Spring Security

Simeon Monov, Hristo Karaperev

Overview

- Spring Security is a framework that provides authentication, authorization, and protection against common attacks
- De-facto standard for securing Spring-based applications
- Provides first class support for securing both imperative and reactive applications

Features

- Spring Security provides comprehensive support for authentication, authorization, and protection against common exploits
- Authentication is how we verify the identity of who is trying to access a particular resource. A common way to authenticate users is by requiring the user to enter a username and password. Once authentication is performed we know the identity and can perform authorization

Authentication - Password Storage

- Spring Security’s PasswordEncoder interface is used to perform a one-way transformation of a password to let the password be stored securely
- Given PasswordEncoder is a one-way transformation, it is not useful when the password transformation needs to be two-way
- Typically, PasswordEncoder is used for storing a password that needs to be compared to a user-provided password at the time of authentication

Authorization

- Spring Security provides comprehensive support for authorization
- Authorization is determining who is allowed to access a particular resource
- Spring Security provides defense in depth by allowing for request based authorization and method based authorization
- Spring Security provides authorization based upon the request for both Servlet and WebFlux environments

Spring Security - Servlet Applications

- Spring Security integrates with the Servlet Container by using a standard Servlet Filter
- This means it works with any application that runs in a Servlet Container
- You do not need to use Spring in your Servlet-based application to take advantage of Spring Security
- To include Spring Security in you project, you have to include it in the classpath

Spring Security And Spring Boot

- There are things we can expect by default from Spring Security:
  - Requires an authenticated user for any endpoint (including Boot’s /error endpoint)
  - Registers a default user with a generated password at startup
  - Protects password storage with BCrypt as well as others
  - Provides form-based login and logout flows
  - Authenticates form-based login as well as HTTP Basic
  - Mitigates CSRF attacks
  - Writes Strict-Transport-Security to ensure HTTPS
  - Writes Cache Control headers that protect authenticated resources
  - Publishes authentication success and failure events
  - Few more…

Architecture (1)

- Spring Security's Servlet support is based on Servlet Filters
- The client sends a request to the application, and the container creates a FilterChain
- Contains the Filter instances and Servlet that should process the HttpServletRequest, based on the path of the request URI

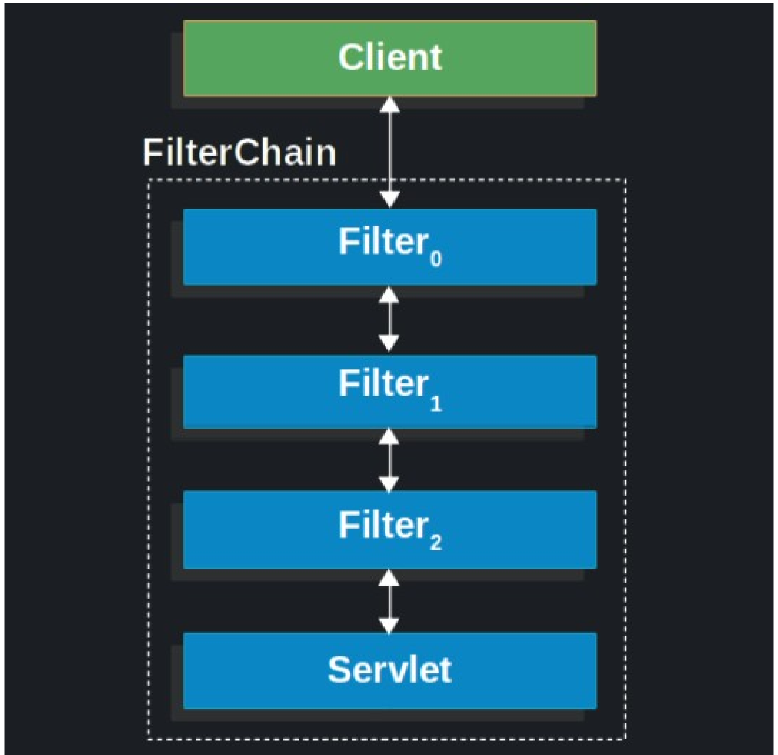


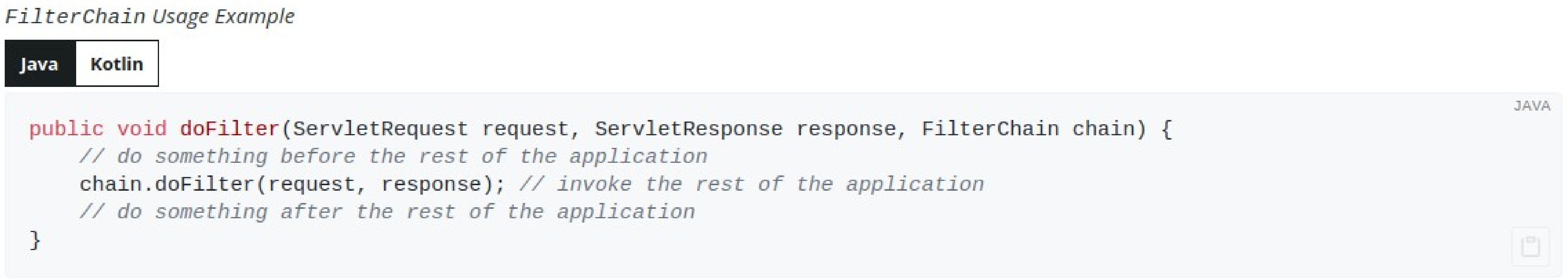
Figure 1. FilterChain

Architecture (2)

- 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

Architecture (3)

- The power of the Filter comes from the FilterChain that is passed into it
- Since a Filter impacts only downstream Filter instances and the Servlet, the order in which each Filter is invoked is extremely important



FilterChain Usage Example

Java | Kotlin

JAVA

```java
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
    // 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
}
```

Architecture – DelegatingFilterProxy (1)

- Spring provides a Filter implementation named DelegatingFilterProxy
- 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 container mechanisms but delegate all the work to a Spring Bean that implements Filter

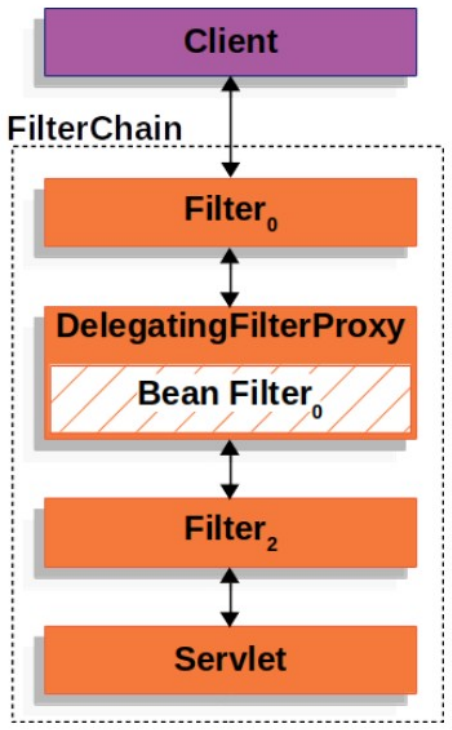
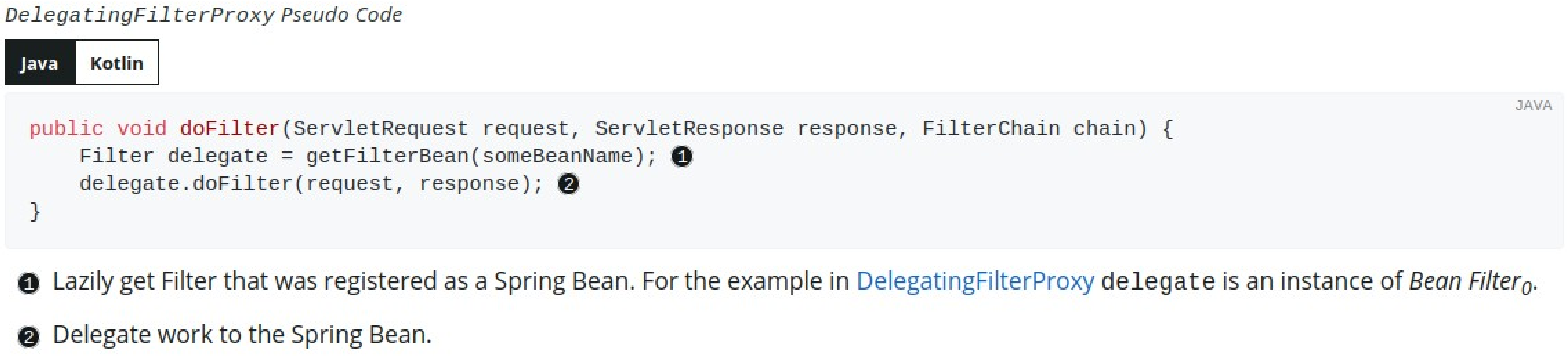


Figure 2. DelegatingFilterProxy

Architecture – DelegatingFilterProxy (2)

- 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



DelegatingFilterProxy Pseudo Code

Java | Kotlin

JAVA

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

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

❷ Delegate work to the Spring Bean.

Architecture – FilterChainProxy (1)

- 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

Architecture – FilterChainProxy (2)

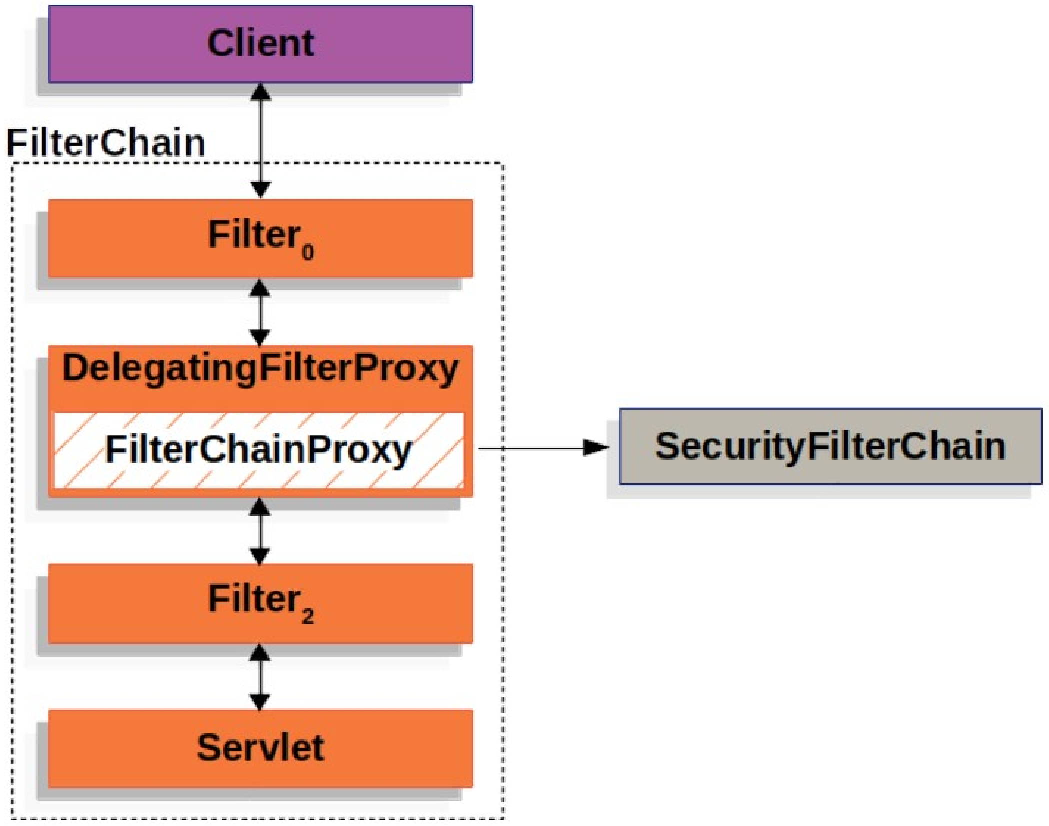


Figure 3. FilterChainProxy

Architecture – SecurityFilterChain (1)

- SecurityFilterChain is used by FilterChainProxy to determine which Spring Security Filter instances should be invoked for the current request
- The Security Filters in SecurityFilterChain are typically Beans, but they are registered with FilterChainProxy instead of DelegatingFilterProxy

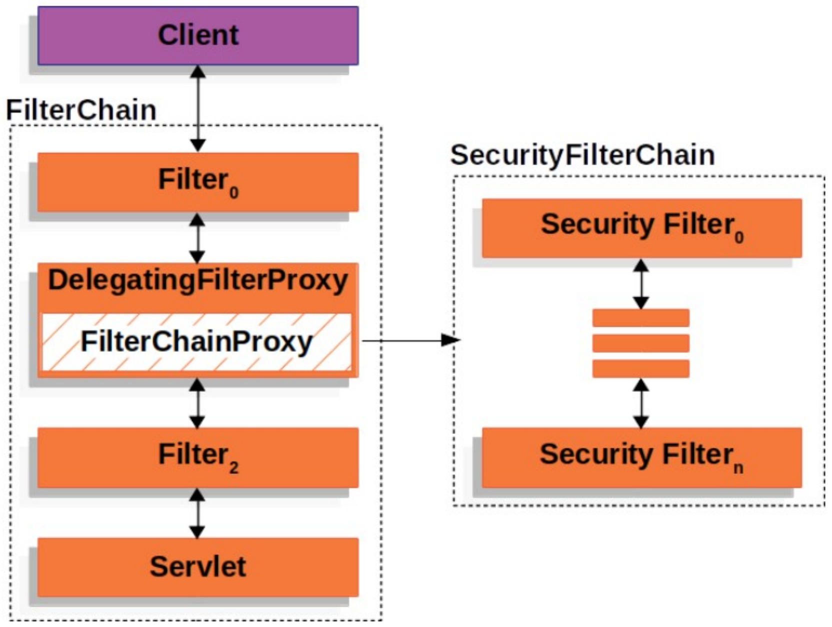


Figure 4. SecurityFilterChain

Architecture – SecurityFilterChain (2)

- FilterChainProxy provides a number of advantages to registering directly with the Servlet container or DelegatingFilterProxy
    - 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

Architecture – SecurityFilterChain (3)

- 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

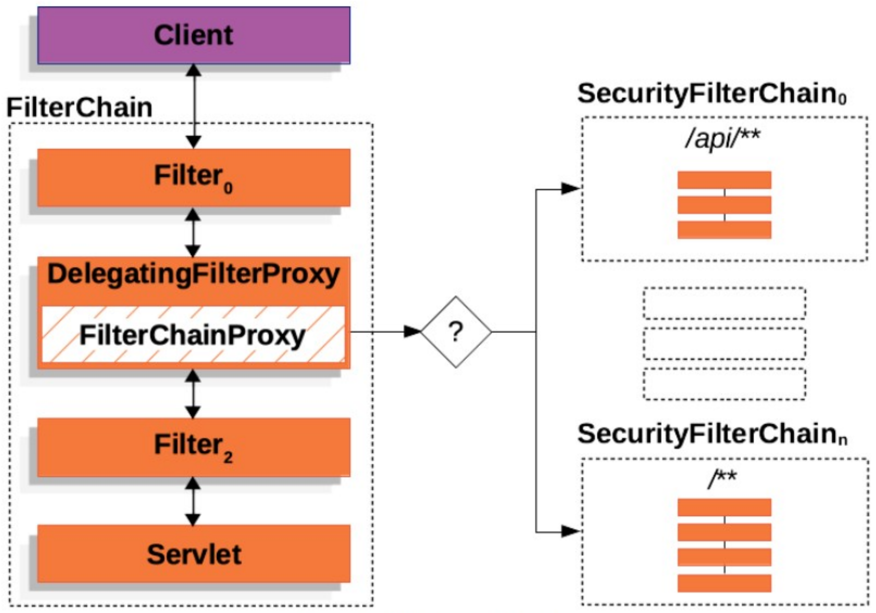


Figure 5. Multiple SecurityFilterChain

Architecture – Security Filters (1)

- The Security Filters are inserted into the FilterChainProxy with the SecurityFilterChain API
- Those filters can be used for a number of different purposes, like authentication, authorization, exploit protection, 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

Architecture – Security Filters (1)

- The Security Filters are inserted into the FilterChainProxy with the SecurityFilterChain API
- Those filters can be used for a number of different purposes, like authentication, authorization, exploit protection, 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

Java Kotlin

JAVA

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

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

}
```

Architecture – Security Filters (1)

- The Security Filters are inserted into the FilterChainProxy with the SecurityFilterChain API
- Those filters can be used for a number of different purposes, like authentication, authorization, exploit protection, 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

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

Adding a Custom Filter to the Filter Chain (1)

- The default security filters are enough to provide security to your application
- There might be times that you want to add a custom Filter to the security filter chain
- 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
- Since we need to know the current user, we need to add it after the authentication filters

JAVA

```java
public class TenantFilter implements Filter {

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

        String tenantId = request.getHeader("X-Tenant-Id"); ❶
        boolean hasAccess = isUserAllowed(tenantId); ❷
        if (hasAccess) {
            filterChain.doFilter(request, response); ❸
            return;
        }
        throw new AccessDeniedException("Access denied"); ❹
    }

}
```

Java

JAVA

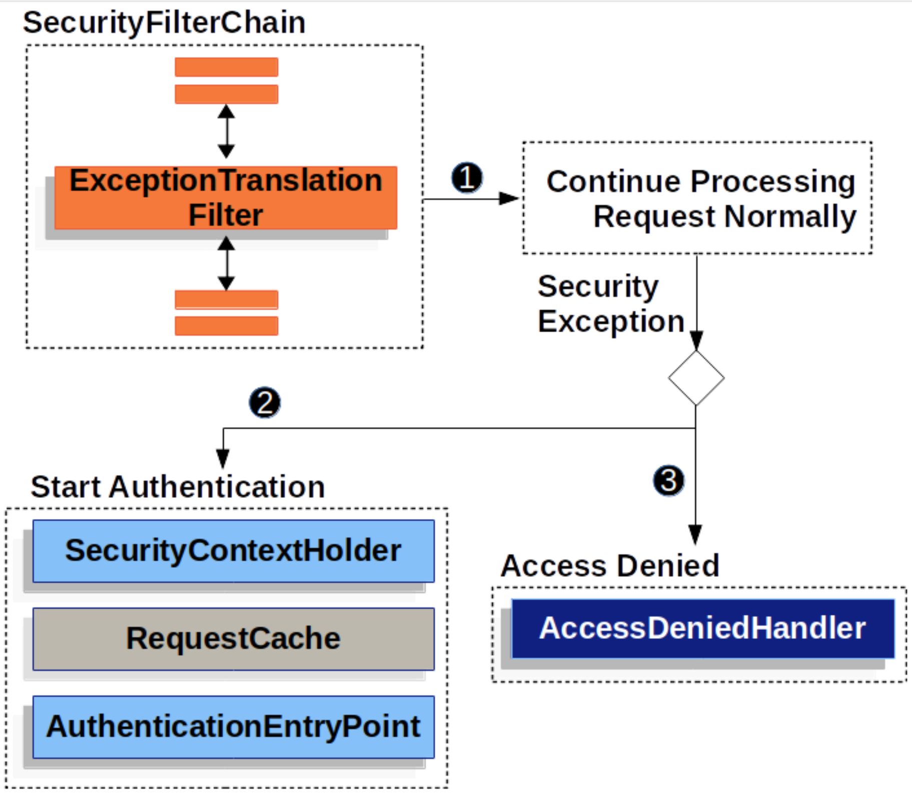
```java
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        // ...
        .addFilterBefore(new TenantFilter(), AuthorizationFilter.class); ❶
    return http.build();
}
```

Adding a Custom Filter to the Filter Chain (2)

- By adding the filter before the AuthorizationFilter we are making sure that the TenantFilter is invoked after the authentication filters
- You can also use HttpSecurity#addFilterAfter to add the filter after a particular filter or HttpSecurity#addFilterAt to add the filter at a particular filter position in the filter chain

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



- 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.

Saving Requests Between Authentication (1)

- 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

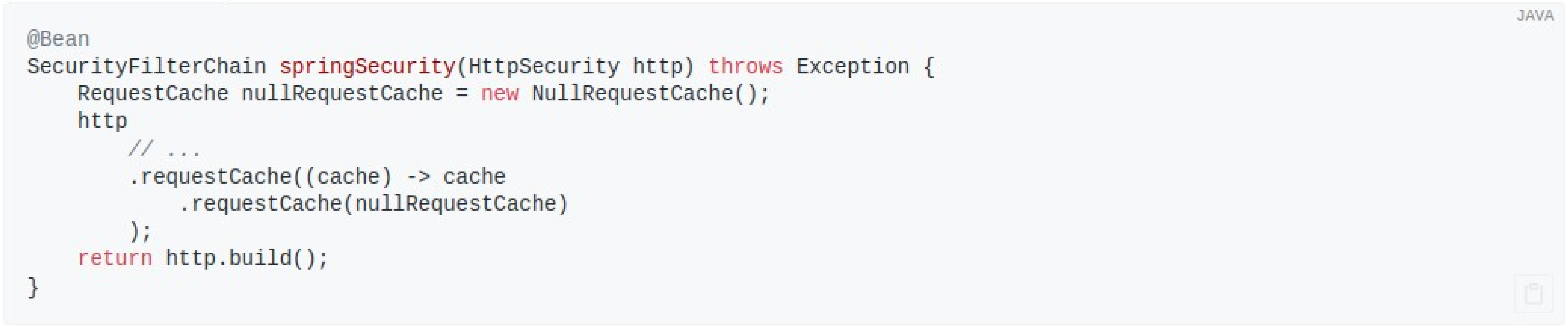
- The HttpServletRequest is saved in the RequestCache. When the user successfully authenticates, the RequestCache is used to replay the original request
- The RequestCacheAwareFilter is what uses the RequestCache to save the HttpServletRequest
- By default, an HttpSessionRequestCache is used

JAVA

```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



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

Logging

- Spring Security provides comprehensive logging of all security related events at the DEBUG and TRACE level
- 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

Servlet Authentication Architecture (1)

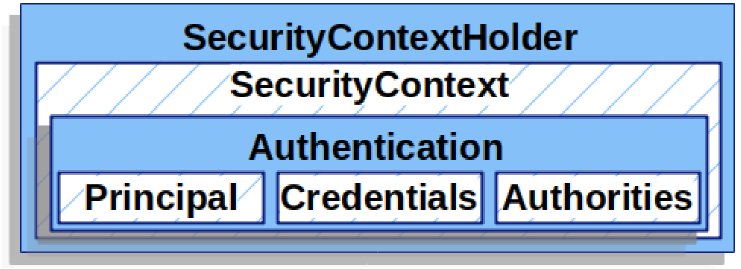
- SecurityContextHolder - The SecurityContextHolder is where Spring Security stores the details of who is authenticated
- SecurityContext - is obtained from the SecurityContextHolder and contains the Authentication of the currently authenticated user
- Authentication - Can be the input to AuthenticationManager to provide the credentials a user has provided to authenticate or the current user from the SecurityContext
- GrantedAuthority - An authority that is granted to the principal on the Authentication (i.e. roles, scopes, etc.)

Servlet Authentication Architecture (2)

- AuthenticationManager - the API that defines how Spring Security’s Filters perform authentication
- ProviderManager - the most common implementation of AuthenticationManager
- AuthenticationProvider - used by ProviderManager to perform a specific type of authentication
- Request Credentials with AuthenticationEntryPoint - used for requesting credentials from a client (i.e. redirecting to a log in page, sending a WWW-Authenticate response, etc.)
- AbstractAuthenticationProcessingFilter - a base Filter used for authentication. This also gives a good idea of the high level flow of authentication and how pieces work together

SecurityContextHolder

- At the heart of Spring Security’s authentication model is the SecurityContextHolder. It contains the SecurityContext
- The SecurityContextHolder is where Spring Security stores the details of who is authenticated
- Spring Security does not care how the SecurityContextHolder is populated. If it contains a value, it is used as the currently authenticated user



Setting `SecurityContextHolder`

Java Kotlin

JAVA

```java
SecurityContext context = SecurityContextHolder.createEmptyContext(); ❶
Authentication authentication =
    new TestingAuthenticationToken("username", "password", "ROLE_USER"); ❷
context.setAuthentication(authentication);

SecurityContextHolder.setContext(context); ❸
```

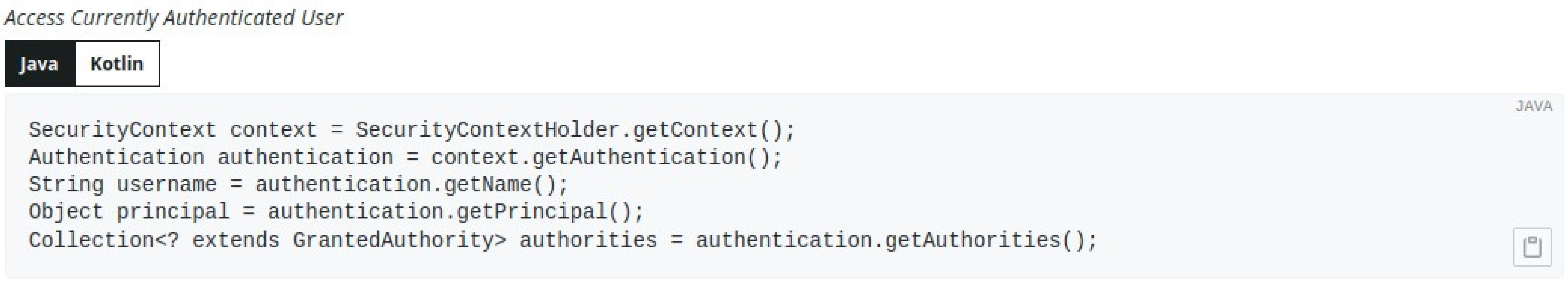
❶ We start by creating an empty `SecurityContext`. You should create a new `SecurityContext` instance instead of using `SecurityContextHolder.getContext().setAuthentication(authentication)` to avoid race conditions across multiple threads.

❷ Next, we create a new `Authentication` object. Spring Security does not care what type of `Authentication` implementation is set on the `SecurityContext`. Here, we use `TestingAuthenticationToken`, because it is very simple. A more common production scenario is `UsernamePasswordAuthenticationToken(userDetails, password, authorities)`.

❸ Finally, we set the `SecurityContext` on the `SecurityContextHolder`. Spring Security uses this information for authorization.

SecurityContext

- The SecurityContext is obtained from the SecurityContextHolder. The SecurityContext contains an Authentication object


Access Currently Authenticated User

```java
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
```

Authentication

- The Authentication interface serves two main purposes within Spring Security:
  - An input to AuthenticationManager to provide the credentials a user has provided to authenticate. When used in this scenario, isAuthenticated() returns false
  - Represent the currently authenticated user. You can obtain the current Authentication from the SecurityContext
- The Authentication contains:
  - principal: Identifies the user. When authenticating with a username/password this is often an instance of UserDetails
  - credentials: Often a password. In many cases, this is cleared after the user is authenticated, to ensure that it is not leaked
  - authorities: The GrantedAuthority instances are high-level permissions the user is granted. Two examples are roles and scopes

GrantedAuthority (1)

- GrantedAuthority instances are high-level permissions that the user is granted. Two examples are roles and scopes
- You can obtain GrantedAuthority instances from the Authentication.getAuthorities() method. This method provides a Collection of GrantedAuthority objects
- A GrantedAuthority is, not surprisingly, an authority that is granted to the principal
- Such authorities are usually “roles”, such as ROLE_ADMINISTRATOR or ROLE_HR_SUPERVISOR

GrantedAuthority (2)

- These roles are later configured for web authorization, method authorization, and domain object authorization
- Usually, the GrantedAuthority objects are application-wide permissions
- They are not specific to a given domain object
- Thus, you would not likely have a GrantedAuthority to represent a permission to Employee object number 54, because if there are thousands of such authorities you would quickly run out of memory

AuthenticationManager

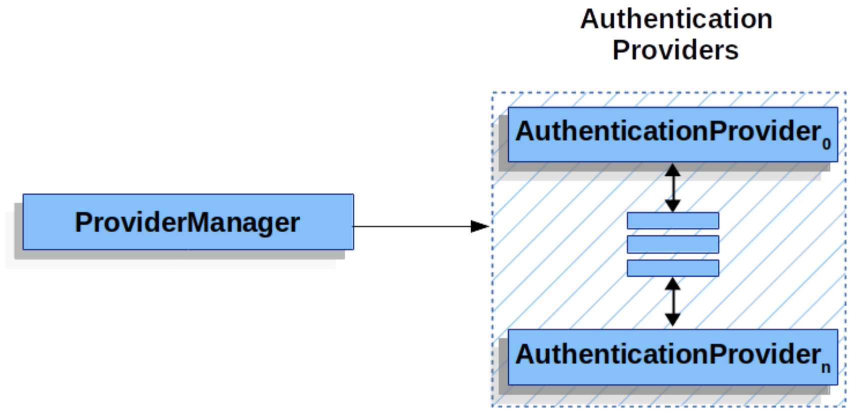
- AuthenticationManager is the API that defines how Spring Security's Filters perform authentication
- The Authentication that is returned is then set on the SecurityContextHolder by the controller (that is, by Spring Security's Filters instances) that invoked the AuthenticationManager
- While the implementation of AuthenticationManager could be anything, the most common implementation is ProviderManager

ProviderManager (1)

- AuthenticationManager is the API that defines how Spring Security’s Filters perform authentication
- The Authentication that is returned is then set on the SecurityContextHolder by the controller (that is, by Spring Security’s Filters instances) that invoked the AuthenticationManager
- While the implementation of AuthenticationManager could be anything, the most common implementation is ProviderManager
- If none of the configured AuthenticationProvider instances can authenticate, authentication fails with a ProviderNotFoundException

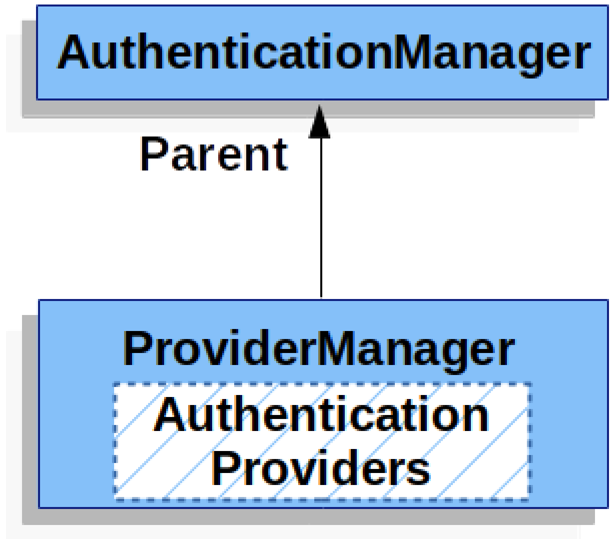
ProviderManager (2)

- In practice each AuthenticationProvider knows how to perform a specific type of authentication
- One AuthenticationProvider might be able to validate a username/password, while another might be able to authenticate a SAML assertion
- This lets each AuthenticationProvider do a very specific type of authentication while supporting multiple types of authentication and expose only a single AuthenticationManager bean



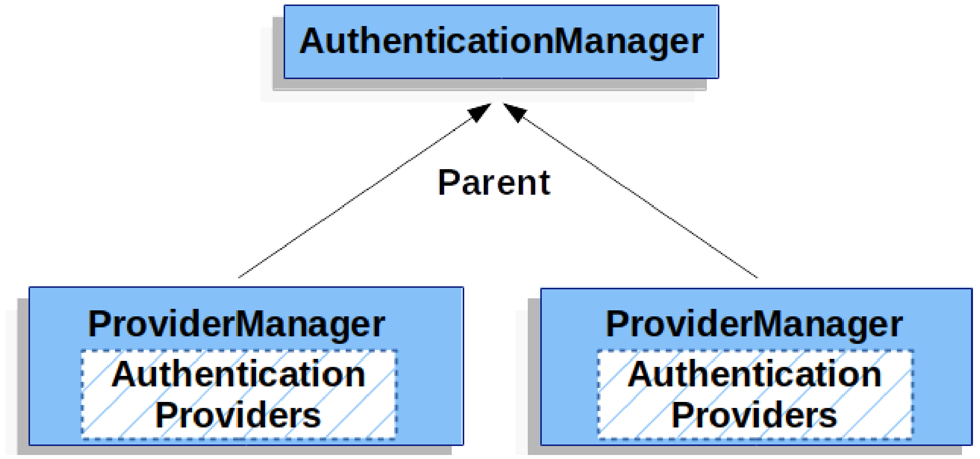
ProviderManager (3)

- ProviderManager also allows configuring an optional parent AuthenticationManager, which is consulted in the event that no AuthenticationProvider can perform authentication
- The parent can be any type of AuthenticationManager, but it is often an instance of ProviderManager



ProviderManager (4)

- In fact, multiple ProviderManager instances might share the same parent AuthenticationManager
- This is somewhat common in scenarios where there are multiple SecurityFilterChain instances that have some authentication in common (the shared parent AuthenticationManager), but also different authentication mechanisms (the different ProviderManager instances)



AuthenticationProvider

- You can inject multiple AuthenticationProviders instances into ProviderManager
- Each AuthenticationProvider performs a specific type of authentication
- For example, DaoAuthenticationProvider supports username/password-based authentication, while JwtAuthenticationProvider supports authenticating a JWT token

Request Credentials with AuthenticationEntryPoint

- Sometimes, a client proactively includes credentials (such as a username and password) to request a resource
- In these cases, Spring Security does not need to provide an HTTP response that requests credentials from the client, since they are already included
- In other cases, a client makes an unauthenticated request to a resource that they are not authorized to access. In this case, an implementation of AuthenticationEntryPoint is used to request credentials from the client
- The AuthenticationEntryPoint implementation might perform a redirect to a log in page, respond with an WWW-Authenticate header, or take other action

AbstractAuthenticationProcessingFilter (1)

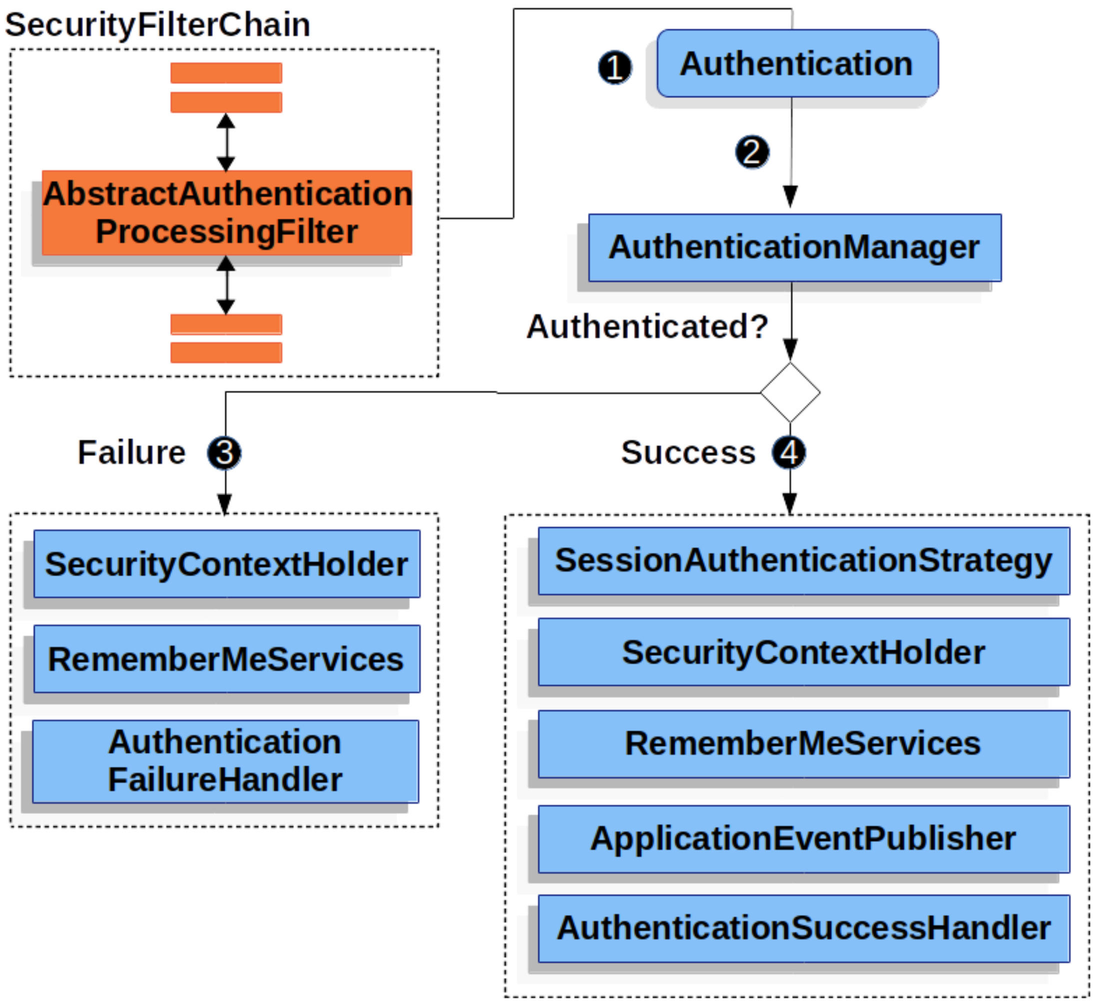
- AbstractAuthenticationProcessingFilter is used as a base Filter for authenticating a user’s credentials
- Before the credentials can be authenticated, Spring Security typically requests the credentials by using AuthenticationEntryPoint
- Next, the AbstractAuthenticationProcessingFilter can authenticate any authentication requests that are submitted to it

AbstractAuthenticationProcessingFilter (2)

- When the user submits their credentials, the AbstractAuthenticationProcessingFilter creates an Authentication from the HttpServletRequest to be authenticated
- The type of Authentication created depends on the subclass of AbstractAuthenticationProcessingFilter
- UsernamePasswordAuthenticationFilter creates a UsernamePasswordAuthenticationToken from a username and password that are submitted in the HttpServletRequest

AbstractAuthenticationProcessingFilter (3)

- Next, the Authentication is passed into the AuthenticationManager to be authenticated
- If authentication fails, then Failure
  - The SecurityContextHolder is cleared out
  - RememberMeServices.loginFail is invoked. If remember me is not configured, this is a no-op
  - AuthenticationFailureHandler is invoked
- If authentication is successful, then Success
  - SessionAuthenticationStrategy is notified of a new login
  - The Authentication is set on the SecurityContextHolder
  - RememberMeServices.loginSuccess is invoked. If remember me is not configured, this is a no-op
  - ApplicationEventPublisher publishes an InteractiveAuthenticationSuccessEvent
  - AuthenticationSuccessHandler is invoked


SecurityFilterChain
AbstractAuthentication ProcessingFilter
1
Authentication
2
AuthenticationManager
Authenticated?
Failure
3
Success
4
SecurityContextHolder
RememberMeServices
Authentication FailureHandler
SessionAuthenticationStrategy
SecurityContextHolder
RememberMeServices
ApplicationEventPublisher
AuthenticationSuccessHandler

Authorization Architecture – Authorities (1)

- Authentication discusses how all Authentication implementations store a list of GrantedAuthority objects. These represent the authorities that have been granted to the principal
- The GrantedAuthority objects are inserted into the Authentication object by the AuthenticationManager and are later read by AccessDecisionManager instances when making authorization decisions
- The GrantedAuthority interface has only one method - String getAuthority();

Authorization Architecture – Authorities (2)

- This method is used by an AuthorizationManager instance to obtain a precise String representation of the GrantedAuthority
- By returning a representation as a String, a GrantedAuthority can be easily "read" by most AuthorizationManager implementations
- If a GrantedAuthority cannot be precisely represented as a String, the GrantedAuthority is considered "complex" and getAuthority() must return null

Authorization Architecture – Authorities (3)

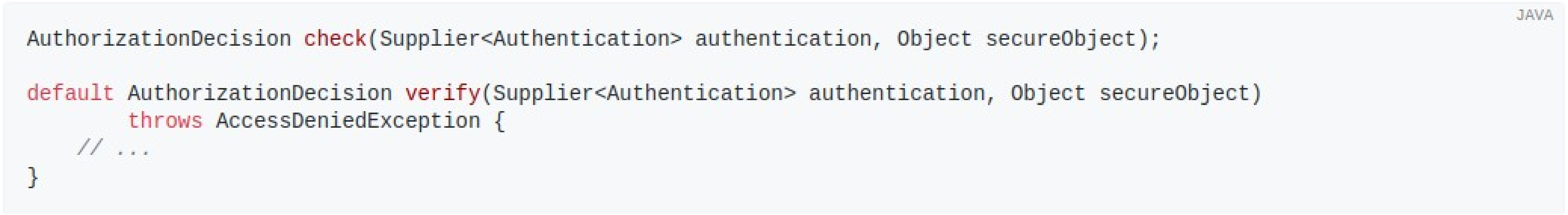
- Spring Security includes one concrete GrantedAuthority implementation: SimpleGrantedAuthority
- This implementation lets any user-specified String be converted into a GrantedAuthority
- All AuthenticationProvider instances included with the security architecture use SimpleGrantedAuthority to populate the Authentication object
- By default, role-based authorization rules include ROLE_ as a prefix. This means that if there is an authorization rule that requires a security context to have a role of "USER", Spring Security will by default look for a GrantedAuthority#getAuthority that returns "ROLE_USER"

Invocation Handling

- Spring Security provides interceptors that control access to secure objects, such as method invocations or web requests
- A pre-invocation decision on whether the invocation is allowed to proceed is made by AuthorizationManager instances
- Also post-invocation decisions on whether a given value may be returned is made by AuthorizationManager instances

Invocation Handling - The AuthorizationManager (1)

- AuthorizationManagers are called by Spring Security’s request-based, method-based, and message-based authorization components and are responsible for making final access control decisions
- The AuthorizationManager interface contains two methods



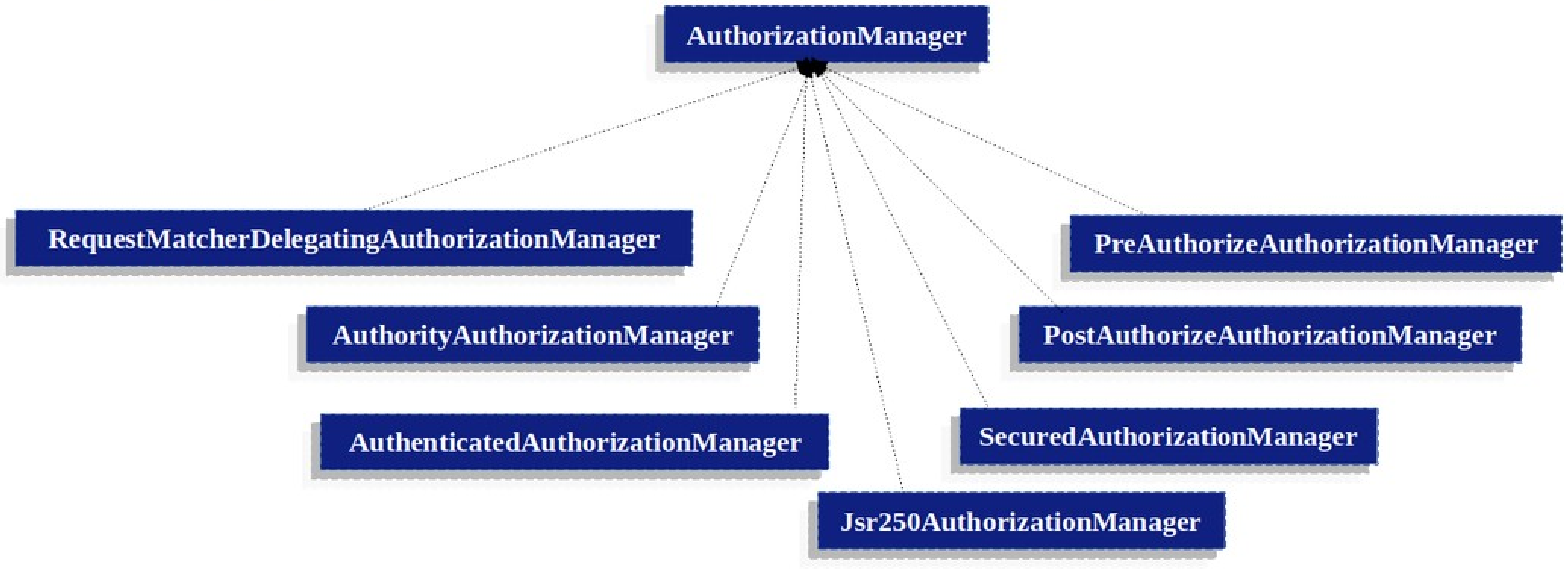
```java
AuthorizationDecision check(Supplier<Authentication> authentication, Object secureObject);

default AuthorizationDecision verify(Supplier<Authentication> authentication, Object secureObject)
        throws AccessDeniedException {
    // ...
}
```

Invocation Handling - The AuthorizationManager (2)

- Implementations are expected to return a positive AuthorizationDecision if access is granted
- Negative AuthorizationDecision if access is denied
- null AuthorizationDecision when abstaining from making a decision
- verify calls check and subsequently throws an AccessDeniedException in the case of a negative AuthorizationDecision

Invocation Handling - The AuthorizationManager (3)

- Whilst users can implement their own AuthorizationManager to control all aspects of authorization, Spring Security ships with a delegating AuthorizationManager that can collaborate with individual AuthorizationManager
- RequestMatcherDelegatingAuthorizationManager will match the request with the most appropriate delegate AuthorizationManager. For method security, you can use AuthorizationManagerBeforeMethodInterceptor and AuthorizationManagerAfterMethodInterceptor
- Using this approach, a composition of AuthorizationManager implementations can be polled on an authorization decision


AuthorizationManager
RequestMatcherDelegatingAuthorizationManager
AuthorityAuthorizationManager
AuthenticatedAuthorizationManager
Jsr250AuthorizationManager
SecuredAuthorizationManager
PostAuthorizeAuthorizationManager
PreAuthorizeAuthorizationManager

AuthorityAuthorizationManager

- The most common AuthorizationManager provided with Spring Security is AuthorityAuthorizationManager
- It is configured with a given set of authorities to look for on the current Authentication
- It will return positive AuthorizationDecision should the Authentication contain any of the configured authorities
- It will return a negative AuthorizationDecision otherwise

AuthenticatedAuthorizationManager

- Another manager is the AuthenticatedAuthorizationManager
- It can be used to differentiate between anonymous, fully-authenticated and remember-me authenticated users
- Many sites allow certain limited access under remember-me authentication, but require a user to confirm their identity by logging in for full access

Custom Authorization Managers

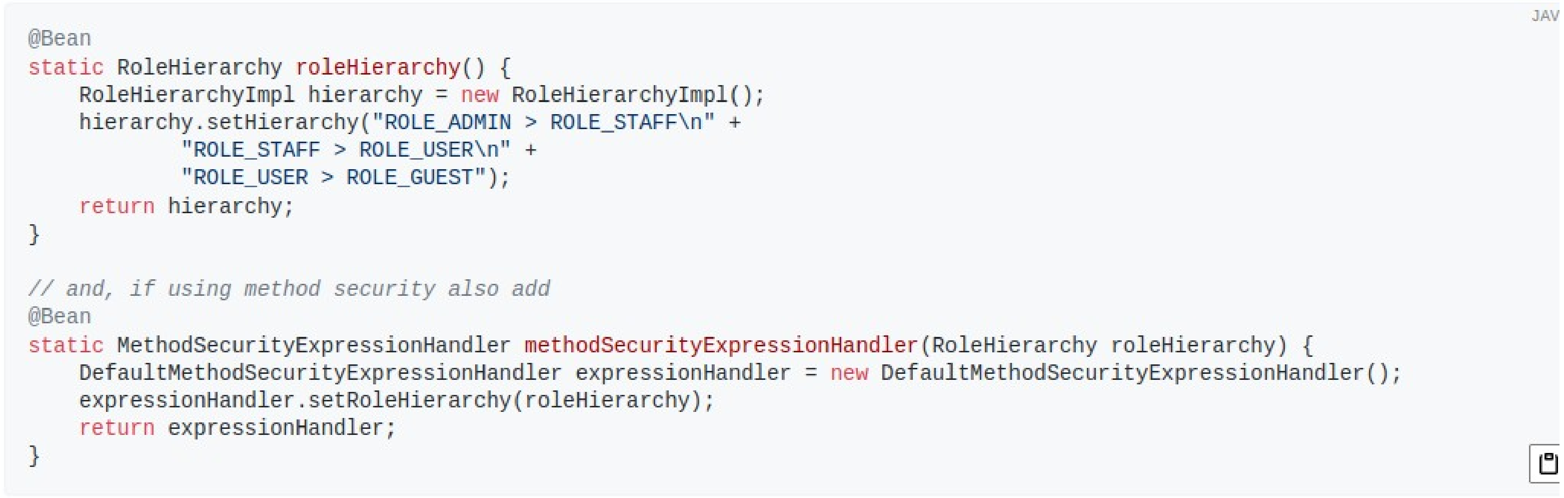
- There are also helpful static factories in AuthenticationManagers for composing individual AuthenticationManagers into more sophisticated expressions
- Obviously, you can also implement a custom AuthorizationManager and you can put just about any access-control logic you want in it
- It might be specific to your application (business-logic related) or it might implement some security administration logic

Hierarchical Roles (1)

- It is a common requirement that a particular role in an application should automatically "include" other roles
- For example, in an application which has the concept of an "admin" and a "user" role, you may want an admin to be able to do everything a normal user can
- To achieve this, you can either make sure that all admin users are also assigned the "user" role
- Alternatively, you can modify every access constraint which requires the "user" role to also include the "admin" role. This can get quite complicated if you have a lot of different roles in your application

Hierarchical Roles (2)

- Here we have four roles in a hierarchy ROLE_ADMIN ⇒ ROLE_STAFF ⇒ ROLE_USER ⇒ ROLE_GUEST
- A user who is authenticated with ROLE_ADMIN, will behave as if they have all four roles when security constraints are evaluated against an AuthorizationManager adapted to call the above RoleHierarchyVoter



```java
@Bean
static RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl hierarchy = new RoleHierarchyImpl();
    hierarchy.setHierarchy("ROLE_ADMIN > ROLE_STAFF\n" +
            "ROLE_STAFF > ROLE_USER\n" +
            "ROLE_USER > ROLE_GUEST");
    return hierarchy;
}

// and, if using method security also add
@Bean
static MethodSecurityExpressionHandler methodSecurityExpressionHandler(RoleHierarchy roleHierarchy) {
    DefaultMethodSecurityExpressionHandler expressionHandler = new DefaultMethodSecurityExpressionHandler();
    expressionHandler.setRoleHierarchy(roleHierarchy);
    return expressionHandler;
}
```