Spring Security Architecture

This guide is a primer for Spring Security, offering insight into the design and basic building blocks of the framework. We cover only the very basics of application security. However, in doing so, we can clear up some of the confusion experienced by developers who use Spring Security. To do this, we take a look at the way security is applied in web applications by using filters and, more generally, by using method annotations. Use this guide when you need a high-level understanding of how a secure application works, how it can be customized, or if you need to learn how to think about application security.

This guide is not intended as a manual or recipe for solving more than the most basic problems (there are other sources for those), but it could be useful for beginners and experts alike. Spring Boot is also often referenced, because it provides some default behaviour for a secure application, and it can be useful to understand how that fits in with the overall architecture.

Note All of the principles apply equally well to applications that do not use Spring Boot.

Authentication and Access Control

Application security boils down to two more or less independent problems: authentication (who are you?) and authorization (what are you allowed to do?). Sometimes people say “access control” instead of "authorization", which can get confusing, but it can be helpful to think of it that way because “authorization” is overloaded in other places. Spring Security has an architecture that is designed to separate authentication from authorization and has strategies and extension points for both.

Authentication

The main strategy interface for authentication is `AuthenticationManager`, which has only one method:

```
public interface AuthenticationManager {

  Authentication authenticate(Authentication authentication)
    throws AuthenticationException;
}
```

An `AuthenticationManager` can do one of 3 things in its `authenticate()` method:

- Return an `Authentication` (normally with `authenticated=true`) if it can verify that the input represents a valid principal.
- Throw an `AuthenticationException` if it believes that the input represents an invalid principal.
- Return `null` if it cannot decide.

`AuthenticationException` is a runtime exception. It is usually handled by an application in a generic way, depending on the style or purpose of the application. In other words, user code is not normally expected to catch and handle it. For example, a web UI might render a page that says that the authentication failed, and a backend HTTP service might send a 401 response, with or without a `WWW-Authenticate` header depending on the context.

The most commonly used implementation of `AuthenticationManager` is `ProviderManager`, which delegates to a chain of `AuthenticationProvider` instances. An `AuthenticationProvider` is a bit like an `AuthenticationManager`, but it has an extra method to allow the caller to query whether it supports a given `Authentication` type:

```
public interface AuthenticationProvider {

	Authentication authenticate(Authentication authentication)
			throws AuthenticationException;

	boolean supports(Class<?> authentication);
}
```

The `Class<?>` argument in the `supports()` method is really `Class<? extends Authentication>` (it is only ever asked if it supports something that

is passed into the `authenticate()` method). A `ProviderManager` can support multiple different authentication mechanisms in the same application by delegating to a chain of `AuthenticationProviders`. If a `ProviderManager` does not recognize a particular `Authentication` instance type, it is skipped.

A `ProviderManager` has an optional parent, which it can consult if all providers return `null`. If the parent is not available, a `null` `Authentication` results in an `AuthenticationException`.

Sometimes, an application has logical groups of protected resources (for example, all web resources that match a path pattern, such as `/api/**`), and each group can have its own dedicated `AuthenticationManager`. Often, each of those is a `ProviderManager`, and they share a parent. The parent is then a kind of "global" resource, acting as a fallback for all providers.

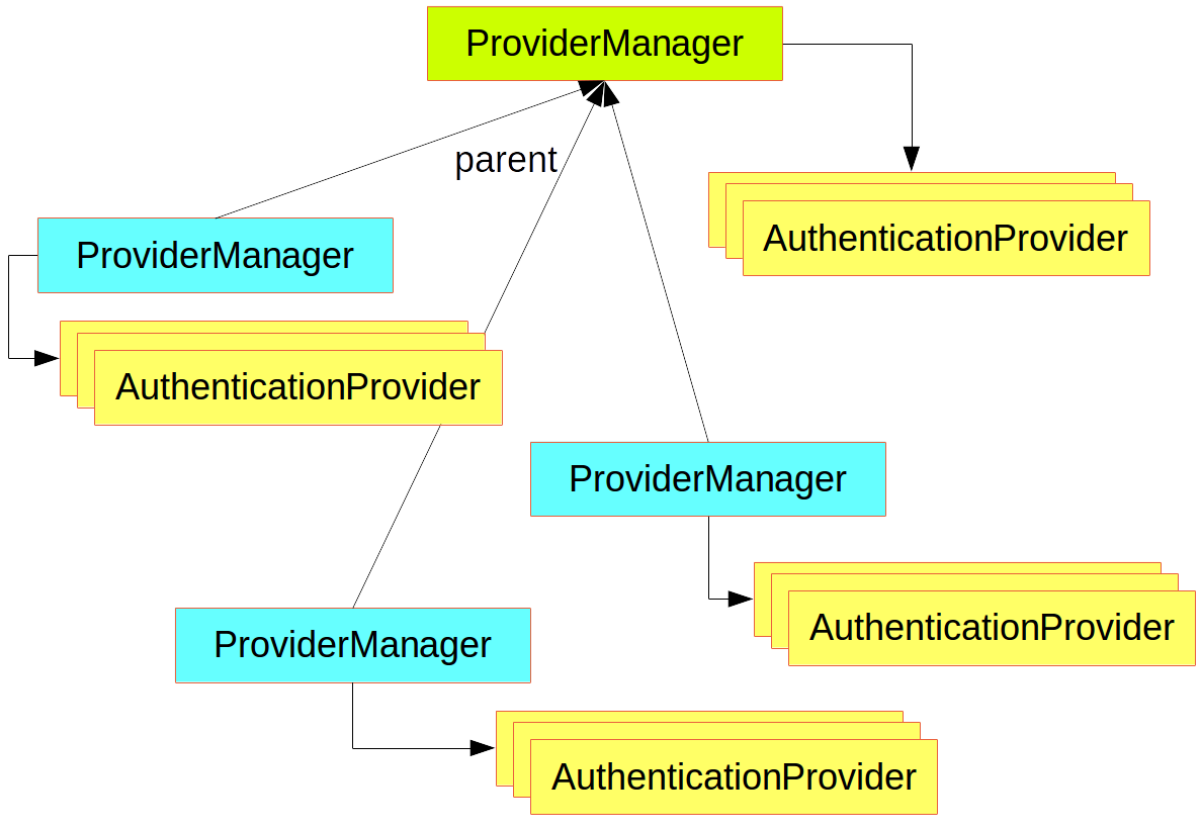


Figure 1. An `AuthenticationManager` hierarchy using `ProviderManager`

Customizing Authentication Managers

Spring Security provides some configuration helpers to quickly get common authentication manager features set up in your application. The most commonly used helper is the `AuthenticationManagerBuilder`, which is great for setting up in-memory, JDBC, or LDAP user details or for adding a custom `UserDetailsService`. The following example shows an application that configures the global (parent) `AuthenticationManager`:

```
@Configuration
public class ApplicationSecurity extends
WebSecurityConfigurerAdapter {

   ... // web stuff here

  @Autowired
  public void initialize(AuthenticationManagerBuilder builder,
DataSource dataSource) {

builder.jdbcAuthentication().dataSource(dataSource).withUser("dav
e")
       .password("secret").roles("USER");
  }

}
```

This example relates to a web application, but the usage of `AuthenticationManagerBuilder` is more widely applicable (see Web Security for more detail on how web application security is implemented). Note that the `AuthenticationManagerBuilder` is `@Autowired` into a method in a `@Bean` — that is what makes it build the global (parent) `AuthenticationManager`. In contrast, consider the following example:

```
@Configuration
public class ApplicationSecurity extends
WebSecurityConfigurerAdapter {

  @Autowired
  DataSource dataSource;

   ... // web stuff here

  @Override
  public void configure(AuthenticationManagerBuilder builder) {
```

```
builder.jdbcAuthentication().dataSource(dataSource).withUser("dav
e")
       .password("secret").roles("USER");
  }

}
```

If we had used an `@Override` of a method in the configurer, the `AuthenticationManagerBuilder` would be used only to build a "local" `AuthenticationManager`, which would be a child of the global one. In a Spring Boot application, you can `@Autowired` the global one into another bean, but you cannot do that with the local one unless you explicitly expose it yourself.

Spring Boot provides a default global `AuthenticationManager` (with only one user) unless you pre-empt it by providing your own bean of type `AuthenticationManager`. The default is secure enough on its own for you not to have to worry about it much, unless you actively need a custom global `AuthenticationManager`. If you do any configuration that builds an `AuthenticationManager`, you can often do it locally to the resources that you are protecting and not worry about the global default.

Authorization or Access Control

Once authentication is successful, we can move on to authorization, and the core strategy here is `AccessDecisionManager`. There are three implementations provided by the framework and all three delegate to a chain of `AccessDecisionVoter` instances, a bit like the `ProviderManager` delegates to `AuthenticationProviders`.

An `AccessDecisionVoter` considers an `Authentication` (representing a principal) and a secure `Object`, which has been decorated with `ConfigAttributes`:

```
boolean supports(ConfigAttribute attribute);

boolean supports(Class<?> clazz);

int vote(Authentication authentication, S object,
```

```
        Collection<ConfigAttribute> attributes);
```

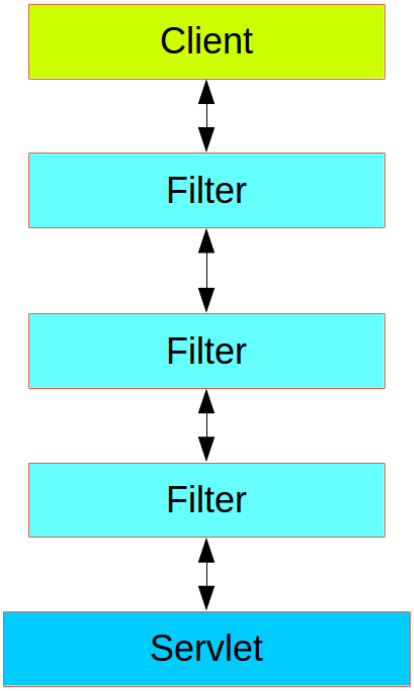
The `Object` is completely generic in the signatures of the `AccessDecisionManager` and `AccessDecisionVoter`. It represents anything that a user might want to access (a web resource or a method in a Java class are the two most common cases). The `ConfigAttributes` are also fairly generic, representing a decoration of the secure `Object` with some metadata that determines the level of permission required to access it. `ConfigAttribute` is an interface. It has only one method (which is quite generic and returns a `String`), so these strings encode in some way the intention of the owner of the resource, expressing rules about who is allowed to access it. A typical `ConfigAttribute` is the name of a user role (like `ROLE_ADMIN` or `ROLE_AUDIT`), and they often have special formats (like the `ROLE_` prefix) or represent expressions that need to be evaluated.

Most people use the default `AccessDecisionManager`, which is `AffirmativeBased` (if any voters return affirmatively, access is granted). Any customization tends to happen in the voters, either by adding new ones or modifying the way that the existing ones work.

It is very common to use `ConfigAttributes` that are Spring Expression Language (SpEL) expressions — for example, `isFullyAuthenticated() && hasRole('user')`. This is supported by an `AccessDecisionVoter` that can handle the expressions and create a context for them. To extend the range of expressions that can be handled requires a custom implementation of `SecurityExpressionRoot` and sometimes also `SecurityExpressionHandler`.

Web Security

Spring Security in the web tier (for UIs and HTTP back ends) is based on Servlet `Filters`, so it is helpful to first look at the role of `Filters` generally. The following picture shows the typical layering of the handlers for a single HTTP request.



The client sends a request to the application, and the container decides which filters and which servlet apply to it based on the path of the request URI. At most, one servlet can handle a single request, but filters form a chain, so they are ordered. In fact, a filter can veto the rest of the chain if it wants to handle the request itself. A filter can also modify the request or the response used in the downstream filters and servlet. The order of the filter chain is very important, and Spring Boot manages it through two mechanisms: `@Beans` of type `Filter` can have an `@Order` or implement `Ordered`, and they can be part of a `FilterRegistrationBean` that itself has an order as part of its API. Some off-the-shelf filters define their own constants to help signal what order they like to be in relative to each other (for example, the `SessionRepositoryFilter` from Spring Session has a `DEFAULT_ORDER` of `Integer.MIN_VALUE + 50`, which tells us it likes to be early in the chain, but it does not rule out other filters coming before it).

Spring Security is installed as a single `Filter` in the chain, and its concrete type is `FilterChainProxy`, for reasons that we cover soon. In a Spring Boot application, the security filter is a `@Bean` in the `ApplicationContext`, and it is

installed by default so that it is applied to every request. It is installed at a position defined by `SecurityProperties.DEFAULT_FILTER_ORDER`, which in turn is anchored by `FilterRegistrationBean.REQUEST_WRAPPER_FILTER_MAX_ORDER` (the maximum order that a Spring Boot application expects filters to have if they wrap the request, modifying its behavior). There is more to it than that, though: From the point of view of the container, Spring Security is a single filter, but, inside of it, there are additional filters, each playing a special role. The following image shows this relationship:

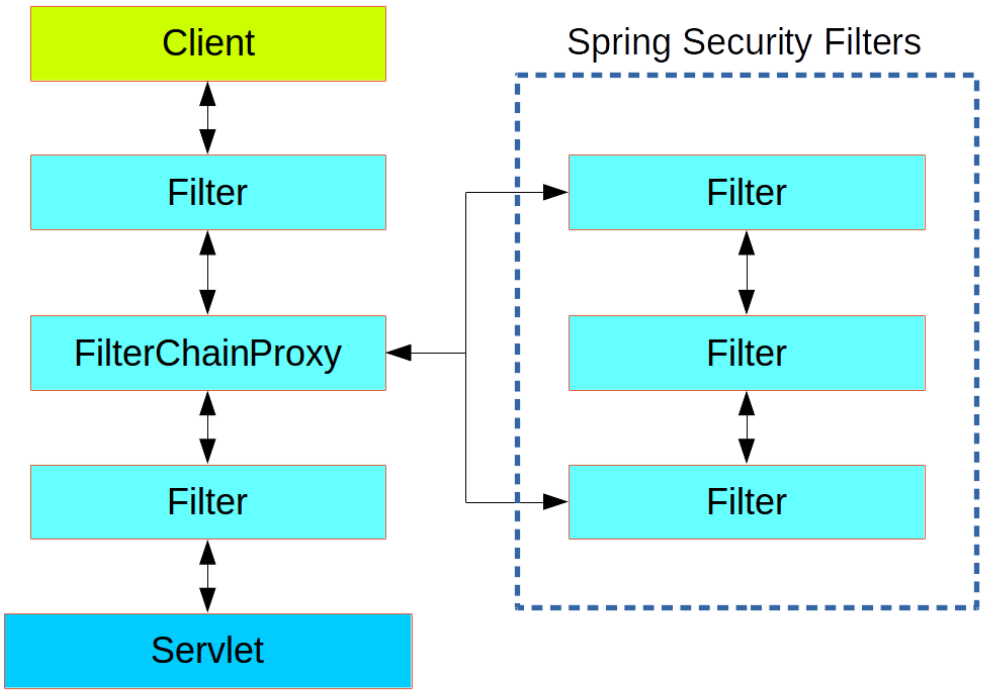


Figure 2. Spring Security is a single physical `Filter` but delegates processing to a chain of internal filters

In fact, there is even one more layer of indirection in the security filter: It is usually installed in the container as a `DelegatingFilterProxy`, which does not have to be a Spring `@Bean`. The proxy delegates to a `FilterChainProxy`, which is always a `@Bean`, usually with a fixed name of `springSecurityFilterChain`. It is the `FilterChainProxy` that contains all

the security logic arranged internally as a chain (or chains) of filters. All the filters have the same API (they all implement the `Filter` interface from the Servlet specification), and they all have the opportunity to veto the rest of the chain.

There can be multiple filter chains all managed by Spring Security in the same top level `FilterChainProxy` and all are unknown to the container. The Spring Security filter contains a list of filter chains and dispatches a request to the first chain that matches it. The following picture shows the dispatch happening based on matching the request path (`/foo/**` matches before `/**`). This is very common but not the only way to match a request. The most important feature of this dispatch process is that only one chain ever handles a request.

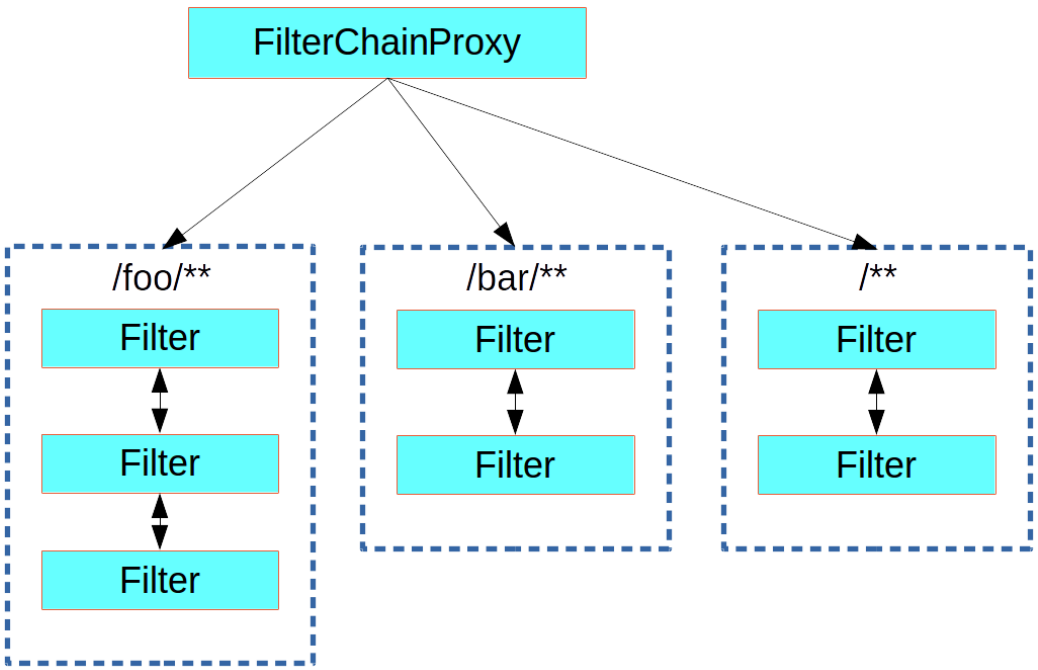


Figure 3. The Spring Security `FilterChainProxy` dispatches requests to the first chain that matches.

A vanilla Spring Boot application with no custom security configuration has a several (call it n) filter chains, where usually n=6. The first (n-1) chains are there just to ignore static resource patterns, like `/css/**` and `/images/**`, and the error view: `/error`. (The paths can be controlled by the user

with `security.ignored` from the `SecurityProperties` configuration bean.) The last chain matches the catch-all path (`/**`) and is more active, containing logic for authentication, authorization, exception handling, session handling, header writing, and so on. There are a total of 11 filters in this chain by default, but normally it is not necessary for users to concern themselves with which filters are used and when.

Note The fact that all filters internal to Spring Security are unknown to the container is important, especially in a Spring Boot application, where, by default, all `@Beans` of type `Filter` are registered automatically with the container. So if you want to add a custom filter to the security chain, you need to either not make it be a `@Bean` or wrap it in a `FilterRegistrationBean` that explicitly disables the container registration.

Creating and Customizing Filter Chains

The default fallback filter chain in a Spring Boot application (the one with the `/**` request matcher) has a predefined order of `SecurityProperties.BASIC_AUTH_ORDER`. You can switch it off completely by setting `security.basic.enabled=false`, or you can use it as a fallback and define other rules with a lower order. To do the latter, add a `@Bean` of type `WebSecurityConfigurerAdapter` (or `WebSecurityConfigurer`) and decorate the class with `@Order`, as follows:

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/match1/**")
     ...;
  }
}
```

This bean causes Spring Security to add a new filter chain and order it before the fallback.

Many applications have completely different access rules for one set of resources compared to another. For example, an application that hosts a UI and a backing API might support cookie-based authentication with a redirect to a login page for the UI parts and token-based authentication with a 401 response to unauthenticated requests for the API parts. Each set of resources has its own `WebSecurityConfigurerAdapter` with a unique order and its own request matcher. If the matching rules overlap, the earliest ordered filter chain wins.

Request Matching for Dispatch and Authorization

A security filter chain (or, equivalently, a `WebSecurityConfigurerAdapter`) has a request matcher that is used to decide whether to apply it to an HTTP request. Once the decision is made to apply a particular filter chain, no others are applied. However, within a filter chain, you can have more fine-grained control of authorization by setting additional matchers in the `HttpSecurity` configurer, as follows:

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/match1/**")
      .authorizeRequests()
        .antMatchers("/match1/user").hasRole("USER")
        .antMatchers("/match1/spam").hasRole("SPAM")
        .anyRequest().isAuthenticated();
  }
}
```

One of the easiest mistakes to make when configuring Spring Security is to forget that these matchers apply to different processes. One is a request matcher for the whole filter chain, and the other is only to choose the access rule to apply.

Combining Application Security Rules with Actuator Rules

If you use the Spring Boot Actuator for management endpoints, you probably want them to be secure, and, by default, they are. In fact, as soon as you add the Actuator to a secure application, you get an additional filter chain that applies only to the actuator endpoints. It is defined with a request matcher that matches only actuator endpoints and it has an order of `ManagementServerProperties.BASIC_AUTH_ORDER`, which is 5 fewer than the default `SecurityProperties` fallback filter, so it is consulted before the fallback.

If you want your application security rules to apply to the actuator endpoints, you can add a filter chain that is ordered earlier than the actuator one and that has a request matcher that includes all actuator endpoints. If you prefer the default security settings for the actuator endpoints, the easiest thing is to add your own filter later than the actuator one, but earlier than the fallback (for example, `ManagementServerProperties.BASIC_AUTH_ORDER + 1`), as follows:

```
@Configuration
@Order(ManagementServerProperties.BASIC_AUTH_ORDER + 1)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/foo/**")
      ...;
  }
}
```

Note Spring Security in the web tier is currently tied to the Servlet API, so it is only really applicable when running an application in a servlet container, either embedded or otherwise. It is not, however, tied to Spring MVC or the rest of the Spring web stack, so it can be used in any servlet application — for instance, one using JAX-RS.

Method Security

As well as support for securing web applications, Spring Security offers support for applying access rules to Java method executions. For Spring Security, this is just a different type of “protected resource”. For users, it means the access rules are declared using the same format of `ConfigAttribute` strings (for example,

roles or expressions) but in a different place in your code. The first step is to enable method security — for example, in the top level configuration for our application:

```
@SpringBootApplication
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SampleSecureApplication {
}
```

Then we can decorate the method resources directly:

```
@Service
public class MyService {

  @Secured("ROLE_USER")
  public String secure() {
    return "Hello Security";
  }

}
```

This example is a service with a secure method. If Spring creates a `@Bean` of this type, it is proxied and callers have to go through a security interceptor before the method is actually executed. If access is denied, the caller gets an `AccessDeniedException` instead of the actual method result.

There are other annotations that you can use on methods to enforce security constraints, notably `@PreAuthorize` and `@PostAuthorize`, which let you write expressions containing references to method parameters and return values, respectively.

Tip: It is not uncommon to combine Web security and method security. The filter chain provides the user experience features, such as authentication and redirect to login pages and so on, and the method security provides protection at a more granular level.

Working with Threads

Spring Security is fundamentally thread-bound, because it needs to make the current authenticated principal available to a wide variety of downstream

consumers. The basic building block is the `SecurityContext`, which may contain an `Authentication` (and when a user is logged in it is an `Authentication` that is explicitly `authenticated`). You can always access and manipulate the `SecurityContext` through static convenience methods in `SecurityContextHolder`, which, in turn, manipulate a `ThreadLocal`. The following example shows such an arrangement:

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
assert(authentication.isAuthenticated);
```

It is **not** common for user application code to do this, but it can be useful if you, for instance, need to write a custom authentication filter (although, even then, there are base classes in Spring Security that you can use so that you could avoid needing to use the `SecurityContextHolder`).

If you need access to the currently authenticated user in a web endpoint, you can use a method parameter in a `@RequestMapping`, as follows:

```
@RequestMapping("/foo")
public String foo(@AuthenticationPrincipal User user) {
  ... // do stuff with user
}
```

This annotation pulls the current `Authentication` out of the `SecurityContext` and calls the `getPrincipal()` method on it to yield the method parameter. The type of the `Principal` in an `Authentication` is dependent on the `AuthenticationManager` used to validate the authentication, so this can be a useful little trick to get a type-safe reference to your user data.

If Spring Security is in use, the `Principal` from the `HttpServletRequest` is of type `Authentication`, so you can also use that directly:

```
@RequestMapping("/foo")
public String foo(Principal principal) {
  Authentication authentication = (Authentication) principal;
  User = (User) authentication.getPrincipal();
```

```
  ... // do stuff with user
}
```

This can sometimes be useful if you need to write code that works when Spring Security is not in use (you would need to be more defensive about loading the `Authentication` class).

Processing Secure Methods Asynchronously

Since the `SecurityContext` is thread-bound, if you want to do any background processing that calls secure methods (for example, with `@Async`), you need to ensure that the context is propagated. This boils down to wrapping the `SecurityContext` with the task (`Runnable`, `Callable`, and so on) that is executed in the background. Spring Security provides some helpers to make this easier, such as wrappers for `Runnable` and `Callable`. To propagate the `SecurityContext` to `@Async` methods, you need to supply an `AsyncConfigurer` and ensure the `Executor` is of the correct type:

```
@Configuration
public class ApplicationConfiguration extends
AsyncConfigurerSupport {

  @Override
  public Executor getAsyncExecutor() {
    return new
DelegatingSecurityContextExecutorService(Executors.newFixedThread
Pool(5));
  }

}
```