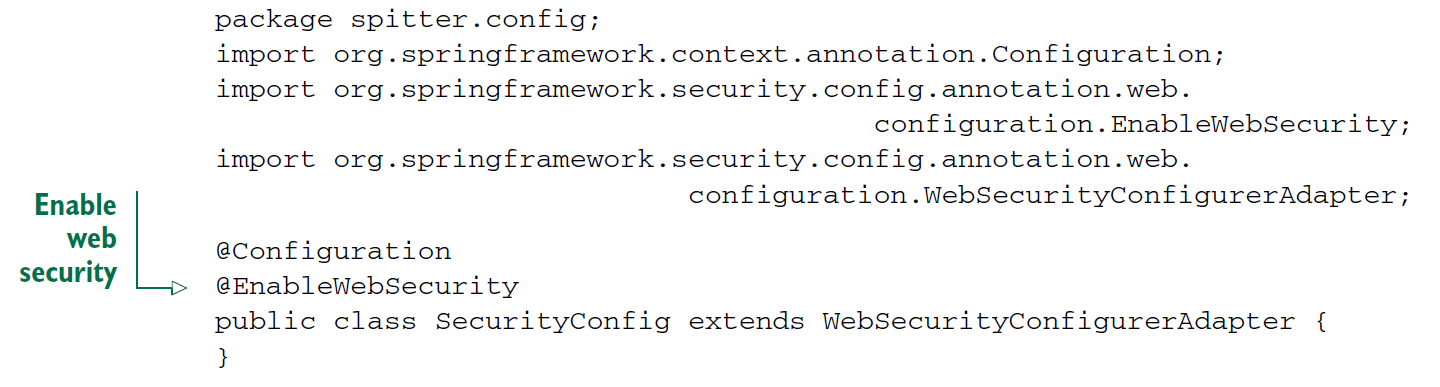
***Writing a simple security configuration***

In the early days of Spring Security (way back when it was known as Acegi Security), you’d need to write hundreds of lines of XML configuration just to enable simple security in a web application. Spring Security 2.0 made things better by offering a security specific XML configuration namespace.

Spring 3.2 introduced a new Java configuration option, altogether eliminating the need for XML security configuration. The following listing shows the simplest possible Java configuration for Spring Security.

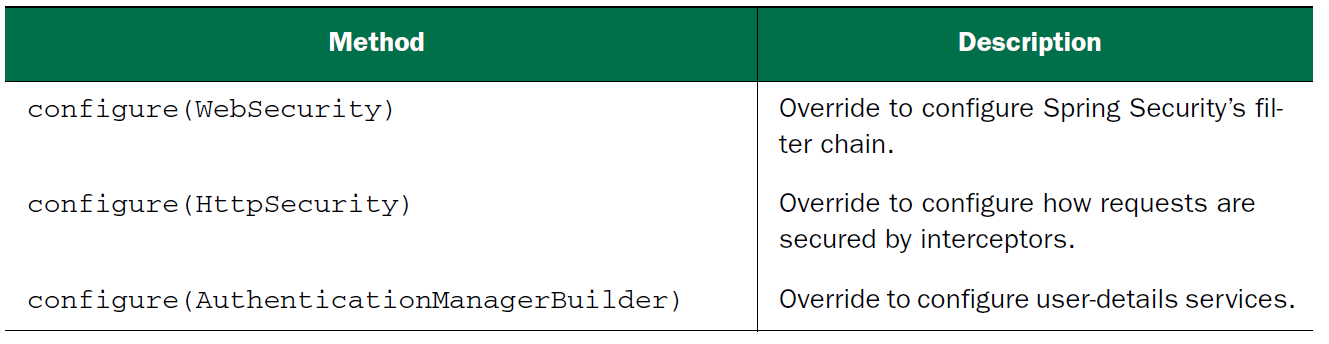


As its name suggests, the @EnableWebSecurity annotation enables web security. It is useless on its own, however. Spring Security must be configured in a bean that implements WebSecurityConfigurer or (for convenience) extends WebSecurityConfigurerAdapter.

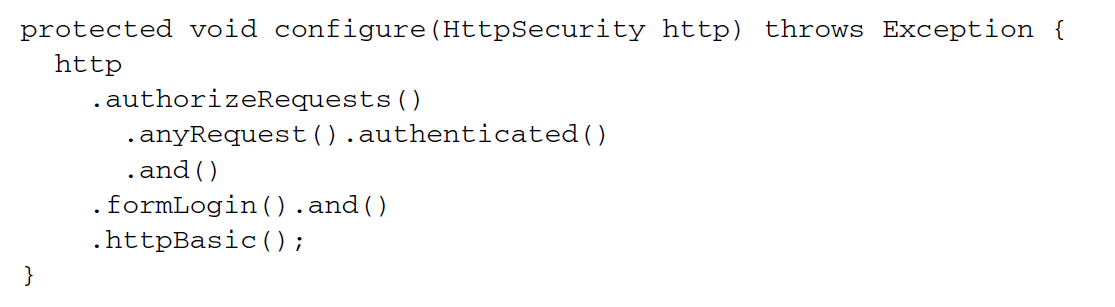
Any bean in the Spring application context that implements WebSecurityConfigurer can contribute to Spring Security configuration, but it’s often most convenient for the configuration class to extend WebSecurityConfigurerAdapter

* You can configure web security by overriding WebSecurityConfigurerAdapter’s three configure() methods and setting behavior on the parameter passed in.

**Overriding WebSecurityConfigurerAdapter’s configure() methods**



* If you don’t override any of the these three cofigure() methods the whole application will be locked down tight.
* The default configure(HttpSecurity) effectively looks like this:



* This simple default configuration specifies how HTTP requests should be secured and what options a client has for authenticating the user. The call to authorizeRequests() and anyRequest().authenticated() demands that all HTTP requests coming into the application be authenticated.
* It also configures Spring Security to
* Support authentication via a form-based login (using a predefined login page) as well as HTTP Basic.
* Meanwhile, because you haven’t overridden the configure(AuthenticationManagerBuilder) method, there’s no user store backing the authentication process. With no user store, there are effectively no users. Therefore, all requests require authentication, but there’s nobody who can log in.
* You’re going to need to add a bit more configuration to bend Spring Security to fit your application’s needs. Specifically, you’ll need to…
* Configure a user store
* Specify which requests should and should not require authentication, as well as what authorities they require
* Provide a custom login screen to replace the plain default login screen

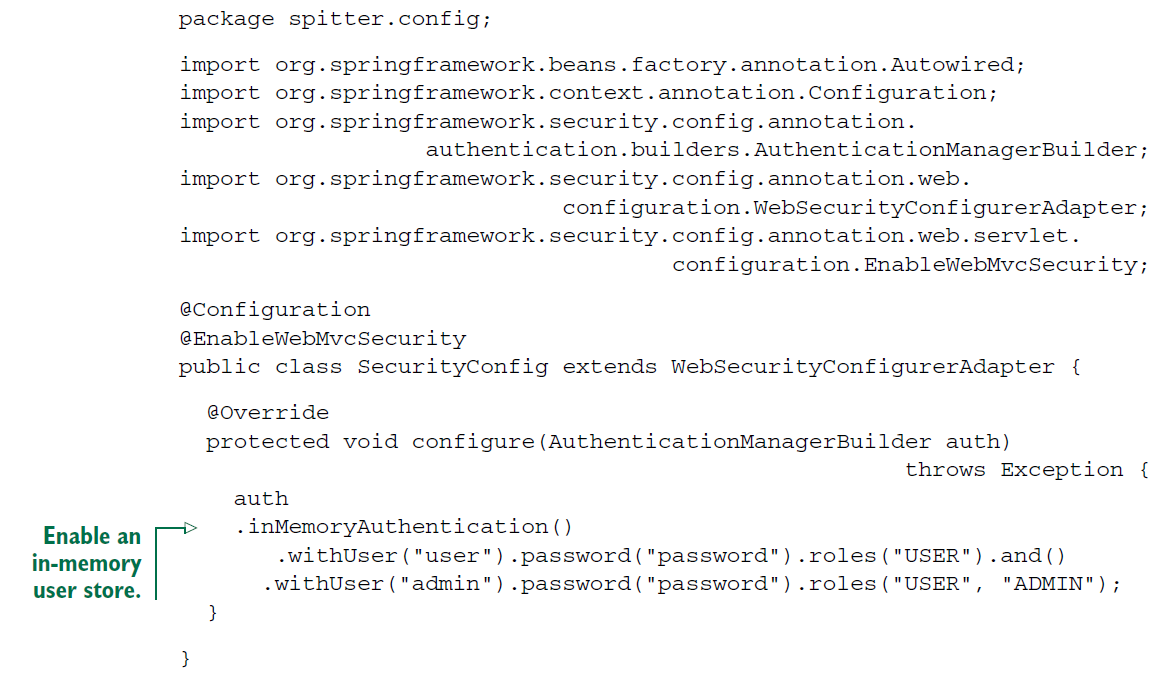
***Selecting user details services***

* What you need is a user store—some place where usernames, passwords, and other data can be kept and retrieved from when making authentication decisions.
* Fortunately, Spring Security is extremely flexible and is capable of authenticating users against virtually any data store. Several common user store situations—such as in-memory, relational database, and LDAP—are provided out of the box. But you can also create and plug in custom user store implementations.
* We’ll start with the simplest user store: one that maintains its user store in memory.

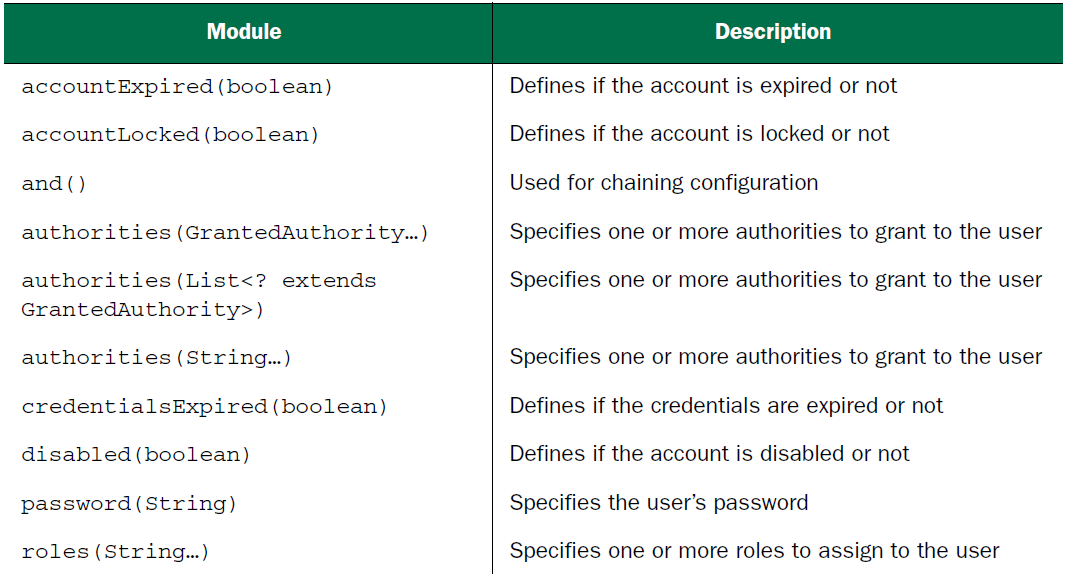
***Working with an in-memory user store***

Since your security configuration class extends WebSecurityConfigurerAdapter, the easiest way to configure a user store is to override the configure() method that takes an AuthenticationManagerBuilder as a parameter. AuthenticationManagerBuilder has several methods that can be used to configure Spring Security’s authentication support. With the inMemoryAuthentication() method, you can enable and configure and optionally populate an in-memory user store.

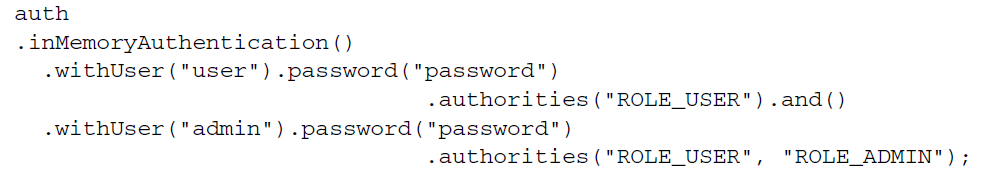
Configuring Spring Security to use an in-memory user store

****

* As you can see, the AuthenticationManagerBuilder given to configure() employs a builder-style interface to build up authentication configuration.
* Simply calling inMemoryAuthentication() will enable an in-memory user store. But you’ll also need some users in there, or else it’s as if you have no user store at all. Therefore, you need to call the withUser() method to add a new user to the inmemory user store. The parameter given is the username. withUser() returns a UserDetailsManagerConfigurer.UserDetailsBuilder,which has several methods for further configuration of the user, including password() to set the user’s password and roles() to give the user one or more role authorities.
* You’re adding two users, “user” and “admin”, both with “password” for a password. The “user” user has the USER role, while the “admin” user has both USER and ADMIN roles. As you can see, the and() method is used to chain together multiple user configurations.
* There are several other methods for configuring user details for in-memory user stores.

****

* Note that the roles() method is a shortcut for the authorities() methods. Any values given to roles() are prefixed with ROLE\_ and assigned as authorities to the user.
* In effect, the following user configuration is equivalent to previous one:

****

Although an in-memory user store is very useful for debugging and developer testing purposes, it’s probably not the most ideal choice for a production application. For production-ready purposes, it’s usually better to maintain user data in a database of some sort.