aircraft> getCurrentAircraft() throws IOException {
        return pfService.getAircraft();
    }

    @ResponseBody
    @GetMapping("/aircraftadmin")
    public Flux<Aircraft> getCurrentAircraftByAdmin() throws IOException {
        return pfService.getAircraft();
    }

    @MessageMapping("acstream")
    public Flux<Aircraft> getCurrentACStream() throws IOException {
        return pfService.getAircraft().concatWith(
                Flux.interval(Duration.ofSeconds(1))
                        .flatMap(l -> pfService.getAircraft()));
    }
}
```

Finally, I must indicate to the application where to go to access the OAuth2 provider in order to validate the incoming JWTs. There may be variations in how this is done, as the specification for OAuth2 provider endpoints has some latitude, but Okta helpfully implements an issuer URI to act as a central URI for configuration from which other necessary URIs can be obtained. This reduces the burden on application developers to adding a single property.

I've converted the *application.properties* file from a key-value pairs format to *application.yml*, allowing for a structured tree of properties, reducing repetition a bit. Note that this is optional but useful when duplication in property keys begins to manifest:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://<your_assigned_subdomain_here>.oktapreview.com/
              oauth2/default
  rsocket:
    server:
      port: 7635

server:
  port: 7634
```

With all elements now in place, I restart both the PlaneFinder OAuth2 resource server and the `Aircraft Positions` OpenID Connect + OAuth2 client application to verify the results. Loading the address for `Aircraft Positions`'s */aircraftadmin* API endpoint (*http://localhost:8080/aircraftadmin*) in a browser, I'm redirected to Okta for authentication, as shown in Figure 10-4.

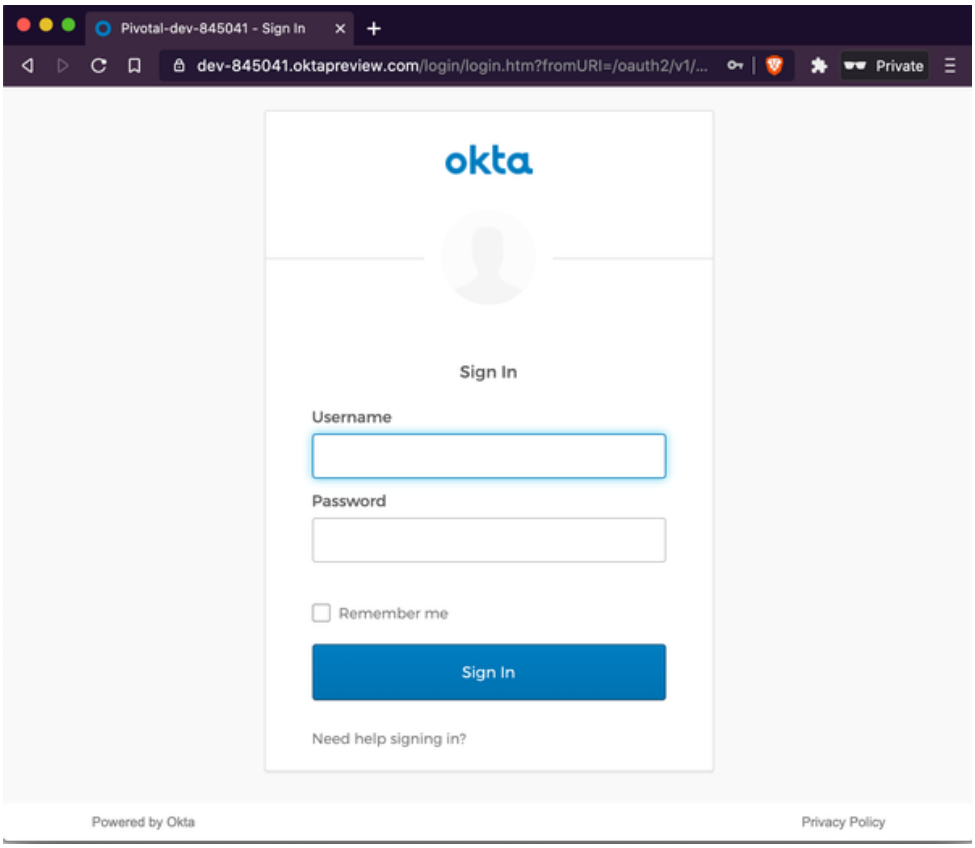

Figure 10-4. Login prompt provided by OpenID Connect provider (Okta)

Once I provide my valid user credentials, Okta redirects the authenticated user (me) to the client application, `Aircraft Positions`. The endpoint I requested in turn requests aircraft positions from PlaneFinder, passing along the JWT supplied to it by Okta. Once PlaneFinder matches the path requested to a resource path and verifies the JWT and its contained authorities, it responds with current aircraft positions to the `Aircraft Positions` client app, which in turn provides them to me, as shown in Figure 10-5.

Figure 10-5. Successful return current aircraft positions

What happens if I request a resource for which I have no authorization? To see an example of a failed authorization, I attempt to access `AircraftPosition` 's */aircraft* endpoint at *http://localhost:8080/aircraft*, with the results shown in Figure 10-6. Note that since I've already authenticated, I needn't reauthenticate to continue accessing the `AircraftPositions` application.

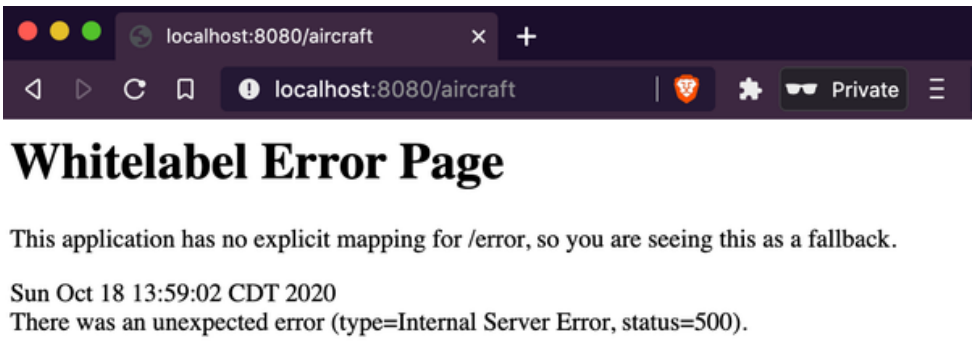

Figure 10-6. Results of failed authorization

Note that the response doesn't provide much information regarding the failure to retrieve results. It is generally considered a good security practice to avoid leaking details that could provide potential hostile actors with information that is helpful toward an eventual compromise. Visiting the logs for `Aircraft Positions`, however, I see the following additional information:

```
Forbidden: 403 Forbidden from GET http://localhost:7634/aircraft with root ca
```

This is exactly the response expected, since PlaneFinder's filter that matched requests for resources at or under */aircraft* expected the undefined authority *closedid*, which of course wasn't supplied.

These examples were distilled to the maximum extent possible, but they represent the key aspects of OpenID Connect authentication and OAuth2 authorization using a respected third-party security provider. Everything else that can be done to customize and extend this type of authentication and authorization for Spring Boot applications builds upon these fundamental principles and steps.

GOING WITH THE FLOW

The examples in this section utilize the Authorization Code Flow resulting in an Authorization Code Grant. The Authorization Code Flow is the process around which secure web applications are typically constructed and also serves as the centerpiece of the Authorization Code Flow with PKCE (Proof Key for Code Exchange) recommended for native applications.

There are other flows, notably the Resource Owner Password Flow, the Implicit Flow, and the Client Credentials Flow, but these other flows and their limitations and particular use cases are beyond the scope of this chapter.

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter10end* from the code repository.

Summary

Understanding the concepts of authentication and authorization are critical to building secure applications, providing the foundations for user verification and access control. Spring Security combines options for au-

thentication and authorization with other mechanisms like the HTTP Firewall, filter chains, extensive use of IETF and W3C standards and options for exchanges, and more to help lock down applications. Adopting a secure out-of-the-box mindset, Spring Security leverages Boot's powerful autoconfiguration to evaluate developer inputs and available dependencies to deliver maximal security for Spring Boot applications with minimal effort.

This chapter discussed several core aspects of security and how they apply to applications. I demonstrated multiple ways to incorporate Spring Security into Spring Boot apps to strengthen an application's security posture, closing dangerous gaps in coverage and reducing attack surface area.

The next chapter examines ways to deploy your Spring Boot application to various target destinations and discusses their relative merits. I also demonstrate how to create these deployment artifacts, provide options for their optimal execution, and show how to verify their components and provenance.