Architecture

Architecture

This section discusses Spring Security's high-level architecture within Servlet based applications. We build on this high-level understanding within the Authentication, Authorization, and Protection Against Exploits sections of the reference.

A Review of Filters

Spring Security's Servlet support is based on Servlet Filters, so it is helpful to look at the role of Filters generally first. The following image shows the typical layering of the handlers for a single HTTP request.

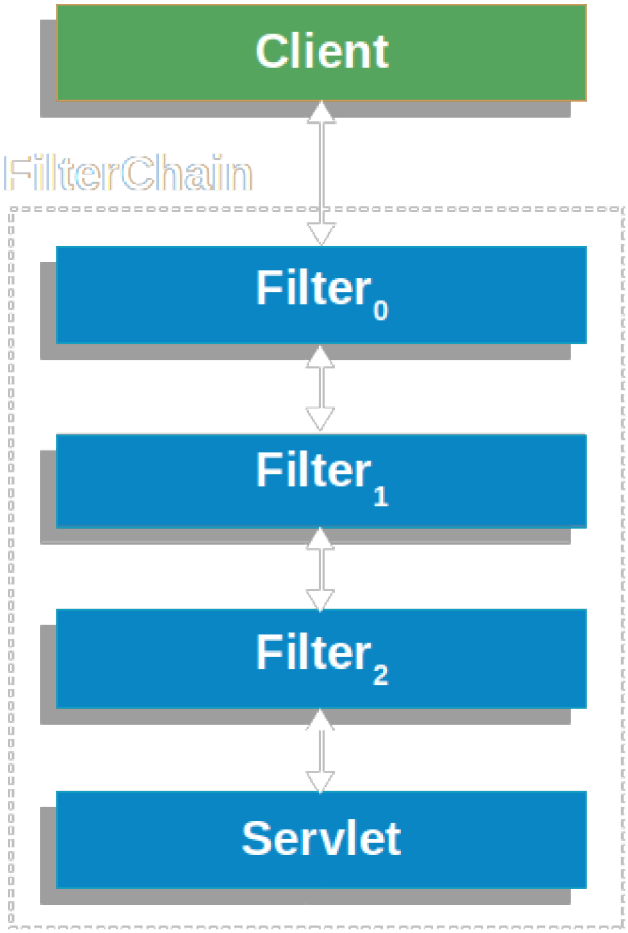


Figure 1. FilterChain

The client sends a request to the application, and the container creates a `FilterChain`, which contains the `Filter` instances and `Servlet` that should process the `HttpServletRequest`, based on the path of the request URI. In a Spring MVC application, the `Servlet` is an instance of `DispatcherServlet`. At most, one `Servlet` can handle a single `HttpServletRequest` and `HttpServletResponse`. However, more than one `Filter` can be used to:

- Prevent downstream `Filter` instances or the `Servlet` from being invoked. In this case, the `Filter` typically writes the `HttpServletResponse`.
- Modify the `HttpServletRequest` or `HttpServletResponse` used by the downstream `Filter` instances and the `Servlet`.

The power of the `Filter` comes from the `FilterChain` that is passed into it.

`FilterChain` Usage Example

Java | **Kotlin**

JAVA

```
@Override
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
    // do something before the rest of the application
    chain.doFilter(request, response); // invoke the rest of the application
    // do something after the rest of the application
}
```

Since a `Filter` impacts only downstream `Filter` instances and the `Servlet`, the order in which each `Filter` is invoked is extremely important.

DelegatingFilterProxy

Spring provides a `Filter` implementation named `DelegatingFilterProxy` that allows bridging between the Servlet container's lifecycle and Spring's `ApplicationContext`. The Servlet container allows registering `Filter` instances by using its own standards, but it is not aware of Spring-defined Beans. You can register `DelegatingFilterProxy` through the standard Servlet container mechanisms but delegate all the work to a Spring Bean that implements `Filter`.

Here is a picture of how `DelegatingFilterProxy` fits into the `Filter` instances and the `FilterChain`.

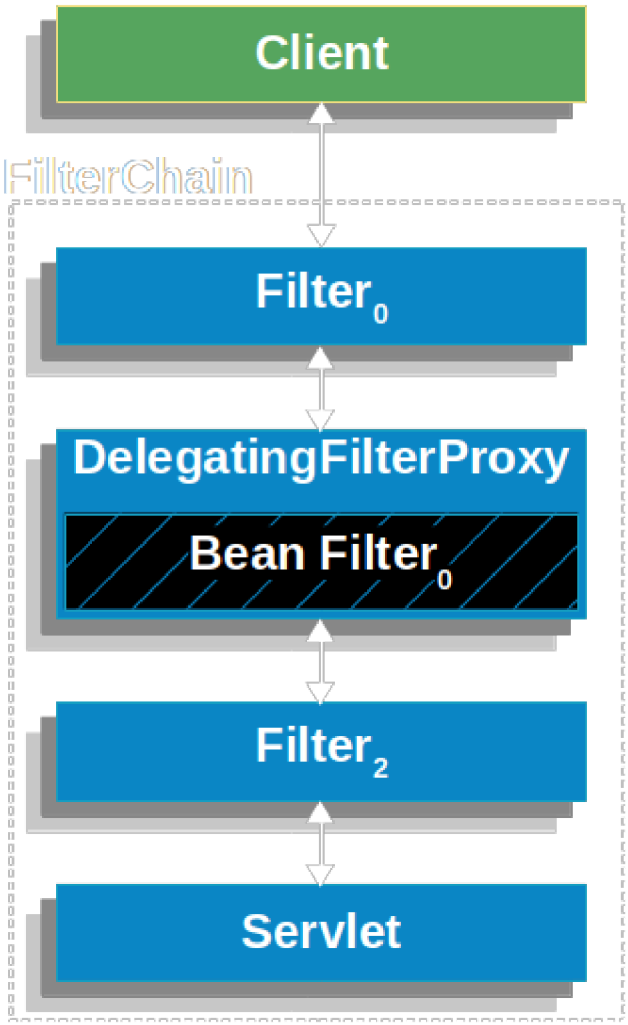


Figure 2. DelegatingFilterProxy

`DelegatingFilterProxy` looks up *Bean Filter*$_0$ from the `ApplicationContext` and then invokes *Bean Filter*$_0$. The following listing shows pseudo code of `DelegatingFilterProxy`:

`DelegatingFilterProxy` Pseudo Code

Java | Kotlin

```java
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) {
    Filter delegate = getFilterBean(someBeanName); 1
    delegate.doFilter(request, response); 2
}
```

1 Lazily get Filter that was registered as a Spring Bean. For the example in DelegatingFilterProxy `delegate` is an instance of *Bean Filter*$_0$.

2 Delegate work to the Spring Bean.

Another benefit of `DelegatingFilterProxy` is that it allows delaying looking up `Filter` bean instances. This is important because the container needs to register the `Filter` instances before the container can start up. However, Spring typically uses a `ContextLoaderListener` to load the Spring Beans, which is not done until after the `Filter` instances need to be registered.

FilterChainProxy

Spring Security's Servlet support is contained within `FilterChainProxy`. `FilterChainProxy` is a special `Filter` provided by Spring Security that allows delegating to many `Filter` instances through `SecurityFilterChain`. Since `FilterChainProxy` is a Bean, it is typically wrapped in a DelegatingFilterProxy.

The following image shows the role of `FilterChainProxy`.

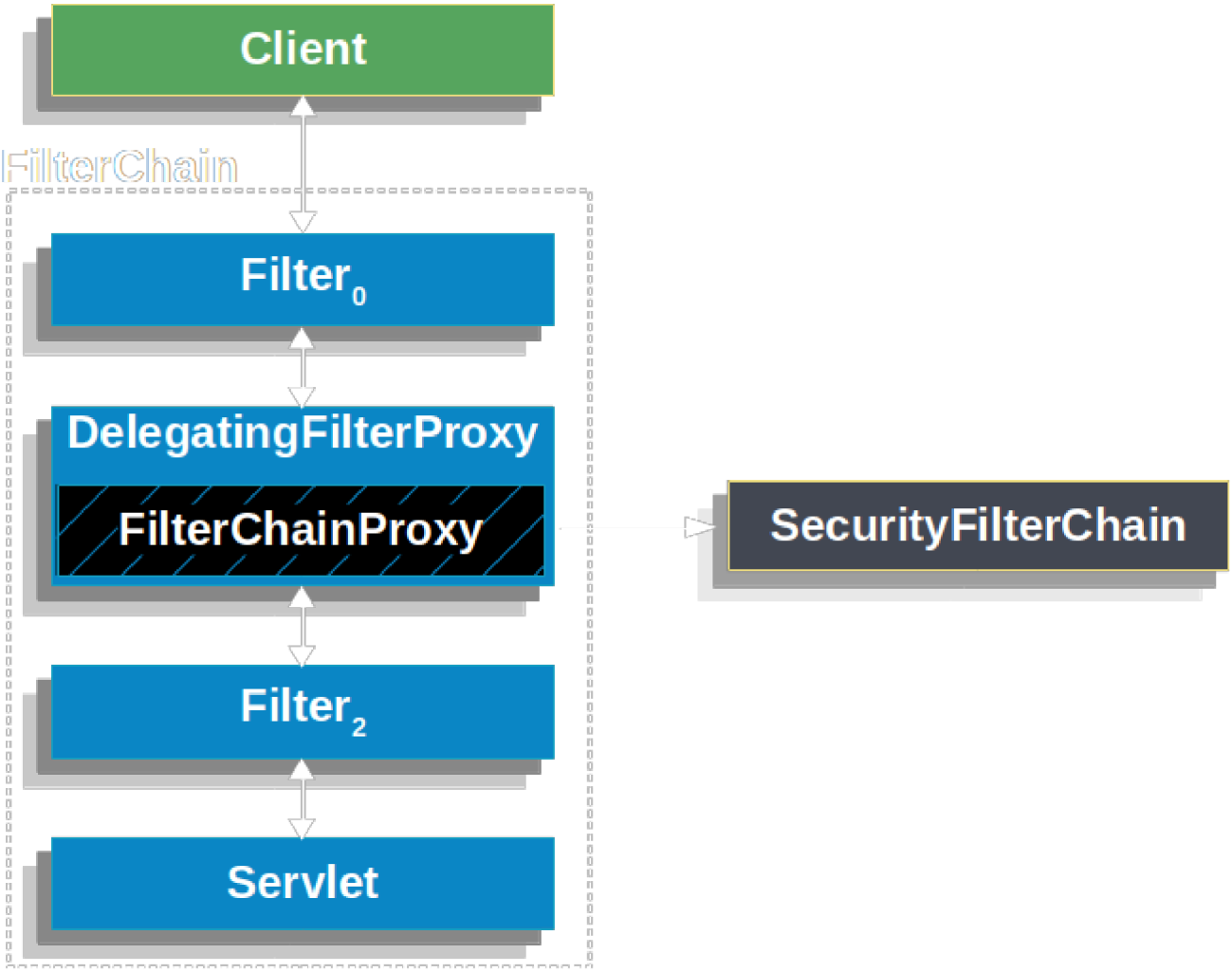


Figure 3. FilterChainProxy

SecurityFilterChain

`SecurityFilterChain` is used by FilterChainProxy to determine which Spring Security `Filter` instances should be invoked for the current request.

The following image shows the role of `SecurityFilterChain`.

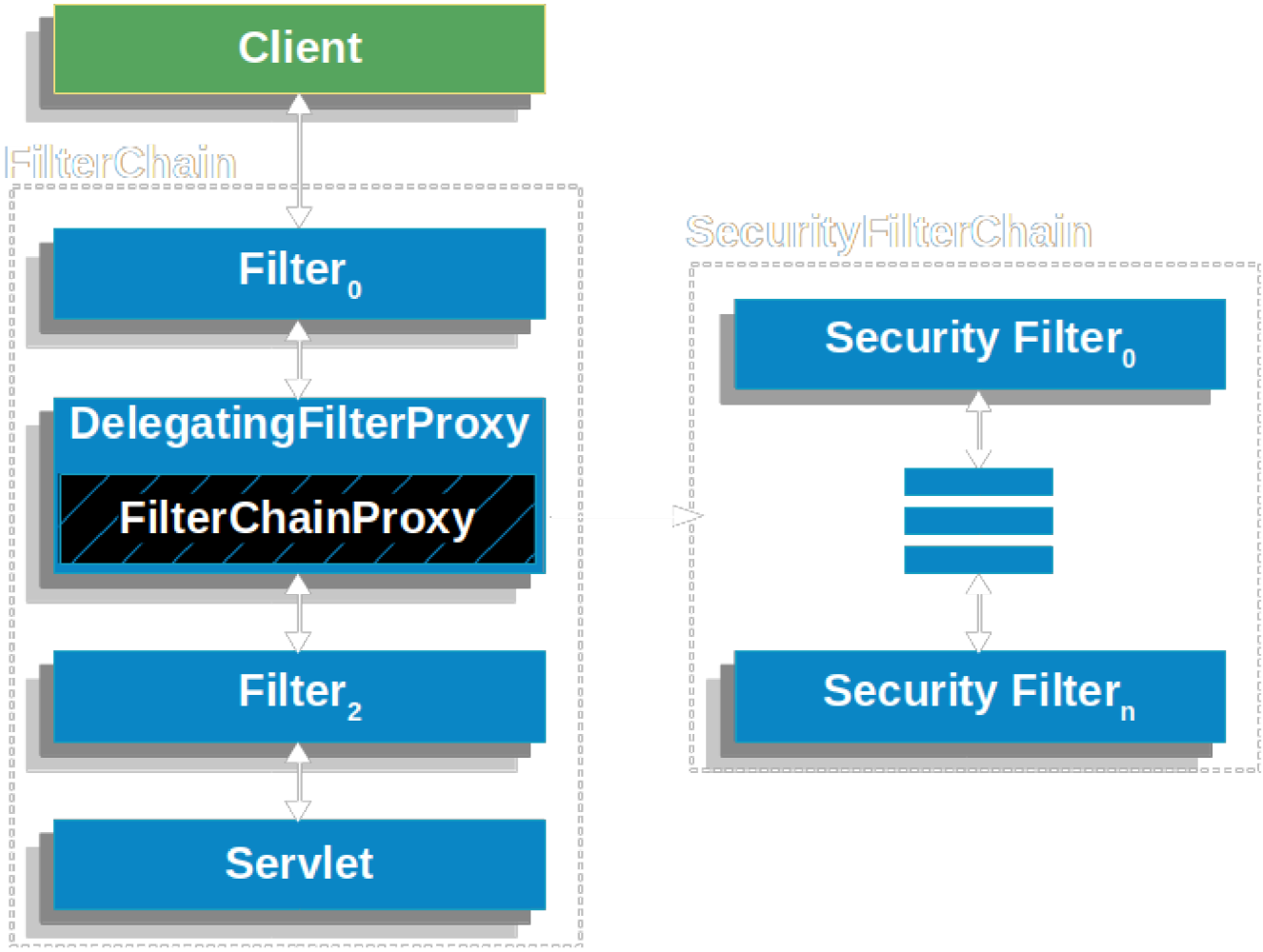


Figure 4. SecurityFilterChain

The Security Filters in `SecurityFilterChain` are typically Beans, but they are registered with `FilterChainProxy` instead of DelegatingFilterProxy. `FilterChainProxy` provides a number of advantages to registering directly with the Servlet container or DelegatingFilterProxy. First, it provides a starting point for all of Spring Security's Servlet support. For that reason, if you try to troubleshoot Spring Security's Servlet support, adding a debug point in `FilterChainProxy` is a great place to start.

Second, since `FilterChainProxy` is central to Spring Security usage, it can perform tasks that are not viewed as optional. For example, it clears out the `SecurityContext` to avoid memory leaks. It also applies Spring Security's `HttpFirewall` to protect applications against certain types of attacks.

In addition, it provides more flexibility in determining when a `SecurityFilterChain` should be invoked. In a Servlet container, `Filter` instances are invoked based upon the URL alone. However, `FilterChainProxy` can determine invocation based upon anything in the `HttpServletRequest` by using the `RequestMatcher` interface.

The following image shows multiple `SecurityFilterChain` instances:

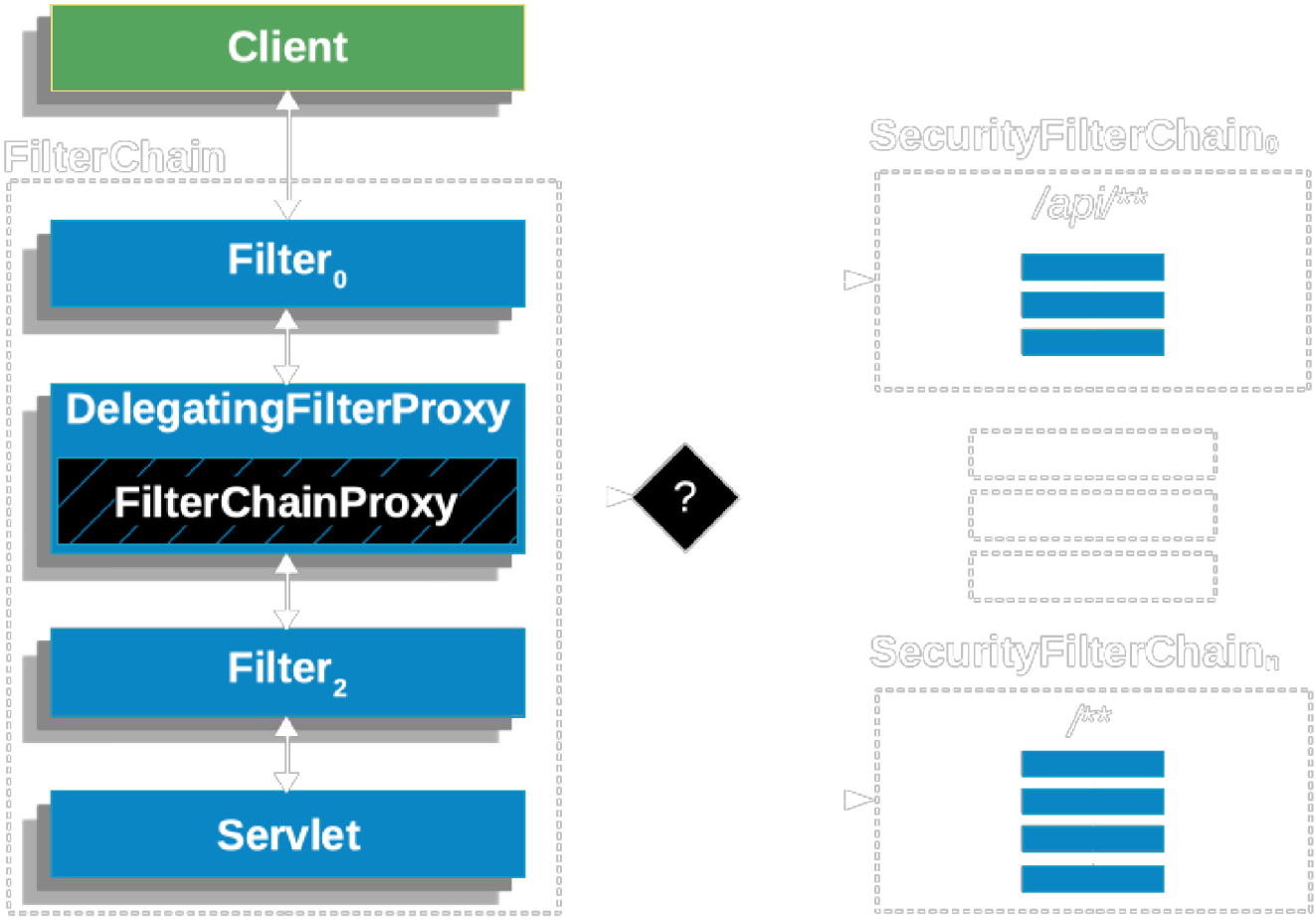


Figure 5. Multiple SecurityFilterChain

In the Multiple SecurityFilterChain figure, `FilterChainProxy` decides which `SecurityFilterChain` should be used. Only the first `SecurityFilterChain` that matches is invoked. If a URL of `/api/messages/` is requested, it first matches on the $SecurityFilterChain_0$ pattern of `/api/**`, so only $SecurityFilterChain_0$ is invoked, even though it also matches on $SecurityFilterChain_n$. If a URL of `/messages/` is requested, it does not match on the $SecurityFilterChain_0$ pattern of `/api/**`, so `FilterChainProxy` continues trying each `SecurityFilterChain`. Assuming that no other `SecurityFilterChain` instances match, $SecurityFilterChain_n$ is invoked.

Notice that $SecurityFilterChain_0$ has only three security `Filter` instances configured. However, $SecurityFilterChain_n$ has four security `Filter` instances configured. It is important to note that each `SecurityFilterChain` can be unique and can be configured in isolation. In fact, a `SecurityFilterChain` might have zero security `Filter` instances if the application wants Spring Security to ignore certain requests.

Security Filters

The Security Filters are inserted into the FilterChainProxy with the SecurityFilterChain API. Those filters can be used for a number of different purposes, like exploit protection, authentication, authorization, and more. The filters are executed in a specific order to guarantee that they are invoked at the right time, for example, the `Filter` that performs authentication should be invoked

before the `Filter` that performs authorization. It is typically not necessary to know the ordering of Spring Security's `Filter`s. However, there are times that it is beneficial to know the ordering, if you want to know them, you can check the `FilterOrderRegistration` code.

These security filters are most often declared using an `HttpSecurity` instance. To exemplify the above paragraph, let's consider the following security configuration:

Java Kotlin

JAVA

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(Customizer.withDefaults())
            .httpBasic(Customizer.withDefaults())
            .formLogin(Customizer.withDefaults())
            .authorizeHttpRequests((authorize) -> authorize
                .anyRequest().authenticated()
            );

        return http.build();
    }

}
```

The above configuration will result in the following `Filter` ordering:

Filter	Added by
CsrfFilter	`HttpSecurity#csrf`
BasicAuthenticationFilter	`HttpSecurity#httpBasic`
UsernamePasswordAuthenticationFilter	`HttpSecurity#formLogin`
AuthorizationFilter	`HttpSecurity#authorizeHttpRequests`

1. First, the `CsrfFilter` is invoked to protect against CSRF attacks.
2. Second, the authentication filters are invoked to authenticate the request.
3. Third, the `AuthorizationFilter` is invoked to authorize the request.

NOTE

There might be other `Filter` instances that are not listed above. If you want to see the list of filters invoked for a particular request, you can print them.

Printing the Security Filters

Often times, it is useful to see the list of security `Filter`s that are invoked for a particular request. For example, you want to make sure that the filter you have added is in the list of the security filters.

The list of filters is printed at DEBUG level on the application startup, so you can see something like the following on the console output for example:

```
2023-06-14T08:55:22.321-03:00  DEBUG 76975 --- [           main]
o.s.s.web.DefaultSecurityFilterChain     : Will secure any request with [
DisableEncodeUrlFilter, WebAsyncManagerIntegrationFilter,
SecurityContextHolderFilter, HeaderWriterFilter, CsrfFilter, LogoutFilter,
UsernamePasswordAuthenticationFilter, DefaultLoginPageGeneratingFilter,
DefaultLogoutPageGeneratingFilter, BasicAuthenticationFilter,
RequestCacheAwareFilter, SecurityContextHolderAwareRequestFilter,
AnonymousAuthenticationFilter, ExceptionTranslationFilter, AuthorizationFilter]
```
TEXT

And that will give a pretty good idea of the security filters that are configured for each filter chain.

But that is not all, you can also configure your application to print the invocation of each individual filter for each request. That is helpful to see if the filter you have added is invoked for a particular request or to check where an exception is coming from. To do that, you can configure your application to log the security events.

Adding Filters to the Filter Chain

Most of the time, the default Security Filters are enough to provide security to your application. However, there might be times that you want to add a custom `Filter` to the SecurityFilterChain.

`HttpSecurity` comes with three methods for adding filters:

- `#addFilterBefore(Filter, Class<?>)` adds your filter before another filter
- `#addFilterAfter(Filter, Class<?>)` adds your filter after another filter
- `#addFilterAt(Filter, Class<?>)` replaces another filter with your filter

Adding a Custom Filter

If you are creating a filter of your own, you will need to determine its location in the filter chain. Please take a look at the following key events that occur in the filter chain:

1. `SecurityContext` is loaded from the session
2. Request is protected from common exploits; secure headers, CORS, CSRF
3. Request is authenticated

4. Request is authorized

Consider which events you need to have happened in order to locate your filter. The following is a rule of thumb:

If your filter is a(n)	Then place it after	As these events have already occurred
exploit protection filter	SecurityContextHolderFilter	1
authentication filter	LogoutFilter	1, 2
authorization filter	AnonymousAuthenticationFilter	1, 2, 3

TIP

Most commonly, applications add a custom authentication. This means they should be placed after `LogoutFilter`.

For example, let's say that you want to add a `Filter` that gets a tenant id header and check if the current user has access to that tenant.

First, let's create the `Filter`:

JAVA

```
public class TenantFilter implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;

        String tenantId = request.getHeader("X-Tenant-Id"); 1
        boolean hasAccess = isUserAllowed(tenantId); 2
        if (hasAccess) {
            filterChain.doFilter(request, response); 3
            return;
        }
        throw new AccessDeniedException("Access denied"); 4
    }

}
```

The sample code above does the following:

1. Get the tenant id from the request header.
2. Check if the current user has access to the tenant id.
3. If the user has access, then invoke the rest of the filters in the chain.
4. If the user does not have access, then throw an `AccessDeniedException`.

TIP

Instead of implementing `Filter`, you can extend from OncePerRequestFilter which is a base class for filters that are only invoked once per request and provides a `doFilterInternal` method with the `HttpServletRequest` and `HttpServletResponse` parameters.

Now, you need to add the filter to the SecurityFilterChain. The previous description already gives us a clue on where to add the filter, since we need to know the current user, we need to add it after the authentication filters.

Based on the rule of thumb, add it after `AnonymousAuthenticationFilter`, the last authentication filter in the chain, like so:

Java Kotlin

JAVA

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        // ...
        .addFilterAfter(new TenantFilter(),
AnonymousAuthenticationFilter.class);  1
    return http.build();
}
```

1 Use `HttpSecurity#addFilterAfter` to add the `TenantFilter` after the `AnonymousAuthenticationFilter`.

By adding the filter after the `AnonymousAuthenticationFilter` we are making sure that the `TenantFilter` is invoked after the authentication filters.

And that's it, now the `TenantFilter` will be invoked in the filter chain and will check if the current user has access to the tenant id.

Declaring Your Filter as a Bean

When you declare a `Filter` as a Spring bean, either by annotating it with `@Component` or by declaring it as a bean in your configuration, Spring Boot automatically registers it with the embedded container. That may cause the filter to be invoked twice, once by the container and once by Spring Security and in a different order.

Because of that, filters are often not Spring beans.

However, if your filter needs to be a Spring bean (to take advantage of dependency injection, for example) you can tell Spring Boot to not register it with the container by declaring a `FilterRegistrationBean` bean and setting its `enabled` property to `false`:

JAVA

```
@Bean
public FilterRegistrationBean<TenantFilter>
tenantFilterRegistration(TenantFilter filter) {
```

```
    FilterRegistrationBean<TenantFilter> registration = new
FilterRegistrationBean<>(filter);
    registration.setEnabled(false);
    return registration;
}
```

This makes so that `HttpSecurity` is the only one adding it.

Customizing a Spring Security Filter

Generally, you can use a filter's DSL method to configure Spring Security's filters. For example, the simplest way to add `BasicAuthenticationFilter` is by asking the DSL to do it:

Java | Kotlin

JAVA

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .httpBasic(Customizer.withDefaults())
        // ...

    return http.build();
}
```

However, in the event that you want to construct a Spring Security filter yourself, you specify it in the DSL using `addFilterAt` like so:

Java | Kotlin

JAVA

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    BasicAuthenticationFilter basic = new BasicAuthenticationFilter();
    // ... configure

    http
        // ...
        .addFilterAt(basic, BasicAuthenticationFilter.class);

    return http.build();
}
```

Note that if that filter has already been added, then Spring Security will throw an exception. For example, calling `HttpSecurity#httpBasic` adds a `BasicAuthenticationFilter` for you. So, the following arrangement fails since there are two calls that are both trying to add `BasicAuthenticationFilter`:

Java | Kotlin

```java
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    BasicAuthenticationFilter basic = new BasicAuthenticationFilter();
    // ... configure

    http
        .httpBasic(Customizer.withDefaults())
        // ... on no! BasicAuthenticationFilter is added twice!
        .addFilterAt(basic, BasicAuthenticationFilter.class);

    return http.build();
}
```

In this case, remove the call to `httpBasic` since you are constructing `BasicAuthenticationFilter` yourself.

TIP

In the event that you are unable to reconfigure `HttpSecurity` to not add a certain filter, you can typically disable the Spring Security filter by calling its DSL's `disable` method like so:

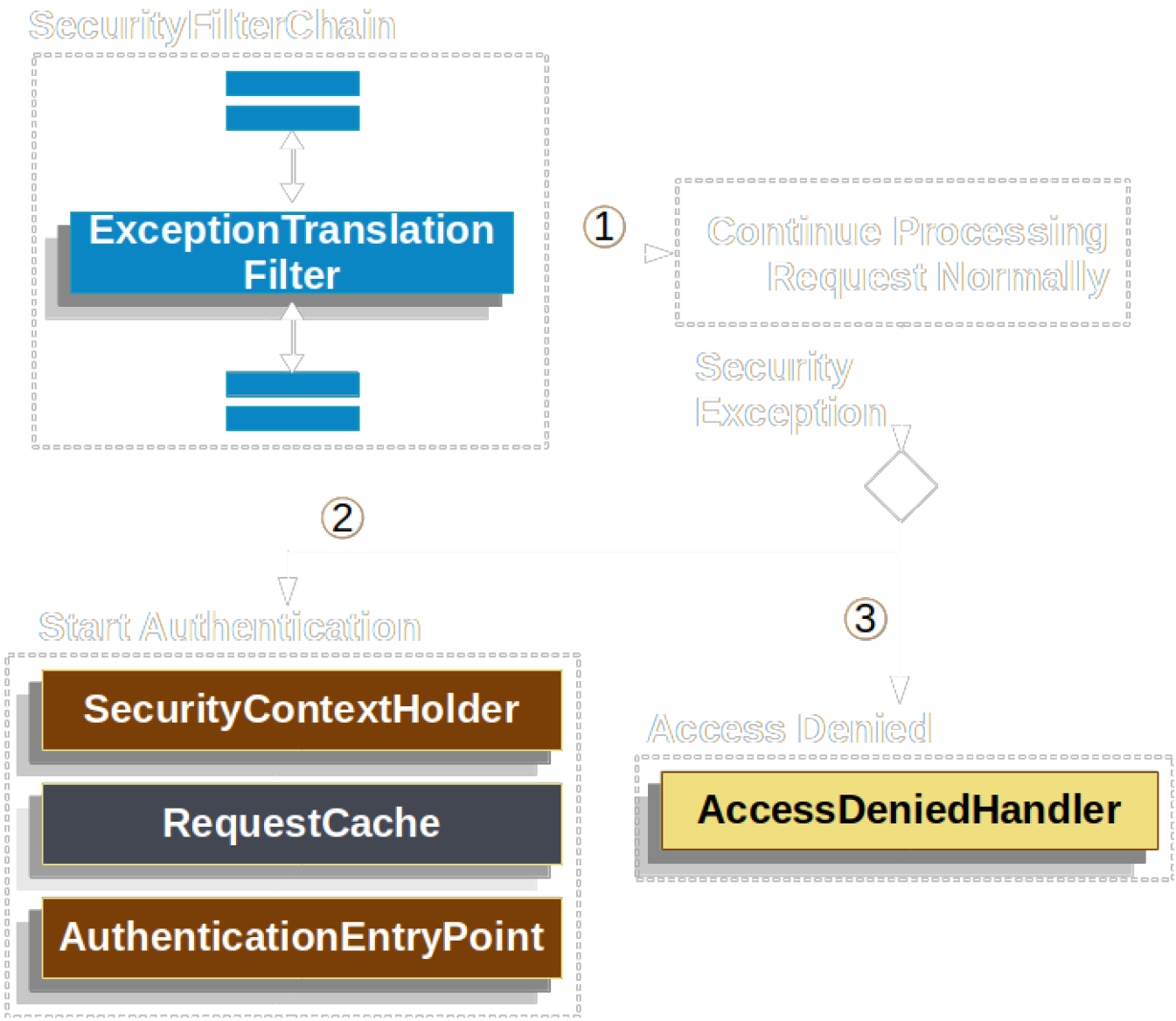
```
.httpBasic((basic) -> basic.disable())
```

Handling Security Exceptions

The `ExceptionTranslationFilter` allows translation of `AccessDeniedException` and `AuthenticationException` into HTTP responses.

`ExceptionTranslationFilter` is inserted into the FilterChainProxy as one of the Security Filters.

The following image shows the relationship of `ExceptionTranslationFilter` to other components:



- ❶ First, the `ExceptionTranslationFilter` invokes `FilterChain.doFilter(request, response)` to invoke the rest of the application.
- ❷ If the user is not authenticated or it is an `AuthenticationException`, then *Start Authentication*.
  - The SecurityContextHolder is cleared out.
  - The `HttpServletRequest` is saved so that it can be used to replay the original request once authentication is successful.
  - The `AuthenticationEntryPoint` is used to request credentials from the client. For example, it might redirect to a log in page or send a `WWW-Authenticate` header.
- ❸ Otherwise, if it is an `AccessDeniedException`, then *Access Denied*. The `AccessDeniedHandler` is invoked to handle access denied.

NOTE

If the application does not throw an `AccessDeniedException` or an `AuthenticationException`, then `ExceptionTranslationFilter` does not do anything.

The pseudocode for `ExceptionTranslationFilter` looks something like this:

ExceptionTranslationFilter pseudocode

```java
try {
    filterChain.doFilter(request, response); 1
} catch (AccessDeniedException | AuthenticationException ex) {
    if (!authenticated || ex instanceof AuthenticationException) {
        startAuthentication(); 2
    } else {
        accessDenied(); 3
    }
}
```

1. As described in A Review of Filters, invoking `FilterChain.doFilter(request, response)` is the equivalent of invoking the rest of the application. This means that if another part of the application, (`AuthorizationFilter` or method security) throws an `AuthenticationException` or `AccessDeniedException` it is caught and handled here.
2. If the user is not authenticated or it is an `AuthenticationException`, *Start Authentication*.
3. Otherwise, *Access Denied*

Saving Requests Between Authentication

As illustrated in Handling Security Exceptions, when a request has no authentication and is for a resource that requires authentication, there is a need to save the request for the authenticated resource to re-request after authentication is successful. In Spring Security this is done by saving the `HttpServletRequest` using a `RequestCache` implementation.

RequestCache

The `HttpServletRequest` is saved in the `RequestCache`. When the user successfully authenticates, the `RequestCache` is used to replay the original request. The `RequestCacheAwareFilter` uses the `RequestCache` to get the saved `HttpServletRequest` after the user authenticates, while the `ExceptionTranslationFilter` uses the `RequestCache` to save the `HttpServletRequest` after it detects `AuthenticationException`, before redirecting the user to the login endpoint.

By default, an `HttpSessionRequestCache` is used. The code below demonstrates how to customize the `RequestCache` implementation that is used to check the `HttpSession` for a saved request if the parameter named `continue` is present.

`RequestCache` Only Checks for Saved Requests if `continue` Parameter Present

Java | Kotlin | XML

```java
@Bean
DefaultSecurityFilterChain springSecurity(HttpSecurity http) throws Exception {
```

```
    HttpSessionRequestCache requestCache = new HttpSessionRequestCache();
    requestCache.setMatchingRequestParameterName("continue");
    http
        // ...
        .requestCache((cache) -> cache
            .requestCache(requestCache)
        );
    return http.build();
}
```

Prevent the Request From Being Saved

There are a number of reasons you may want to not store the user's unauthenticated request in the session. You may want to offload that storage onto the user's browser or store it in a database. Or you may want to shut off this feature since you always want to redirect the user to the home page instead of the page they tried to visit before login.

To do that, you can use the NullRequestCache implementation.

Prevent the Request From Being Saved

Java | Kotlin | XML

JAVA

```
@Bean
SecurityFilterChain springSecurity(HttpSecurity http) throws Exception {
    RequestCache nullRequestCache = new NullRequestCache();
    http
        // ...
        .requestCache((cache) -> cache
            .requestCache(nullRequestCache)
        );
    return http.build();
}
```

RequestCacheAwareFilter

The `RequestCacheAwareFilter` uses the `RequestCache` to replay the original request.

Logging

Spring Security provides comprehensive logging of all security related events at the DEBUG and TRACE level. This can be very useful when debugging your application because for security measures Spring Security does not add any detail of why a request has been rejected to the response body. If you come across a 401 or 403 error, it is very likely that you will find a log message that will help you understand what is going on.

Let's consider an example where a user tries to make a `POST` request to a resource that has CSRF protection enabled without the CSRF token. With no logs, the user will see a 403 error with no explanation of why the request was rejected. However, if you enable logging for Spring Security, you will see a log message like this:

```text
2023-06-14T09:44:25.797-03:00 DEBUG 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Securing POST /hello
2023-06-14T09:44:25.797-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking DisableEncodeUrlFilter
(1/15)
2023-06-14T09:44:25.798-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking
WebAsyncManagerIntegrationFilter (2/15)
2023-06-14T09:44:25.800-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking SecurityContextHolderFilter
(3/15)
2023-06-14T09:44:25.801-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking HeaderWriterFilter (4/15)
2023-06-14T09:44:25.802-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking CsrfFilter (5/15)
2023-06-14T09:44:25.814-03:00 DEBUG 76975 --- [nio-8080-exec-1]
o.s.security.web.csrf.CsrfFilter         : Invalid CSRF token found for
http://localhost:8080/hello
2023-06-14T09:44:25.814-03:00 DEBUG 76975 --- [nio-8080-exec-1]
o.s.s.w.access.AccessDeniedHandlerImpl   : Responding with 403 status code
2023-06-14T09:44:25.814-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.s.w.header.writers.HstsHeaderWriter  : Not injecting HSTS header since it
did not match request to [Is Secure]
```

It becomes clear that the CSRF token is missing and that is why the request is being denied.

To configure your application to log all the security events, you can add the following to your application:

application.properties in Spring Boot

```properties
logging.level.org.springframework.security=TRACE
```

logback.xml

```xml
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <!-- ... -->
    </appender>
    <!-- ... -->
    <logger name="org.springframework.security" level="trace"
additivity="false">
        <appender-ref ref="Console" />
```

```
    </logger>
</configuration>
```