CHAPTER 5

Securing the Service Layer

This chapter will drill down further into the core functionality of Spring Security.

Unlike in the previous chapter where I focused only on the web-level access to the application, here you will see another important area in which Spring Security can help you secure your applications. It can be seen as a more invasive type of configuration because it involves securing at the code level, while the web layer simply was concerned with URL matching. However, you will see how the elegant way in which Spring can manage aspect-oriented programming (AOP) concerns will make business-level security as unobtrusive as its web-level counterpart.

The Limitations of Web-Level Security

In the previous chapter, you established security constraints at the web layer of the application, mainly at the URL level and the view-rendering layer. This is very nice and powerful; however, it is not a 100% effective solution in every use case. The main concerns with applying only this kind of security are both functional and convenience related, as I will explain next.

First, by definition, web-layer security applies only to web applications, which makes it unusable for any other kind of Java application. Although Spring Security’s main focus is on securing web applications, there is no reason why some of its parts can’t be used for different kinds of applications.

Second, the URL pattern-matching mechanism for security, although flexible, requires the developer to adopt certain rules or conventions just for the sake of security (like creating an /admin/ URL namespace for admin users and setting administrator rules on those URLs).

Third, securing at the URL level creates only a coarse-grained security, as a URL is the entry point into the application. This means that security constraints are enforced on a per-request basis, greatly reducing flexibility. For example, if you want to ensure that a particular Data Access Object (DAO) in your application is called only by an administrator user, you can’t do that with web security alone. You need to make sure that all the URLs that call this DAO are secured for Admin users. If, for some reason, you have a URL that is not secured correctly, that request will freely reach the DAO layer, where it could execute a potentially delicate operation.

You can surely find a few more reasons why web-only security is not enough sometimes, or simply not convenient enough to use by itself in your applications.

What Is Business Service-Level Security?

Service-layer security (or more accurately method-level security) is a feature of Spring Security you use to enforce security constraints at the method level, much as web-level security does at the URL level.

At its core, method-level security relies on Spring AOP’s powerful support for providing you with its services. This is the main difference in implementation with web-based security that depends on Servlet Filters, although it is worth noticing that under the hood most of the core code that will take care of the security constraints is the same. This is of great importance, as it shows good care in designing a set of reusable and encapsulated components in the architecture.
As I said, service-layer security is normally used in combination with web-based security, and I will cover this scenario mostly in this chapter. However, as you will see later, you can use service-layer security by itself without the context of a web application.

The traditional scenario for working with Spring Security is this: You have a web-based application, with a relatively thin web layer, backed by one Spring-implemented business service layer. With regard to security, the web layer is configured to take care of ensuring that there is a user authenticated in the system (that is, it takes care of the authentication part of the security, using forms, http status codes, and so on). The service layer has the authorization rules in per-operation criteria and with the needed level of granularity. Most of the time, you secure the business services; some other times, you might need to secure the DAOs.

Setting Up the Example for the Chapter

Let’s start doing some work and see how this whole thing works. You will be using the same application from last chapter, so make sure you have it at hand.

You will use Spring AOP in this chapter. So you need to add support for it in your application. The only thing you will do to support it at the moment is break your classes (the ones you want to decorate with security concerns) into interface-class hierarchies. For a start, you do this with the AdminController, creating an AdminController interface and an AdminControllerImpl implementation class, as you can see in Listing 5-1. You can see that I copied the Spring model view controller (MVC) annotations into this new interface. If I had not done this and had left them on the implementation only, Spring MVC wouldn’t find them when looking for a method to handle the incoming requests.

Note  Normally, in simple controllers, you can keep the @RequestMapping annotations on the implementing class without needing to create an interface. However, in the case of this example, you need to put the annotation on the interface because you will use security annotations in the class, which automatically will create a proxy object that needs the presence of an interface—for using standard Java Development Kit (JDK) proxy objects, as you will see in the upcoming paragraphs. This proxy won’t know about the @RequestMapping annotation, so Spring’s MVC mechanism won’t be able to find the handler methods. Later I will put the security annotations where they belong, in the service layer. I’m putting them now in the controller simply to illustrate the simplest scenario. In real life you would not do it this way and instead you would add it to the service layer as I will show you later in the chapter.

It is important anyway to bear in mind this behavior when working with Spring MVC or any other part of Spring that uses proxy objects around your objects.

Spring AOP comes in two flavors: standard java proxies that work with interfaces, and CGLIB support that creates proxies at the implementation-class level. Each has its strengths and weaknesses. They work in the following way.

- Using standard JDK proxies  If you have a Spring bean that implements an interface, and it should be enhanced with some AOP concern (for example, if it has configured Spring support of standard AOP concerns like transactionality, security, or synchronicity), Spring creates, when starting up and instantiating the beans, a new class that implements the same interfaces as your original bean, adds the specific functionality that it needs (transaction awareness, security, and so on) to that new class, and then wraps your original object in this new object, creating in essence a “decorated” version of your object. This new decorated object will be the actual bean used in the framework, transparently replacing the original object whenever it is accessed within the Spring application. Basically, if you inject your defined bean into another bean, the object instance that gets injected is the proxy.