Customizing and Extending Spring Security

Spring Security is a very extendable and customizable framework. This is primarily because the framework is built using object-oriented principles and design practices so that it is open for extension and closed for modification. In the previous chapter, you saw one of the major extension points in Spring Security—namely, the pluggability of different authentication providers. This chapter covers some other extension points in the framework that you can take advantage of to extend Spring Security’s functionality or to modify or customize functionality that doesn’t work exactly the way you need in your applications.

I also briefly cover the Spring Security Extensions project (http://static.springsource.org/spring-security/site/extensions.html), an environment you can use to create extension modules for the core Spring Security project.

The next section defines what I consider to be some of the major extension points in Spring Security and describes how to use them to add or modify behavior in your security solution.

Spring Security Extension Points

Spring Security offers a comprehensive set of extension points that can be customized (or completely overridden) with your own implementations and still leverage the core of the framework. Some of the extension points are evident, while some others are a bit more subtle and, in some cases, not even intended. However, because the framework is so flexible, you can take advantage of that flexibility to tweak its configuration to fit your intentions.

Plug into the Spring Security Event System

Spring Security supports an event model that is built on top of Spring Framework’s own event model. You can use Spring’s event model to develop applications that can listen to different events that happen within the framework and act accordingly.

I won’t explain in any depth why an event model is such a powerful programming practice to have at your disposal. Instead, I’ll just point out one big advantage: it allows you to decouple your applications, because in general the event producer or producers and the event consumer or consumers don’t need to know anything about each other in order to operate correctly. In theory (and, indeed, in practice for events in general, although not for Spring events), you can have a completely heterogeneous application where you can write and evolve each module at its own pace without affecting other parts, and then integrate them all together through the exclusive use of events.

All Spring events should extend from the abstract class org.springframework.context.ApplicationEvent, and Spring Security’s own events are no exception.
One of the main concrete implementations of the ApplicationEvent abstract class is 
org.springframework.context.event.ApplicationContextEvent, which itself serves as the parent class of a 
series of events that involve the life cycle of the application context (ContextClosedEvent, ContextRefreshedEvent, 
ContextStartedEvent, ContextStoppedEvent).

To register your application to be notified of Spring events, you need one (or more) of the beans defined in 
your application to implement the interface org.springframework.context.ApplicationListener\langle E \rangle extends 
ApplicationEvent>. This interface defines a single method, void onApplicationEvent(E event), that you can use to 
listen to a particular type of event in the application.

Publishing events is equally easy. You only need to define a bean in your Spring application context that 
implements the interface org.springframework.context.ApplicationEventPublisherAware, which again defines 
only one method:

```java
void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher)
```

This method is called automatically by Spring when the application starts up, and an instance of 
ApplicationEventPublisher (an implementation of it because it is an interface) is passed in. The instance of 
ApplicationEventPublisher that is passed in is normally the ApplicationContext instance itself that contains 
the application.

The default implementation of ApplicationEventPublisher's publishEvent method (which lives in 
the class AbstractApplicationContext) delegates the publishing of the events to an implementation of 
ApplicationEventMulticaster—the only current implementation of which is 
org.springframework.context.event.SimpleApplicationEventMulticaster.

Broadcasting an event to all interested listeners is also straightforward. You simply need to create an instance 
of one of the implementing classes of ApplicationEvent (any subclass of it will do as an event) and then call the 
ApplicationEventPublisher's publishEvent(ApplicationEvent event) method, passing your ApplicationEvent 
instance to it. Spring then takes care of ensuring that all the listeners registered for that particular event are notified 
of the event publication. By default, all listeners are invoked on the same thread as the publisher; however, you also 
could configure an org.springframework.core.task.TaskExecutor (which is an interface, so you could use an 
implementation class like org.springframework.core.task.SimpleAsyncTaskExecutor) to call the listeners in 
different threads. You will use this when you configure the listeners in our examples.

Spring Security comes with its own suite of ApplicationEvent implementations, so you can hook into different 
points of the security life cycle in an unobtrusive and decoupled way. The ApplicationEvent implementations that 
Spring Security provides are categorized as Authorization, Authentication, or javax.servlet.http.Session types, 
and they have descriptive names that hint what they do and where they are published. Here, I will give a concrete 
explanation of them and when they are published within the framework.

Events in Spring work as shown in Figure 8-1.

![Image of Spring event mechanism]

**Figure 8-1. Spring event mechanism. Spring Security has its own set of ApplicationEvent implementations**