***Intercepting requests***

In any given application, not all requests should be secured equally. Some may require authentication; some may not. Some requests may only be available to users

with certain authorities and unavailable to those without those authorities.

For example, consider the requests served by the Spittr application. Certainly, the

home page is public and doesn’t need to be secured. Likewise, since all Spittle

objects are essentially public, the pages that display Spittle s don’t require security.

Requests that create a Spittle , however, should only be performed by an authenti-

cated user. Similarly, although user profile pages are public and don’t require authentication, if you were to handle a request for /spitters/me to display the current user’s profile, then authentication is required to know whose profile to show.

The key to fine-tuning security for each request is to override the configure

(HttpSecurity) method. The following code snippet shows how you might override

configure(HttpSecurity) to selectively apply security to different URL paths.



The HttpSecurity object given to configure() can be used to configure several aspects of HTTP security. Here you’re calling authorizeRequests() and then calling methods on the object it returns to indicate that you want to configure request-level security details. The first call to antMatchers() specifies that requests whose path is /spitters/me should be authenticated. The second call to antMatchers() is even more specific, saying that any HTTP POST request to /spittles must be authenticated. Finally, a call to anyRequests() says that all other requests should be permitted, not requiring authentication or any authorities.

The path given to antMatchers() supports Ant-style wildcarding. Although we’re not using it here, you could specify a path with a wildcard like this:



You could also specify multiple paths in a single call to antMatchers():

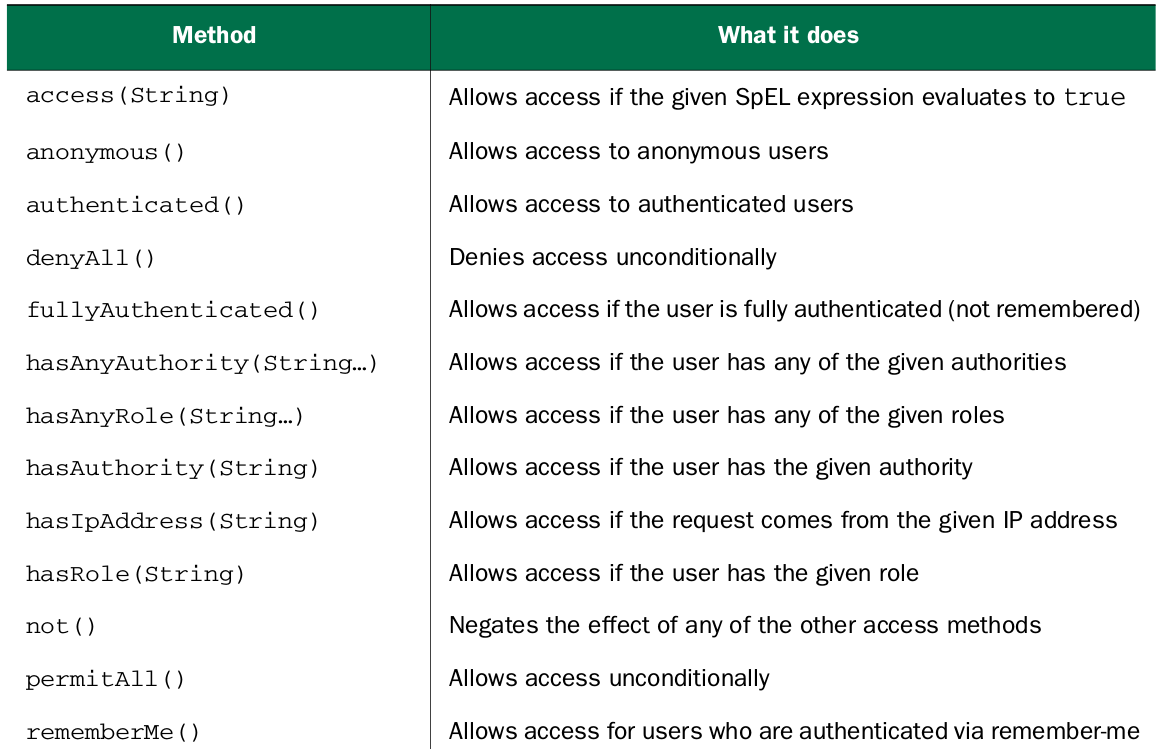


Whereas the antMatchers() method works with paths that may contain Ant-style wildcards, there’s also a regexMatchers() method that accepts regular expressions to define request paths. For example, the following snippet uses a regular expression that’s equivalent to /spitters/\*\* (Ant-style):



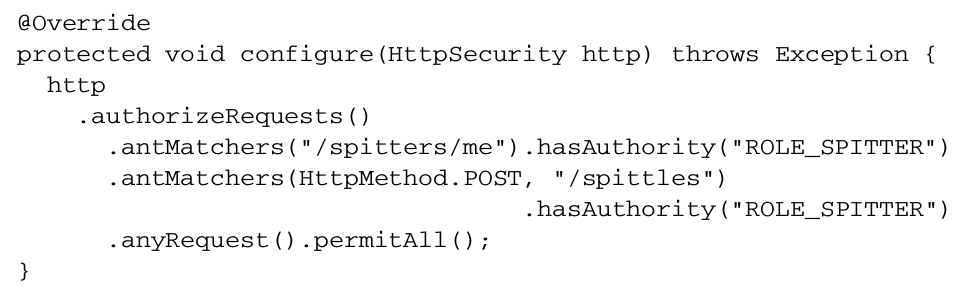
Aside from path selection, we’ve also used authenticated() and permitAll() to define how the paths should be secured. The authenticated() method demands that the user have logged into the application to perform the request. If the user isn’t authenticated, Spring Security’s filters will capture the request and redirect the user to the application’s login page. Meanwhile, the permitAll() method allows the requests without any security demands.

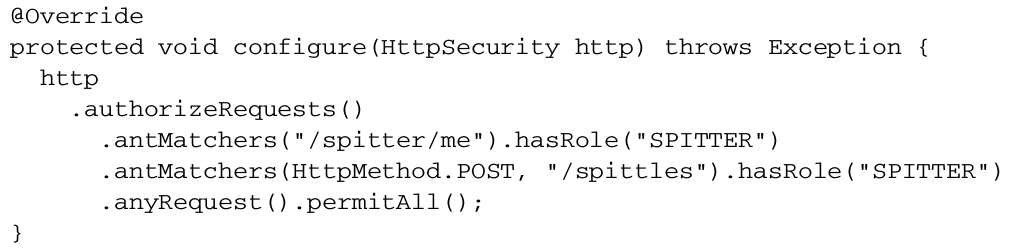
In addition to authenticated() and permitAll() , there are other methods that can be used to define how a request should be secured.



Using methods from table you can configure security to require more than just an

authenticated user. For example, you could change the previous configure() method to require that the user not only be authenticated, but also have ROLE\_SPITTER authority:

Optionally, you can use the hasRole() method to have the ROLE\_ prefix applied automatically:

You can chain as many calls to antMatchers() , regexMatchers() , and anyRequest() as you need to fully establish the security rules around your web application. You should know, however, that they’ll be applied in the order given. For that reason, it’s important to configure the most specific request path patterns first and the least specific ones (such as anyRequest() ) last. If not, then the least specific paths will trump the more specific ones.