M. Nardone, C. Scarioni, *Pro Spring Security*
https://doi-org.ezproxy.sfpl.org/10.1007/979-8-8688-0035-1_2

2. Introducing Spring Security

Massimo Nardone[1] and Carlo Scarioni[2]
(1) HELSINKI, Finland
(2) Surbiton, UK

In this chapter, you learn what Spring Security is and how to use it to address security concerns about your application.

We describe what's new in Spring Framework and Spring Security version 6. Using Spring Security 6 with authentication and authorization is discussed in detail.

Finally, you look at the framework's source code, how to build it, and the different modules forming the powerful Spring Security project.

What Is Spring Security?

Spring Security is a framework dedicated to providing a full array of security services to Java applications in a developer-friendly and flexible way. It adheres to the well-established practices introduced by the Spring Framework. Spring Security tries to address all the layers of security inside your application. In addition, it comes packed with an extensive array of configuration options that make it very flexible and powerful.

Recall from Chapter 1 that it can be said that Spring Security is simply a comprehensive authentication/authorization framework built on top of the Spring Framework. Although most applications that use the framework are web-based, Spring Security's core can also be used in standalone applications.

Many things make Spring Security immediately attractive to Java developers. To name just a few, consider the following list.

- **It's built on top of the successful Spring Framework.** This is an important strength of Spring Security. The Spring Framework has become "the way" to build enterprise Java applications, and with good reason. It is built around good practices and two simple yet powerful concepts: dependency injection (DI) and aspect-oriented programming (AOP). Also important is that many developers have experience with Spring, so they can leverage that experience when introducing Spring Security in their projects.
- **It provides out-of-the-box support for many authentication models.** Even more important than the previous point, Spring Security supports out-of-the-box integration with Lightweight Directory Access Protocol (LDAP), OpenID, SAML 2.0, form authentication, OAuth 2.0, Certificate X.509 authentication, database authentication, Jasypt cryptography, and lots more. All this support means that Spring Security adapts to your security needs—and not only that, it can change if your needs change, without much effort involved for the developer. More information on Jasypt cryptography is at `www.jasypt.org/`.

  This is also important from a business point of view because the application can either adapt to the corporate authentication services or implement its own, thus requiring only straightforward configuration changes.

  This also means that there is a lot less software for you to write, because you are using a great amount of ready-to-use code that has been written and tested by a large and active user community. To a certain point, you can trust that this code works and use it with confidence. And if it does not work, you can always fix it and send a patch to those in charge of maintaining the project.
- **It offers layered security services.** Spring Security allows you to secure your application at different levels, and to secure your web URLs, views, service methods, and domain model. You can pick and combine these features to achieve your security goals.

  It is very flexible in practice. Imagine, for instance, that you offer services exposed through RMI, the Web, JMS, and others. You could secure all these interfaces, but maybe it's better to secure just the busi-

ness layer so that all requests are secured when they reach this layer. Also, maybe you don't care about securing individual business objects, so you can omit that module and use the functionality you need.
- **It is open source software.** As part of the Pivotal portfolio, Spring Security is an open source software tool. It also has a large community and user base dedicated to testing and improving the framework. Having the opportunity to work with open source software is an attractive feature for most developers. The ability to look into the source code of the tools you like and work with is an exciting prospect. Whether our goal is to improve the tools or simply to understand how they work internally, we developers love to read code and learn from it.

Where Does Spring Security Fit In?

Spring Security is without question a powerful and versatile tool. But like anything else, it is not a tool that adapts to everything you want to do. Its offerings have a defined scope.

Where and why would you use Spring Security? The following lists reasons and scenarios.

- **You need to develop web security.** Spring Security provides robust security features for web applications, including protection against common web vulnerabilities, such as cross-site scripting (XSS), cross-site request forgery (CSRF), and clickjacking.
- **You need strong mechanisms for securing URLs.** You want to restrict access to specific resources and enforce secure communication over HTTPS.
- **Your application is in Java, Groovy, or Kotlin.** The first thing to take into account is that Spring Security can be written in languages like Java, Groovy, or Kotlin and generally in any language supported by the JVM. So if you plan to work in a non-JVM language, Spring Security won't be useful.
- **You need role-based authentication/authorization.** This is the main use case of Spring Security. You have a list of users and resources and operations on those resources. You group the users into roles and

allow certain roles to access certain operations on certain resources. That's the core functionality.

- **You want to secure a web application from malicious users.** Spring Security is mostly used in web application environments. When this is the case, the first thing to do is allow only the users you want to access your application, while forbidding all others from even reaching it.
- **You need to integrate with OpenID, LDAP, Active Directory, and databases as security providers.** If you need to integrate with a particular Users and Roles or Groups provider, you should look at the vast array of options Spring Security offers because integration might already be implemented, saving you from writing lots of unnecessary code. Sometimes you might not be exactly sure what provider your business requires to authenticate against. In this case, Spring Security makes your life easy by allowing you to switch between different providers painlessly.
- **You need to secure your domain model and allow only certain users to access certain objects in your application.** If you need fine-grained security (that is, you need to secure on a per object, per user basis), Spring Security offers the access control list (ACL) module, which help you to do just that in a straightforward way.
- **You want a nonintrusive, declarative way for adding security around your application.** Security is a cross-cutting concern, not a core business functionality of your application (unless you work in a security provider firm). As such, it is better to be treated as a separate and modular add-on that you can declare, configure, and manage independently of your main business concerns. Spring Security is built with this in mind. Using servlet filters, XML configuration, and AOP concepts, the framework tries not to pollute your application with security rules. Even when using annotations, they are still metadata on top of your code. They don't mess with your code logic. As a Java developer, you must try to isolate the Java configuration into a configuration library and decouple it from the rest of the application in a similar way you do with XML.
- **You want to secure your service layer the same way you secure your URLs, and you need to add rules at the method level for allowing or disallowing user access.** Spring Security allows you to use

a consistent security model throughout the layers of your application because it internally enforces this consistent model itself. You configure users, roles, and providers in just one place, and both the service and web layers use this centralized security configuration transparently.

- **You need your application to remember its users on their next visit and allow them access.** Sometimes you don't want or need the users of your application to log in every time they visit your site. Spring supports out-of-the-box, remember-me functionality so that a user can be automatically logged in on subsequent visits to your site, allowing them full or partial access to their profile's functionality.
- **You want to use public/private key certificates to authenticate against your application.** Spring Security allows you to use X.509 certificates to verify the server's identity. The server can also request a valid certificate from the client for establishing mutual authentication.
- **You need to hide elements in your web pages from certain users and show them to others.** View security is the first layer of security in a secured web application. It is normally not enough for guaranteeing security. But it is very important from a usability point of view because it allows the application to show or hide content depending on the user currently logged in to the system.
- **You need more flexibility than simple role-based authentication for your application.** For example, suppose that you want to allow access only to users over 18 years of age using simple script expressions. Spring Security 3.1 uses the Spring Expression Language (SpEL) to allow you to customize access rules for your application.
- **You want your application to automatically handle HTTP status codes related to authorization errors (401, 403, and others).** The built-in exception-handling mechanism of Spring Security for web applications automatically translates the more common exceptions to their corresponding HTTP status codes; for example, `AccessDeniedException` gets translated to the 403 status code.
- **You want to configure your application to be used from other applications (not browsers) and allow these other applications to authenticate themselves against yours.** Another application accessing your application should be forced to use authentication mechanisms to gain access. For example, you can expose your application

through REST endpoints that other applications can access with HTTP security.

- **You are running an application outside a Java EE Server.** If you run your application in a simple web container like Apache Tomcat, you probably don't have support for the full Java EE security stack. Spring Security can be easily leveraged in these environments.
- **You are running an application inside a Java EE Server.** Even if you are running a full Java EE container, Spring Security is arguably more complete, flexible, and easy to use than the Java EE counterpart.
- **You are already using Spring in your application and want to leverage your knowledge.** You already know some of the great advantages of Spring. If you are currently using Spring, you probably like it a lot. You will probably like Spring Security as well.

Spring Security Overview

Spring Security 6 includes the following projects.

- Spring Security
- Spring Boot 3.0
- Spring Framework
- Spring Cloud Data Flow
- Spring Cloud
- Spring Data
- Spring Integration
- Spring Authorization Server
- Spring for GraphQL
- Spring Batch
- Spring Hateoas
- Spring REST Docs
- Spring Amqp
- Spring Mobile
- Spring For Android
- Spring Web Flow
- Spring Web Services
- Spring LDAP
- Spring Session

- Spring Shell
- Spring Flo
- Spring Kafka
- Spring Statemachine
- Spring Io Platform
- Spring Roo
- Spring Scala
- Spring Blazeds Integration
- Spring Loaded
- Spring Xd
- Spring Social

For more information, please refer to the Spring project web page at `https://spring.io/projects`.

Each of these projects is built on top of the facilities provided by the Spring Framework itself, which is the original project that started it all. Think of Spring as the hub of all these satellite projects, providing them with a consistent programming model and a set of established practices. The main points you see throughout the different projects is the use of DI, XML namespace-based configuration, and AOP, which, as you see in the next section, are the pillars upon which Spring is built. In the later versions of Spring, annotations have become the most popular way to configure both DI and AOP concerns.

This book introduces Spring Boot, analyzes Spring Framework, and develops Spring Security version 6. Let's start with Spring Boot.

What Is Spring Boot?

Spring Boot is an open source Java-based framework generally used for developing microservice, enterprise-ready applications. Pivotal developed it to help developers create stand-alone and production-ready Spring applications.

Spring Boot is an easy starting point for building all Spring-based applications and running them as quickly as possible, with minimal upfront configuration of Spring.

When this book was written, Spring Boot 3.0 was the latest release (November 2022) using Java 17+ and Jakarta EE 9.

Note Remember that a Spring Security application can be developed with Maven or Gradle.

Spring Security is one of the Spring projects; it is dedicated exclusively to addressing security concerns in applications.

For more information, please refer to the documentation at **`https://spring.io/projects/spring-security`**.

Spring Security began as a non-Spring project. It was originally known as the “Acegi Security System for Spring” and was not the big and powerful framework it is today. Originally, it dealt only with authorization and leveraged container-provided authentication. Because of public demand, the project started gaining traction, as more people started using it and contributing to its continuously growing code base. This eventually led to it becoming a Spring Framework portfolio project, and then later it was rebranded as Spring Security.

The following lists Spring Security’s major releases dates.

- 2.0.0 (April 2008)
- 3.0.0 (December 2009)
- 4.0.0 (March 2015)
- 5.0.0 (November 2017)
- 5.1.4 (February 2019)
- 6.1.0 (May 2023)

Java configuration for Spring Security was added to the Spring Framework in Spring 3.1 and extended to Spring Security in Spring 3.2 and is defined in a class annotated @Configuration.

Spring Security 6 requires JDK 17 and uses the Jakarta namespace.

The project for many years now has been under the Pivotal umbrella of projects, powered by the Spring Framework itself. But what exactly is the Spring Framework?

Spring Framework 6: A Quick Overview

We have mentioned the Spring Framework project a lot. It makes sense to give an overview of it at this point, because many of the Spring Security characteristics we cover in the rest of the book rely on the building blocks of Spring.

We admit we're biased. We love Spring and have loved it for many years now. We think Spring has so many advantages and great features that we can't start a new Java project without using it. Additionally, we tend to carry its concepts around when working with other languages and look for a way to apply them because they now feel natural.

Spring Framework 5 was published in September 2017 and can be considered the first major Spring Framework release since version 4 was released in December 2013.

Spring Framework latest release when this manuscipt was written is version 6.0.9 (May 2023).

Next, let's briefly review the most important new features in Spring Framework 6.

JDK 17+ and Jakarta EE 9+ Baseline

- Entire framework based on Java 17 source code level
- Migration from javax to jakarta namespace for Jakarta Servlet, JPA, and so on
- Runtime compatibility with Jakarta EE 9 and Jakarta EE 10 APIs
- Compatible with latest web servers—Tomcat 101, Jetty 11, Undertow 23
- Early compatibility with virtual threads (in preview as of JDK 19)

General Core Revision

- Upgrade to ASM 94 and Kotlin 17
- Complete CGLIB fork with support for capturing CGLIB-generated classes

- Comprehensive foundation for ahead-of-time transformations
- First-class support for GraalVM native images

Core Container

- First-class configuration options for virtual threads on JDK 21
- Lifecycle integration with Project CRaC for JVM checkpoint restore
- Support for resolving SequencedCollection/Set/Map at injection points
- Support for registering a MethodHandle as a SpEL function
- Validator factory methods for programmatic validator implementations
- Basic bean property determination without javabeansIntrospector by default
- AOT processing support in GenericApplicationContext (refreshForAotProcessing)
- Bean definition transformation based on pre-resolved constructors and factory methods
- Support for early proxy class determination for AOP proxies and configuration classes
- PathMatchingResourcePatternResolver uses NIO and module path APIs for scanning, enabling support for classpath scanning within a GraalVM native image and the Java module path, respectively
- DefaultFormattingConversionService supports ISO-based default java-time type parsing

Data Access and Transactions

- Failed CompletableFuture triggers rollback for async transactional method
- Support for predetermining JPA managed types (for inclusion in AOT processing)
- JPA support for Hibernate ORM 61 (retaining compatibility with Hibernate ORM 56)
- Upgrade to R2DBC 10 (including R2DBC transaction definitions)
- Aligned data access exception translation between JDBC, R2DBC, JPA and Hibernate
- Removal of JCA CCI support

Spring Messaging

- RSocket interface client based on @RSocketExchange service interfaces
- Early support for Reactor Netty 2 based on Netty 5 alpha
- Support for Jakarta WebSocket 21 and its standard WebSocket protocol upgrade mechanism

General Web Revision

- HTTP interface client based on @HttpExchange service interfaces
- Support for RFC 7807 problem details
- Unified HTTP status code handling
- Support for Jackson 214
- Alignment with Servlet 60 (while retaining runtime compatibility with Servlet 50)

Spring MVC

- PathPatternParser used by default (with the ability to opt into PathMatcher)
- Removal of outdated Tiles and FreeMarker JSP support

Spring WebFlux

- New PartEvent API to stream multipart form uploads (both on client and server)
- New ResponseEntityExceptionHandler to customize WebFlux exceptions and render RFC 7807 error responses
- Flux return values for non-streaming media types
- Early support for Reactor Netty 2 based on Netty 5 alpha
- JDK HttpClient integrated with WebClient

Observability

Spring 6 introduces **Spring Observability** , a new initiative that builds on Micrometer and Micrometer Tracing (formerly Spring Cloud Sleuth). The goal is to efficiently record application metrics with Micrometer and implement tracing through providers, such as OpenZipkin or OpenTelemetry.

- Direct observability instrumentation with micrometer observation is in several parts of the Spring Framework. The spring-web module now requires io.micrometer:micrometer-observation:1.10+ as a compile dependency.
- RestTemplate and WebClient are instrumented to produce HTTP client request observations.
- Spring MVC can be instrumented for HTTP server observations using org.springframework.web.filter.ServerHttpObservationFilter.
- Spring WebFlux can be instrumented for HTTP server observations using org.springframework.web.filter.reactive.ServerHttpObservationFilter.
- Integration with micrometer context propagation for flux and mono return values from controller methods.

Pattern Matching

Pattern matching was elaborated in Project Amber.

Testing

- Support for testing AOT-processed application contexts on the JVM or within a GraalVM native image
- Integration with HtmlUnit 264+ request parameter handling
- Servlet mocks (MockHttpServletRequest, MockHttpSession) are based on Servlet API 60 now

Many things attract us to Spring, but the main ones are the two major building blocks of the framework: dependency injection and aspect-oriented programming.

Why are these two concepts so important? They are important because they allow you to develop loosely coupled, single-responsibility, DRY (Don’t Repeat Yourself) code practically by default. These two concepts, and Spring itself, are covered extensively in other books and online tutorials; however, we’ll give you a brief overview here.

Spring Framework 6 and Spring Boot 3 need the following minimum versions.

- Kotlin 1.7+
- Lombok 1.18.22+ (JDK17 support)
- Gradle 7.3+

Dependency Injection

The basic idea of DI, a type of Inversion of Control (IoC), is that instead of having an object instantiate its needed dependencies, the dependencies are somehow given to the object. In a polymorphic way, the objects given as dependencies to the target object that depends on them are known to this target object just by an abstraction (like an interface in Java) and not by the exact implementation of the dependency.

The major advantages of the IoC architecture are

- Easier switching between different implementations
- Offering a good modularity of a program
- A great feature for testing programs by isolating components dependencies and allowing them to communicate through contracts
- Dividing the execution of a certain task from its implementation

It's easier to look at this in code than explain it; see Listing 2-1.

```
public class NonDiObject {

private Helper helper ;

public NonDiObject ( ) {

 helper = new HelperImpl ( ) ;
 }

public void doStuffWithHelp( ) {

 helper.help( ) ;

 }

}
```

Listing 2-1
The Object Itself Instantiates Its Dependencies (No Dependency Injection)

In this example, every instance of `NonDiObject` is responsible for instantiating its own `Helper` in the constructor. It instantiates a `HelperImpl`, creating a tight, unnecessary coupling to this particular `Helper` implementation (see Listing 2-2).

```
public class DiObject {

private Helper helper ;

public DiObject(Helper helper) {
 this.helper = helper;

 }

public void doStuffWithHelp( ) {

 helper.help( ) ;

 }

}
```

Listing 2-2

The Object Receives Its Dependencies from Some External Source (with Dependency Injectio

In this version, `Helper` is passed to `DiObject` at construction time. `DiObject` is not required to instantiate any dependency. It doesn't need to know how to do that, what particular implementation type the `Helper` is, or where it comes from. It just needs a helper and uses it for whatever requirement it has.

The advantage of this approach should be clear. The second version is loosely coupled to the `Helper`, depending only on the `Helper` interface, allowing the concrete implementation to be decided at runtime and thus giving lots of flexibility to the design.

Spring dependency injection configuration is normally defined in XML files, although later versions have turned more to annotation-based and Java-based configurations.

Aspect-Oriented Programming

AOP is a technique for extracting cross-cutting concerns from the main application code and transversely applying them across the points where they are needed. Typical examples of AOP concerns are transactions, logging, and security.

The main idea is that you decouple the main business logic of your application from special-purpose concerns that are peripheral to this core logic, and then apply this functionality in a transparent, unobtrusive way through your application. By encapsulating this functionality (which is simply general application logic and not core business logic) in its own modules, they can be used by many parts of the application that need them, avoiding the need to duplicate this code all over the place. The entities encapsulating this cross-cutting logic are referred to as *aspects* in AOP terms.

There are many implementations of AOP in Java. The most popular, perhaps, is AspectJ, which requires a special compilation process. Spring supports AspectJ, but it also includes its own AOP implementation, known simply as Spring AOP, which is a pure Java implementation that requires no special compilation process.

Spring AOP using proxies is available only at the public-method level and only when it is called outside the proxied object. This makes sense because calling a method from inside the object won't call the proxy; instead, it calls the real self object directly (basically a call on the `this` object). This is very important to be aware of when working with Spring, and sometimes, novice Spring developers overlook it.

Even when using its own AOP implementation, Spring leverages the AspectJ syntax and concepts for defining Aspects.

Spring AOP is a big subject, but the principle behind the way it works is not difficult to understand. Spring AOP works with dynamically created proxy objects that take care of the AOP concerns around invocating your main business objects. You can think of the proxy and Spring AOP in general simply as a decorator pattern implementation, where your business object is the component and the AOP proxy is the decorator. Figure 2-1 shows a simple graphical representation of the concept. Thinking about it this way, you should be able to understand Spring AOP easily. Listing 2-3 shows how the magic happens conceptually.

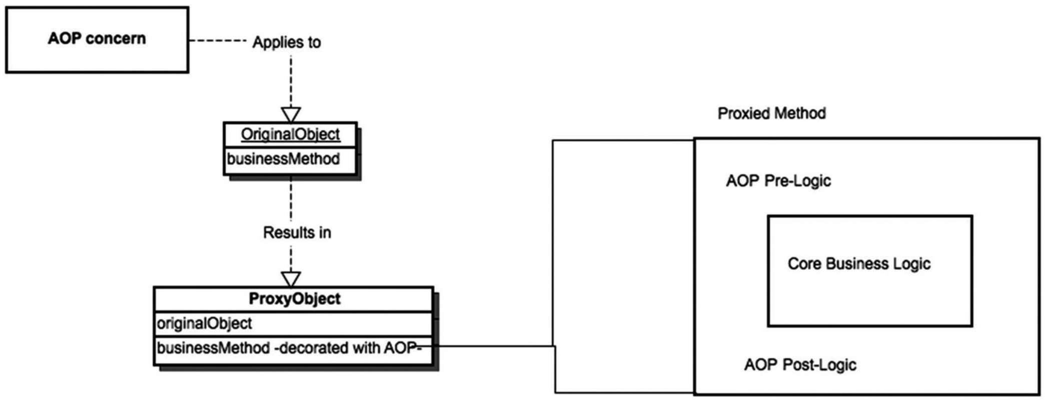

Figure 2-1 Spring AOP in action

```
public class Business Object implements BusinessThing {

public void doBusinessThing( ) {

 / / Some business stuff

 }

}
```

Listing 2-3 The Business Object, Not Transactional

Suppose you have an aspect for transactions. Spring creates dynamically at runtime an object that conceptually looks like Listing 2-4.

```
public class BusinessObjectTransactionalDecorator implements BusinessThing {

private BusinessThing componen t ;

public BusinessObjectTransactionalDecorator(BusinessThing component ) {

 t h i s . co mponent = component ;

 }

public void doBusinessThing( ) {

 / / some start transaction code

  component.doBusinessThing( ) ;

 / / some commit transaction code

 }

}
```

Listing 2-4

Spring AOP Magic

Again, remember this simple idea and Spring AOP should be easier to understand.

What's New in Spring Security 6?

The previous version of this book utilizes Spring Security 5. There are not massive changes from version 5 to 6. The following describes what's new in Spring Security 6.

- The Spring Boot 3 and Spring Security 6 baseline is now Java 17.
- The WebSecurityConfigurerAdapter class has been deprecated and removed in Spring Security 6, so now you must create a bean of type SecurityFilterChain.
- Instead of using authorizeRequests, which has been deprecated, you should now use authorizeHttpRequests, which is part of the HttpSecurity configuration allowing you to configure fine-grained request matching for access control.
- In Spring Security 6, AntMatcher, MvcMatcher, and RegexMatcher have been deprecated and replaced by requestMatchers or securityMatchers for path-based access control, allowing you to match requests based on patterns or other criteria without relying on specific matchers.

  Spring Security 5's main features are still valid and in use.

- By default, ContextPath is `/`. Use `/app_name` if you need to define a specific contextPath or use the via properties; for instance, `server.servlet.contextPath=/springbootapp`.
- The CSRF token filter has been added to the filter chain and turned on by default since version 3+.
- `j_username`/`j_password` parameters: Starting with version 4, you no longer receive the `username` value in the authentication request.

Plus, they were updated to `username` and `password`, removing the `j_` prefix.

- CSRF protection was added in version 5.
- Password encoding is mandatory in version 5.
- `web.xml` files are no longer needed starting with Servlet 3.0.
- Easier Spring Security configurations using Java configuration.
- There is an option to use a combination by setting the debug level to DEBUG in the Log4J2 configuration file.

If you need to migrate from version 5 to version 6, we recommend to follow the official Spring migration documentation at **https://docs.spring.io/spring-security/reference/migration/index.html**.

The following are some of the most important new functionalities included in the Spring Security 6.1.9.RELEASE.

- Compressing simple class name for observation
- Add new DaoAuthenticationProvider constructor
- Add NimbusJwtDecoder#withIssuerLocation
- Clarify documentation code snippet(s)
- Deprecate shouldFilterAllDispatcherTypes
- Document in the reference how to migrate to lambda
- Documentation should mention that an empty SecurityContext should also be saved
- Don't use raw XML SAML authentication request for response validation
- Ensure access token isn't resolved from query for form-encoded requests
- Expression-Based Access Control do not working as explain in Spring Security document for 6.0.2 also tried 6.0.5 the issue persist
- Remove OpenSaml deprecation warnings
- Replace deprecated OpenSaml methods
- Deprecate .and() along with non-lambda DSL methods

Spring Security 6 provides many new features.

The following lists the highlights.

- Core
  - SecuredAuthorizationManager allows customizing underlying AuthorizationManager
  - Add AuthorityCollectionAuthorizationManager
- OAuth 2.0
  - Add Nimbus(Reactive)JwtDecoder#withIssuerLocation
  - Configure principal claim name in ReactiveJwtAuthenticationConverter
- SAML 2.0
  - Support AuthnRequestSigned metadata attribute
  - Metadata supports multiple entities and EntitiesDescriptor
  - Add saml2Metadata to DSL
  - Allow Relying Party to be Deduced from LogoutRequest
  - Allow Relying Party to be Deduced from SAML Response
  - Add RelyingPartyRegistration placeholder resolution component
  - Support issuing LogoutResponse after already logged out
- Observability
  - Customize authentication and authorization observation conventions
- Web
  - Add RequestMatchers factory class
  - Propagate variables through And and OrRequestMatcher
- Docs
  - Revisit Authorization documentation
  - Revisit Session Management documentation
  - Revisit Logout documentation
  - Revisit CSRF Documentation

Spring Security 5 fundamentals are still in use; they include the following.

- Authentication confirms truth of credentials.
- Authorization defines access policy for principal.
- AuthenticationManager is a controller in the authentication process.
- AuthenticationProvider is an interface that maps to a data store which stores your user data.
- Authentication object is created upon authentication to hold the login credentials.

- GrantedAuthority means application permission is granted to a principal.
- Principal is the user that performs the action.
- SecurityContext holds the authentication and other security information.
- SecurityContextHolder: Provides access to SecurityContext.
- UserDetails is a data object that contains the user credentials and roles.
- UserDetailsService collects the user credentials and authorities (roles), and builds an UserDetails object.

HTTP, LDAP, JAAS API, and CAS are some of the most important technologies Spring Security 6 supports integration with.

Note The Spring Security 6.0.1.RELEASE can be downloaded at `https://github.com/spring-projects/spring-security/releases`.

Authentication and authorization are some of the fundamental functionalities in Spring Security 6. They are very important functionalities because they allow the Spring Security application to identify and authorize user, prevent unauthorized access, and control the user authorization to access application resources.

This book presents examples demonstrating how to develop an application to authorize and authenticate users.

The Spring Security authentication/authorization flow is shown in Figure 2-2.

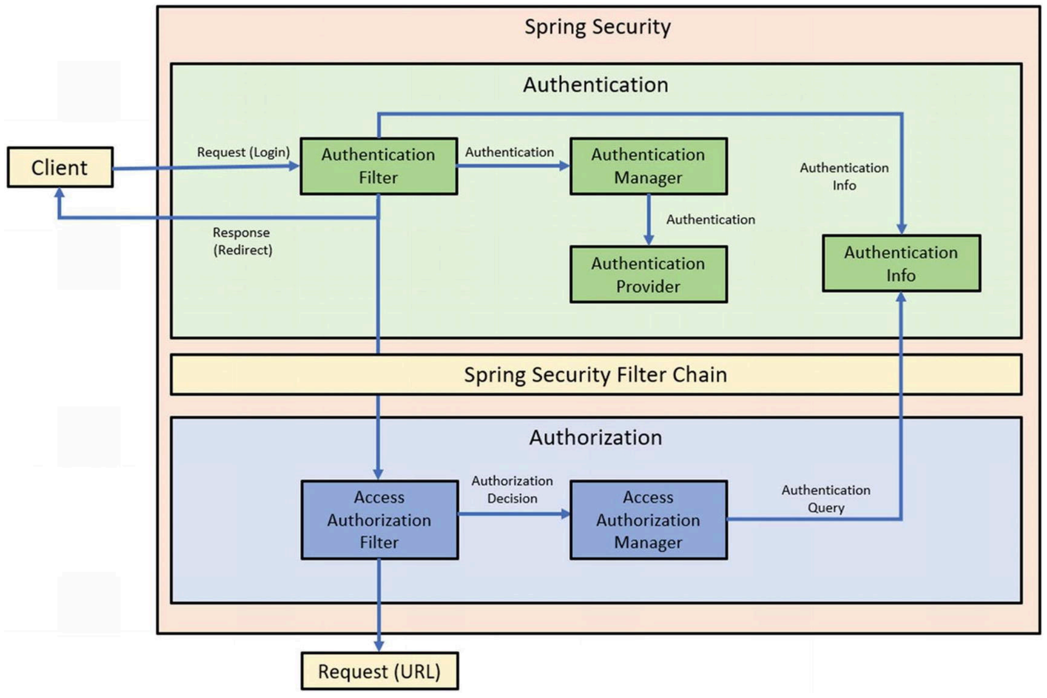

Figure 2-2 Spring Security authentication/authorization functionalities flow

Spring Security is utilized via some specific modules as JAR files. The `spring-security-core.jar` file contains the core.

- Authentication and access-control classes and interfaces
- Remoting support and basic provisioning APIs

JAR files are required by any application that uses Spring Security and supports the following.

- Stand-alone applications
- Remote clients
- Method (service layer) security
- JDBC user provisioning

The Spring Security 6 project's most important modules (JAR files) include

- Core: spring-security-core.jar
  - – org.springframework.security.core
  - – org.springframework.security.access
  - – org.springframework.security.authentication
  - – org.springframework.security.provisioning
- Remoting: spring-security-remoting.jar
- Web: spring-security-web.jar
- Config: spring-security-config.jar

- LDAP: spring-security-ldap.jar
- OAuth 2.0 Core: spring-security-oauth2-core.jar
- OAuth 2.0 Client: spring-security-oauth2-client.jar
- OAuth 2.0 JOSE: spring-security-oauth2-jose.jar
- OAuth 2.0 Resource Server: spring-security-oauth2-resource-server.jar
- ACL: spring-security-acl.jar
- CAS: spring-security-cas.jar
- Test: spring-security-test.jar
- Taglibs: spring-security-taglibs.jar

Spring Security XML and Java annotations can still be used in version 6 when developing Spring Security applications.

Summary

By now, you should understand what Spring Security is useful for. You also learned what's new in the Spring Security 6. Along the way, we introduced some of the major architectural and design principles behind it and how they are layered on top of the great Spring Framework 6. Dependency injection and AOP were also discussed.

The next chapter sets up the development scene, and you will build your first Spring Security–powered web application.