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.

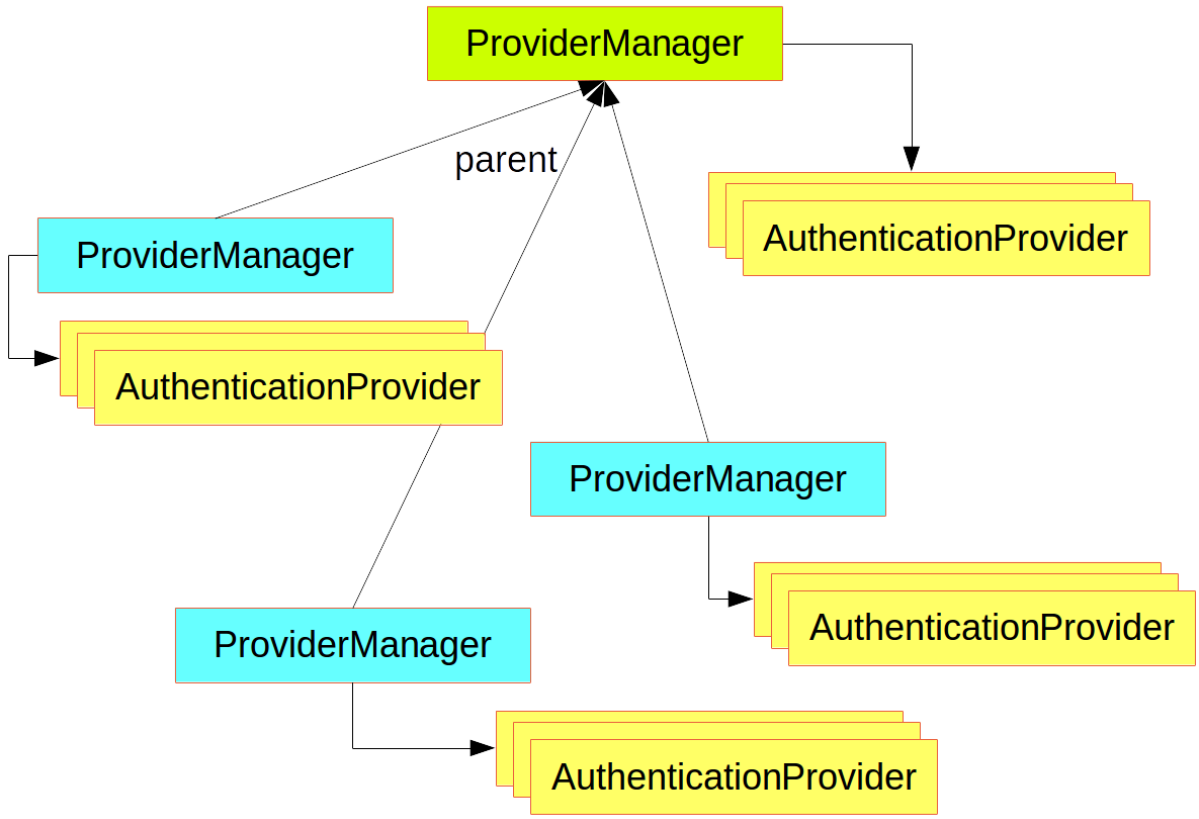


Figure 1. An `AuthenticationManager` hierarchy using `ProviderManager`

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);
```

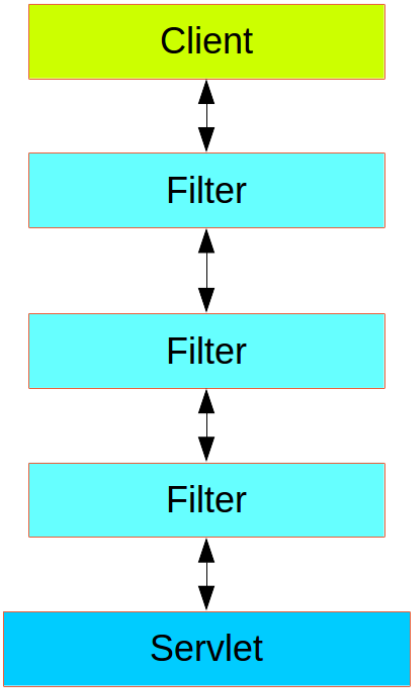
The `Object` is completely generic in the signatures of the `AccessDecisionManager` and `AccessDecisionVoter`. It represents anything that a user might want to access (a web resource or a method in a Java class are the two most common cases). The `ConfigAttributes` are also fairly generic, representing 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:

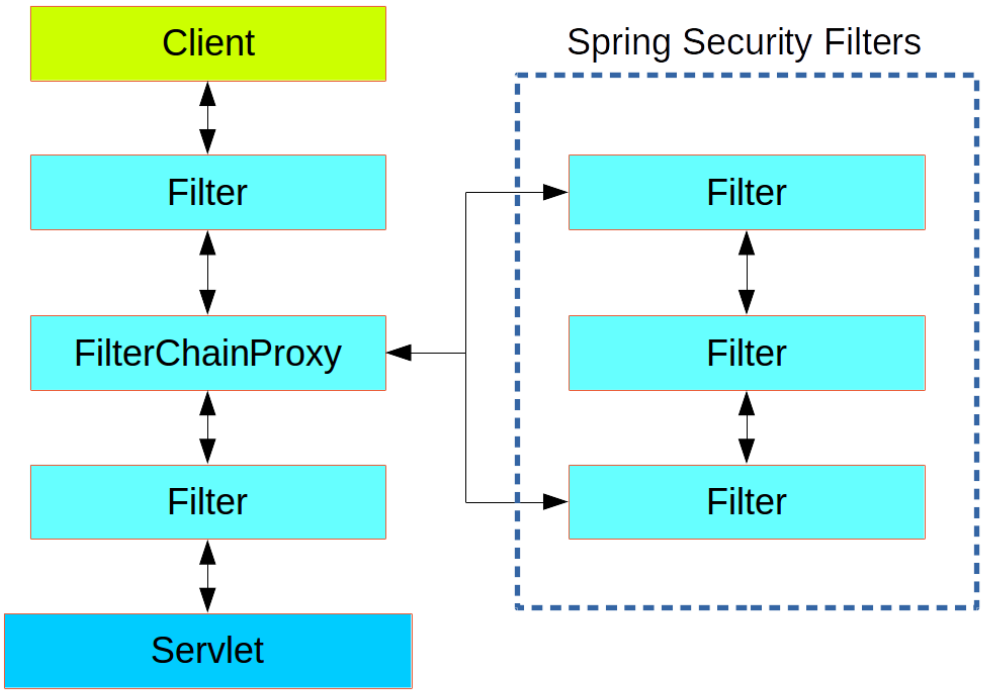


Figure 2. Spring Security is a single physical `Filter` but delegates processing to a chain of internal filters

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.

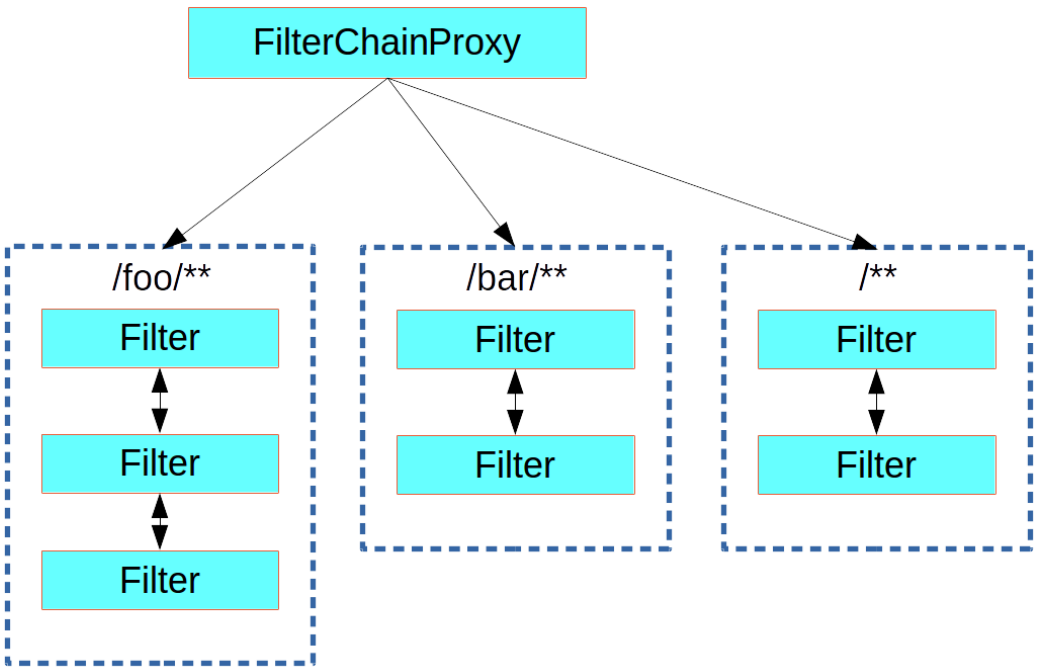


Figure 3. The Spring Security `FilterChainProxy` dispatches requests to the first chain that matches.

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));
  }

}
```

Spring Security Tutorial

Spring Security Tutorial provides basic and advanced concepts of Spring Security. Our Spring Security Tutorial is designed for beginners and professionals both.

Our Spring Security Tutorial includes all topics of Spring Security such as spring security introduction, features, project modules, xml example, java example, login logout, spring boot etc.

Prerequisite

To learn Spring Security, you must have the basic knowledge of HTML and CSS

Introduction

Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.

It is a sub-project of Spring framework which was started in 2003 by Ben Alex. Later on, in 2004, It was released under the Apache License as Spring Security 2.0.0.

It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application.

This framework targets two major areas of application are authentication and authorization. Authentication is the process of knowing and identifying the user that wants to access.

Authorization is the process to allow authority to perform actions in the application.

We can apply authorization to authorize web request, methods and access to individual domain.

Technologies that support Spring Security Integration

Spring Security framework supports wide range of authentication models. These models either provided by third parties or framework itself. Spring Security supports integration with all of these technologies.

- HTTP BASIC authentication headers
- HTTP Digest authentication headers
- HTTP X.509 client certificate exchange
- LDAP (Lighweight Directory Access Protocol)
- Form-based authentication

- OpenID authentication
- Automatic remember-me authentication
- Kerberos
- JOSSO (Java Open Source Single Sign-On)
- AppFuse
- AndroMDA
- Mule ESB
- DWR(Direct Web Request)

The beauty of this framework is its flexible authentication nature to integrate with any software solution. Sometimes, developers want to integrate it with a legacy system that does not follow any security standard, there Spring Security works nicely.

Advantages

Spring Security has numerous advantages. Some of that are given below.

- Comprehensive support for authentication and authorization.
- Protection against common tasks
- Servlet API integration
- Integration with Spring MVC
- Portability
- CSRF protection
- Java Configuration support

Spring Security History

In late 2003, a project **Acegi Security System for Spring** started with the intention to develop a Spring-based security system. So, a simple security system was implemented but not released officially. Developers used that code internally for their solutions and by 2004 about 20 developers were using that.

Initially, authentication module was not part of the project, around a year after, module was added and complete project was reconfigure to support more technologies.

After some time this project became a subproject of Spring framework and released as 1.0.0 in 2006.

in 2007, project is renamed to Spring Security and widely accepted. Currently, it is recognized and supported by developers open community world wide.

Spring Security Features

- LDAP (Lightweight Directory Access Protocol)
- Single sign-on
- JAAS (Java Authentication and Authorization Service) LoginModule
- Basic Access Authentication
- Digest Access Authentication
- Remember-me
- Web Form Authentication
- Authorization
- Software Localization
- HTTP Authorization

LDAP (Lightweight Directory Access Protocol)

It is an open application protocol for maintaining and accessing distributed directory information services over an Internet Protocol.

Single sign-on

This feature allows a user to access multiple applications with the help of single account(user name and password).

JAAS (Java Authentication and Authorization Service) LoginModule

It is a Pluggable Authentication Module implemented in Java. Spring Security supports it for its authentication process.

Basic Access Authentication

Spring Security supports Basic Access Authentication that is used to provide user name and password while making request over the network.

Digest Access Authentication

This feature allows us to make authentication process more secure than Basic Access Authentication. It asks to the browser to confirm the identity of the user before sending sensitive data over the network.

Remember-me

Spring Security supports this feature with the help of HTTP Cookies. It remember to the user and avoid login again from the same machine until the user logout.

Web Form Authentication

In this process, web form collect and authenticate user credentials from the web browser. Spring Security supports it while we want to implement web form authentication.

Authorization

Spring Security provides the this feature to authorize the user before accessing resources. It allows developers to define access policies against the resources.

Software Localization

This feature allows us to make application user interface in any language.

HTTP Authorization

Spring provides this feature for HTTP authorization of web request URLs using Apache Ant paths or regular expressions.

Features added in Spring Security 5.0

OAuth 2.0 Login

This feature provides the facility to the user to login into the application by using their existing account at GitHub or Google. This feature is implemented by using the Authorization Code Grant that is specified in the OAuth 2.0 Authorization Framework.

Reactive Support

From version Spring Security 5.0, it provides reactive programming and reactive web runtime support and even, we can integrate with Spring WebFlux.

Modernized Password Encoding

Spring Security 5.0 introduced new Password encoder **DelegatingPasswordEncoder** which is more modernize and solve all the problems of previous encoder **NoOpPasswordEncoder**.

Spring Project Modules

In Spring Security 3.0, the Security module is divided into separate jar files. The purpose was to divide jar files based on their functionalities, so, the developer can integrate according to their requirement.

It also helps to set required dependency into pom.xml file of maven project.

The following are the jar files that are included into Spring Security module.

- spring-security-core.jar
- spring-security-remoting.jar

- spring-security-web.jar
- spring-security-config.jar
- spring-security-ldap.jar
- spring-security-oauth2-core.jar
- spring-security-oauth2-client.jar
- spring-security-oauth2-jose.jar
- spring-security-acl.jar
- spring-security-cas.jar
- spring-security-openid.jar
- spring-security-test.jar

Core - spring-security-core.jar

This is core jar file and required for every application that wants to use Spring Security. This jar file includes core access-control and core authentication classes and interfaces. We can use it in standalone applications or remote clients applications.

It contains top level packages:

- org.springframework.security.core
- org.springframework.security.access
- org.springframework.security.authentication
- org.springframework.security.provisioning

Remoting - spring-security-remoting.jar

This jar is used to integrate security feature into the Spring remote application. We don't need it until or unless we are creating remote application. All the classes and interfaces are located into **org.springframework.security.remoting** package.

Web - spring-security-web.jar

This jar is useful for Spring Security web authentication and URL-based access control. It includes filters and web-security infrastructure.

All the classes and interfaces are located into the **org.springframework.security.web** package.

Config - spring-security-config.jar

This jar file is required for Spring Security configuration using XML and Java both. It includes Java configuration code and security namespace parsing code. All the classes and interfaces are stored in **org.springframework.security.config** package.

LDAP - spring-security-ldap.jar

This jar file is required only if we want to use LDAP (Lighweight Directory Access Protocol). It includes authentication and provisioning code. All the classes and interfaces are stored into **org.springframework.security.ldap** package.

OAuth 2.0 Core - spring-security-oauth2-core.jar

This jar is required to integrate Oauth 2.0 Authorization Framework and OpenID Connect Core 1.0 into the application. This jar file includes the core classes for OAuth 2.0 and classes are stored into the **org.springframework.security.oauth2.core** package.

OAuth 2.0 Client - spring-security-oauth2-client.jar

This jar file is required to get client support for OAuth 2.0 Authorization Framework and OpenID Connect Core 1.0. This module provides OAuth login and OpenID client support. All the classes and interfaces are available from **org.springframework.security.oauth2.client** package.

OAuth 2.0 JOSE - spring-security-oauth2-jose.jar

It provides Spring Security's support for the JOSE (Javascript Object Signing and Encryption) framework. The JOSE framework provides methods to establish secure connection between clients. It contains following collection of specifications:

- JWT (JSON Web Token)
- JWS (JSON Web Signature)
- JWE (JSON Web Encryption)
- JWK (JSON Web Key)

All the classes and interfaces are available into these two packages:

org.springframework.security.oauth2.jwt and **org.springframework.security.oauth2.jose.**

ACL - spring-security-acl.jar

This jar is used to apply security to domain object in the application. We can access classes and code from the **org.springframework.security.acls** package.

CAS - spring-security-cas.jar

It is required for Spring Security?s CAS client integration. We can use it to integrate Spring Security web authentication with CAS single sign-on server. The source code is located into **org.springframework.security.cas** package.

OpenID - spring-security-openid.jar

This jar is used for OpenID web authentication support. We can use it to authenticate users against an external OpenID server. It requires OpenID4Java and top level package is **org.springframework.security.openid**.

Test - spring-security-test.jar

This jar provides support for testing Spring Security application.

---

1. **Cross Site Request Forgery (CSRF)**

Before we discuss how Spring Security can protect applications from CSRF attacks, we will explain what a CSRF attack is. Let's take a look at a concrete example to get a better understanding.

Assume that your bank's website provides a form that allows transferring money from the currently logged in user to another bank account. For example, the HTTP request might look like:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876
```

Now pretend you authenticate to your bank's website and then, without logging out, visit an evil website. The evil website contains an HTML page with the following form:

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden"
        name="amount"
        value="100.00"/>
<input type="hidden"
        name="routingNumber"
        value="evilsRoutingNumber"/>
<input type="hidden"
```

```
        name="account"
        value="evilsAccountNumber"/>
<input type="submit"
        value="Win Money!"/>
</form>
```

You like to win money, so you click on the submit button. In the process, you have unintentionally transferred $100 to a malicious user. This happens because, while the evil website cannot see your cookies, the cookies associated with your bank are still sent along with the request.

Worst yet, this whole process could have been automated using JavaScript. This means you didn't even need to click on the button. So how do we protect ourselves from such attacks?

1.2 Synchronizer Token Pattern

The issue is that the HTTP request from the bank's website and the request from the evil website are exactly the same. This means there is no way to reject requests coming from the evil website and allow requests coming from the bank's website. To protect against CSRF attacks we need to ensure there is something in the request that the evil site is unable to provide.

One solution is to use the Synchronizer Token Pattern. This solution is to ensure that each request requires, in addition to our session cookie, a randomly generated token as an HTTP parameter. When a request is submitted, the server must look up the expected value for the parameter and compare it against the actual value in the request. If the values do not match, the request should fail.

We can relax the expectations to only require the token for each HTTP request that updates state. This can be safely done since the same origin policy ensures the evil site cannot read the response. Additionally, we do not want to include the random token in HTTP GET as this can cause the tokens to be leaked.

Let's take a look at how our example would change. Assume the randomly generated token is present in an HTTP parameter named _csrf. For example, the request to transfer money would look like this:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876&_csrf=<secure-random>
```

You will notice that we added the _csrf parameter with a random value. Now the evil website will not be able to guess the correct value for the _csrf parameter (which must be explicitly provided on the evil website) and the transfer will fail when the server compares the actual token to the expected token.

1.3 When to use CSRF protection

When should you use CSRF protection? Our recommendation is to use CSRF protection for any request that could be processed by a browser by normal users. If you are only creating a service that is used by non-browser clients, you will likely want to disable CSRF protection.

1.3.1 CSRF protection and JSON

A common question is "do I need to protect JSON requests made by javascript?" The short answer is, it depends. However, you must be very careful as there are CSRF exploits that can impact JSON requests. For example, a malicious user can create a CSRF with JSON using the following form:

```
<form action="https://bank.example.com/transfer" method="post"
enctype="text/plain">
<input
name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAcco
untNumber", "ignore_me":"' value='test"}' type='hidden'>
<input type="submit"
        value="Win Money!"/>
</form>
```

This will produce the following JSON structure

```
{ "amount": 100,
"routingNumber": "evilsRoutingNumber",
"account": "evilsAccountNumber",
"ignore_me": "=test"
}
```

If an application were not validating the Content-Type, then it would be exposed to this exploit. Depending on the setup, a Spring MVC application that validates the Content-Type could still be exploited by updating the URL suffix to end with ".json" as shown below:

```
<form action="https://bank.example.com/transfer.json" method="post"
enctype="text/plain">
<input
name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAcco
untNumber", "ignore_me":"' value='test"}' type='hidden'>
<input type="submit"
        value="Win Money!"/>
</form>
```

1.3.2 CSRF and Stateless Browser Applications

What if my application is stateless? That doesn't necessarily mean you are protected. In fact, if a user does not need to perform any actions in the web browser for a given request, they are likely still vulnerable to CSRF attacks.

For example, consider an application uses a custom cookie that contains all the state within it for authentication instead of the JSESSIONID. When the CSRF attack is made the custom cookie will be sent with the request in the same manner that the JSESSIONID cookie was sent in our previous example.

Users using basic authentication are also vulnerable to CSRF attacks since the browser will automatically include the username password in any requests in the same manner that the JSESSIONID cookie was sent in our previous example.

1.4 Using Spring Security CSRF Protection

So what are the steps necessary to use Spring Security's to protect our site against CSRF attacks? The steps to using Spring Security's CSRF protection are outlined below:

- **Use proper HTTP verbs**
- **Configure CSRF Protection**
- **Include the CSRF Token**

1.4.1 Use proper HTTP verbs

The first step to protecting against CSRF attacks is to ensure your website uses proper HTTP verbs. Specifically, before Spring Security's CSRF support can be of use, you need to be certain that your application is using PATCH, POST, PUT, and/or DELETE for anything that modifies state.

This is not a limitation of Spring Security's support, but instead a general requirement for proper CSRF prevention. The reason is that including private information in an HTTP GET can cause the information to be leaked. See RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's for general guidance on using POST instead of GET for sensitive information.

1.4.2 Configure CSRF Protection

The next step is to include Spring Security's CSRF protection within your application. Some frameworks handle invalid CSRF tokens by invalidating the user's session, but this causes its own problems. Instead by default Spring Security's CSRF protection will produce an HTTP 403 access denied. This can be customized by configuring the AccessDeniedHandler to process `InvalidCsrfTokenException` differently.

As of Spring Security 4.0, CSRF protection is enabled by default with XML configuration. If you would like to disable CSRF protection, the corresponding XML configuration can be seen below.

```
<http>
        <!-- ... -->
        <csrf disabled="true"/>
```

```
</http>
```

CSRF protection is enabled by default with Java Configuration. If you would like to disable CSRF, the corresponding Java configuration can be seen below. Refer to the Javadoc of csrf() for additional customizations in how CSRF protection is configured.

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http
			.csrf().disable();
	}
}
```

1.4.3 Include the CSRF Token

Form Submissions

The last step is to ensure that you include the CSRF token in all PATCH, POST, PUT, and DELETE methods. One way to approach this is to use the `_csrf` request attribute to obtain the current `CsrfToken`. An example of doing this with a JSP is shown below:

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
	method="post">
<input type="submit"
	value="Log out" />
<input type="hidden"
	name="${_csrf.parameterName}"
	value="${_csrf.token}"/>
</form>
```

An easier approach is to use the csrfInput tag from the Spring Security JSP tag library.

If you are using Spring MVC `<form:form>` tag or Thymeleaf 2.1+ and are using `@EnableWebSecurity`, the `CsrfToken` is automatically included for you (using the `CsrfRequestDataValueProcessor`).

Ajax and JSON Requests

If you are using JSON, then it is not possible to submit the CSRF token within an HTTP parameter. Instead you can submit the token within a HTTP header. A typical pattern

would be to include the CSRF token within your meta tags. An example with a JSP is shown below:

```
<html>
<head>
	<meta name="_csrf" content="${_csrf.token}"/>
	<!-- default header name is X-CSRF-TOKEN -->
	<meta name="_csrf_header" content="${_csrf.headerName}"/>
	<!-- ... -->
</head>
<!-- ... -->
```

Instead of manually creating the meta tags, you can use the simpler csrfMetaTags tag from the Spring Security JSP tag library.

You can then include the token within all your Ajax requests. If you were using jQuery, this could be done with the following:

```
$(function () {
var token = $("meta[name='_csrf']").attr("content");
var header = $("meta[name='_csrf_header']").attr("content");
$(document).ajaxSend(function(e, xhr, options) {
	xhr.setRequestHeader(header, token);
});
});
```

As an alternative to jQuery, we recommend using cujoJS's rest.js. The rest.js module provides advanced support for working with HTTP requests and responses in RESTful ways. A core capability is the ability to contextualize the HTTP client adding behavior as needed by chaining interceptors on to the client.

```
var client = rest.chain(csrf, {
token: $("meta[name='_csrf']").attr("content"),
name: $("meta[name='_csrf_header']").attr("content")
});
```

The configured client can be shared with any component of the application that needs to make a request to the CSRF protected resource. One significant difference between rest.js and jQuery is that only requests made with the configured client will contain the CSRF token, vs jQuery where all requests will include the token. The ability to scope which requests receive the token helps guard against leaking the CSRF token to a third party. Please refer to the rest.js reference documentation for more information on rest.js.

CookieCsrfTokenRepository

There can be cases where users will want to persist the `CsrfToken` in a cookie. By default the `CookieCsrfTokenRepository` will write to a cookie named `XSRF-TOKEN` and

read it from a header named `X-XSRF-TOKEN` or the HTTP parameter `_csrf`. These defaults come from AngularJS

You can configure `CookieCsrfTokenRepository` in XML using the following:

```
<http>
        <!-- ... -->
        <csrf token-repository-ref="tokenRepository"/>
</http>
<b:bean id="tokenRepository"
        class="org.springframework.security.web.csrf.CookieCsrfTokenRepository"
        p:cookieHttpOnly="false"/>
```

The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` to improve security.

You can configure `CookieCsrfTokenRepository` in Java Configuration using:

```
@EnableWebSecurity
public class WebSecurityConfig extends
                WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .csrf()

        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
        }
}
```

The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` (by using `new CookieCsrfTokenRepository()` instead) to improve security.

19.5 CSRF Caveats

There are a few caveats when implementing CSRF.

19.5.1 Timeouts

One issue is that the expected CSRF token is stored in the HttpSession, so as soon as the HttpSession expires your configured `AccessDeniedHandler` will receive a InvalidCsrfTokenException. If you are using the default `AccessDeniedHandler`, the browser will get an HTTP 403 and display a poor error message.

> One might ask why the expected `CsrfToken` isn't stored in a cookie by default. This is because there are known exploits in which headers (i.e. specify the cookies) can be set by another domain. This is the same reason Ruby on Rails no longer skips CSRF checks when the header X-Requested-With is present. See this webappsec.org thread for details on how to perform the exploit. Another disadvantage is that by removing the state (i.e. the timeout) you lose the ability to forcibly terminate the token if it is compromised.

A simple way to mitigate an active user experiencing a timeout is to have some JavaScript that lets the user know their session is about to expire. The user can click a button to continue and refresh the session.

Alternatively, specifying a custom `AccessDeniedHandler` allows you to process the `InvalidCsrfTokenException` any way you like. For an example of how to customize the `AccessDeniedHandler` refer to the provided links for both xml and Java configuration.

Finally, the application can be configured to use CookieCsrfTokenRepository which will not expire. As previously mentioned, this is not as secure as using a session, but in many cases can be good enough.

1.5.2 Logging In

In order to protect against forging log in requests the log in form should be protected against CSRF attacks too. Since the `CsrfToken` is stored in HttpSession, this means an HttpSession will be created as soon as `CsrfToken` token attribute is accessed. While this sounds bad in a RESTful / stateless architecture the reality is that state is necessary to implement practical security. Without state, we have nothing we can do if a token is compromised. Practically speaking, the CSRF token is quite small in size and should have a negligible impact on our architecture.

A common technique to protect the log in form is by using a JavaScript function to obtain a valid CSRF token before the form submission. By doing this, there is no need to think about session timeouts (discussed in the previous section) because the session is created right before the form submission (assuming that CookieCsrfTokenRepository isn't configured instead), so the user can stay on the login page and submit the username/password when he wants. In order to achieve this, you can take advantage of the `CsrfTokenArgumentResolver` provided by Spring Security and expose an endpoint like it's described on here.

1.5.3 Logging Out

Adding CSRF will update the LogoutFilter to only use HTTP POST. This ensures that log out requires a CSRF token and that a malicious user cannot forcibly log out your users.

One approach is to use a form for log out. If you really want a link, you can use JavaScript to have the link perform a POST (i.e. maybe on a hidden form). For browsers with JavaScript that is disabled, you can optionally have the link take the user to a log out confirmation page that will perform the POST.

If you really want to use HTTP GET with logout you can do so, but remember this is generally not recommended. For example, the following Java Configuration will perform logout with the URL /logout is requested with any HTTP method:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .logout()
                                .logoutRequestMatcher(new
AntPathRequestMatcher("/logout"));
        }
}
```

1.5.4 Multipart (file upload)

There are two options to using CSRF protection with multipart/form-data. Each option has its tradeoffs.

- **Placing MultipartFilter before Spring Security**
- **Include CSRF token in action**

Before you integrate Spring Security's CSRF protection with multipart file upload, ensure that you can upload without the CSRF protection first. More information about using multipart forms with Spring can be found within the 17.10 Spring's multipart (file upload) support section of the Spring reference and the MultipartFilter javadoc.

Placing MultipartFilter before Spring Security

The first option is to ensure that the `MultipartFilter` is specified before the Spring Security filter. Specifying the `MultipartFilter` before the Spring Security filter means that there is no authorization for invoking the `MultipartFilter` which means anyone can place temporary files on your server. However, only authorized users will be able to submit a File that is processed by your application. In general, this is the recommended approach because the temporary file upload should have a negligble impact on most servers.

To ensure `MultipartFilter` is specified before the Spring Security filter with java configuration, users can override beforeSpringSecurityFilterChain as shown below:

```
public class SecurityApplicationInitializer extends
AbstractSecurityWebApplicationInitializer {

	@Override
	protected void beforeSpringSecurityFilterChain(ServletContext
servletContext) {
		insertFilters(servletContext, new MultipartFilter());
	}
}
```

To ensure `MultipartFilter` is specified before the Spring Security filter with XML configuration, users can ensure the <filter-mapping> element of the `MultipartFilter` is placed before the springSecurityFilterChain within the web.xml as shown below:

```
<filter>
	<filter-name>MultipartFilter</filter-name>
	<filter-
class>org.springframework.web.multipart.support.MultipartFilter</filter-
class>
</filter>
<filter>
	<filter-name>springSecurityFilterChain</filter-name>
	<filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
	<filter-name>MultipartFilter</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
	<filter-name>springSecurityFilterChain</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
```

Include CSRF token in action

If allowing unauthorized users to upload temporariy files is not acceptable, an alternative is to place the `MultipartFilter` after the Spring Security filter and include the CSRF as a query parameter in the action attribute of the form. An example with a jsp is shown below

```
<form action="./upload?${_csrf.parameterName}=${_csrf.token}" method="post"
enctype="multipart/form-data">
```

The disadvantage to this approach is that query parameters can be leaked. More genearlly, it is considered best practice to place sensitive data within the body or headers to ensure it is not leaked. Additional information can be found in RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's.

1.5.5 HiddenHttpMethodFilter

The HiddenHttpMethodFilter should be placed before the Spring Security filter. In general this is true, but it could have additional implications when protecting against CSRF attacks.

Note that the HiddenHttpMethodFilter only overrides the HTTP method on a POST, so this is actually unlikely to cause any real problems. However, it is still best practice to ensure it is placed before Spring Security's filters.

1.6 Overriding Defaults

Spring Security's goal is to provide defaults that protect your users from exploits. This does not mean that you are forced to accept all of its defaults.

For example, you can provide a custom CsrfTokenRepository to override the way in which the `CsrfToken` is stored.

You can also specify a custom RequestMatcher to determine which requests are protected by CSRF (i.e. perhaps you don't care if log out is exploited). In short, if Spring Security's CSRF protection doesn't behave exactly as you want it, you are able to customize the behavior. Refer to the Section 43.1.18, "<csrf>" documentation for details on how to make these customizations with XML and the `CsrfConfigurer` javadoc for details on how to make these customizations when using Java configuration.
