SPRING SECURITY CORE CONCEPT

Ebenezer Arkoh-Addo

Contents

1 . SecurityFilterChain

This bean configures the security filters that Spring Security will use to handle HTTP requests. It replaces the deprecated `WebSecurityConfigurerAdapter` for setting up security.

Security Filters: are responsible for processing incoming requests to your application and applying security rules like authentication and authorization.

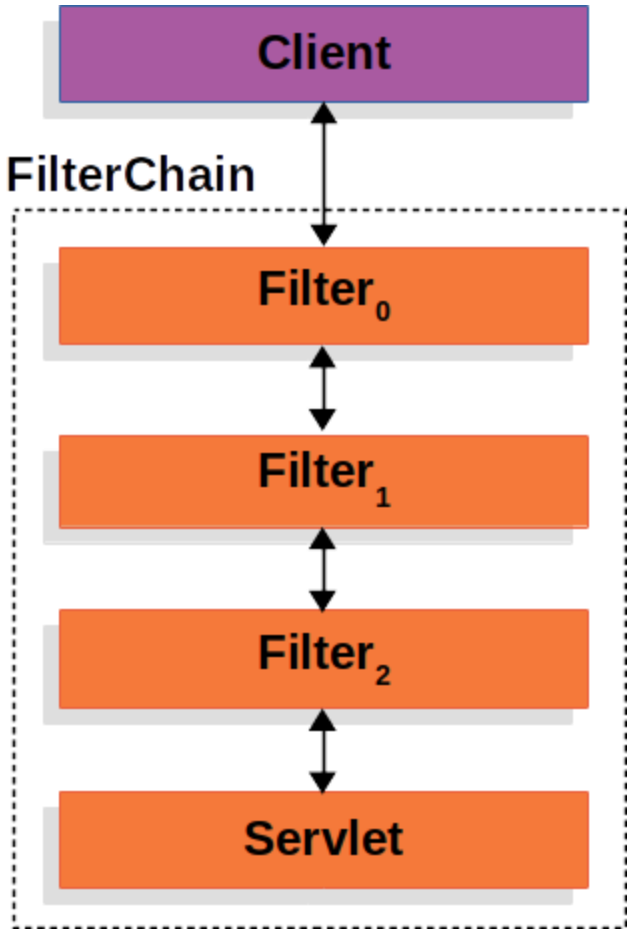
In the `SecurityFilterChain` bean, you define which endpoints are protected, what kind of authentication is required (e.g., form login, basic auth), and any other security settings.

Example

```
@Bean
public SecurityFilterChain securityFilterChain (HttpSecurity http) throws Exception {
   http
      . authorizeRequests(authorizeRequests ->
         authorizeRequests
            . antMatchers("/public/**").permitAll()  // Allow access to public endpoints
            . anyRequest().authenticated()          // Require authentication for other requests
      )
      . formLogin(withDefaults ());  // Use form-based login by default
   return http. Build();
}
```

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.



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

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

        // do something before the rest of the application
```

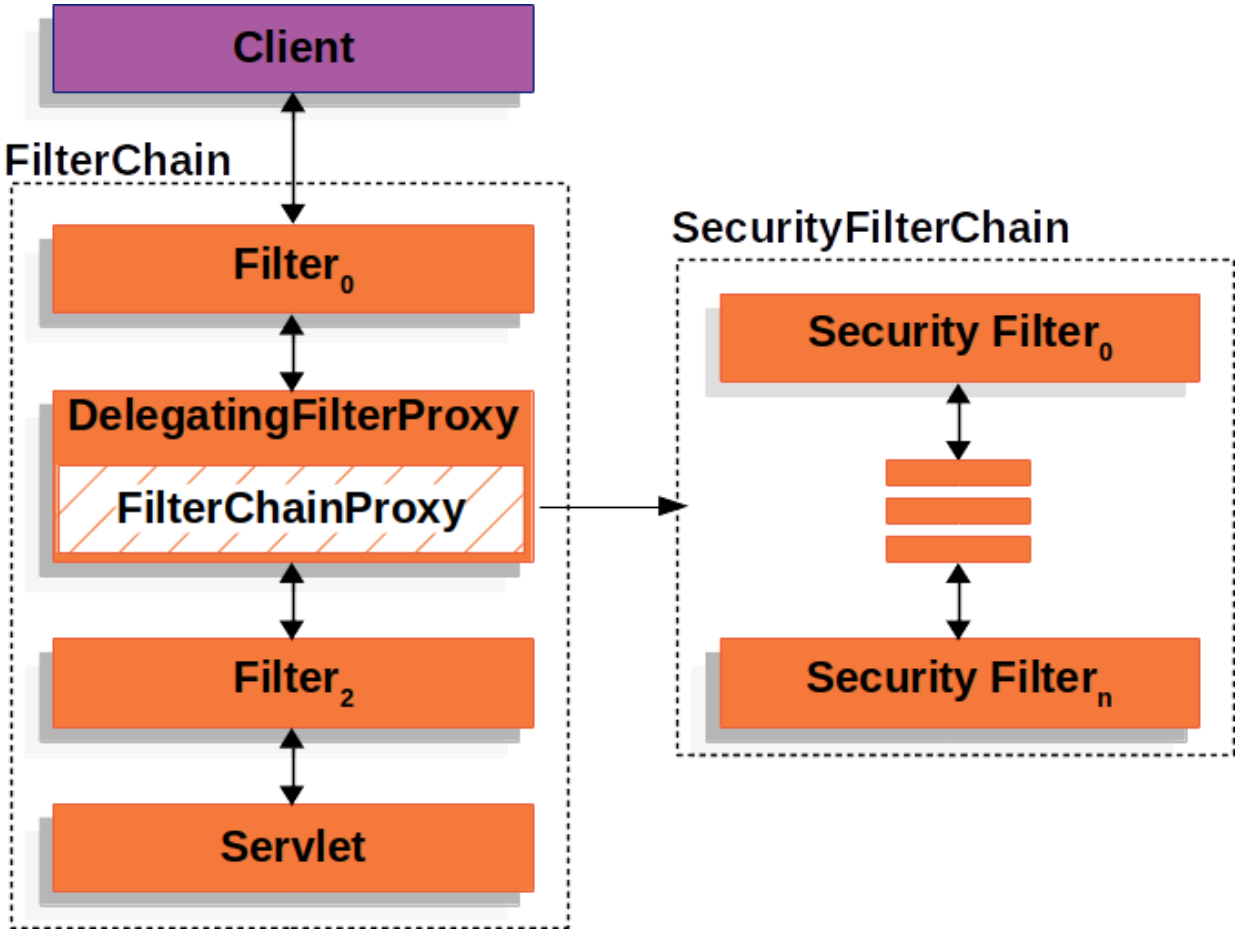
```
    chain.doFilter(request, response); // invoke the rest of the application

    // do something after the rest of the application
}
```

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

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.



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

Adding a Custom Filter 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 security filter chain.

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. 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.

First, let's create the Filter:

```
import java.io.IOException;


import jakarta.servlet.Filter;

import jakarta.servlet.FilterChain;

import jakarta.servlet.ServletException;

import jakarta.servlet.ServletRequest;

import jakarta.servlet.ServletResponse;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;


import org.springframework.security.access.AccessDeniedException;


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");
        boolean hasAccess = isUserAllowed(tenantId);
        if (hasAccess) {
            filterChain.doFilter(request, response);
            return;
        }
        throw new AccessDeniedException("Access denied");
    }

}
```

The sample code above does the following:

Get the tenant id from the request header.

Check if the current user has access to the tenant id.

If the user has access, then invoke the rest of the filters in the chain.

If the user does not have access, then throw an AccessDeniedException.

2. User Details Service Bean

This bean is responsible for retrieving user-related data. It is used by Spring Security to load user-specific data during the authentication process.

User details service is an interface provided by Spring Security that has a method (`loadUserByUsername`) to fetch user details based on the username.

Example

```
@Bean
public UserDetailsService userDetailsService() {
   return username -> {
      User user = userRepository.findByUsername(username);
      return User.withUsername(user.getUsername())
               .password(user.getPassword())
               .authorities(user.getAuthorities())
               .build();
   };
}
```

UserDetailsService is used by DaoAuthenticationProvider for retrieving a username, a password, and other attributes for authenticating with a username and password. Spring Security provides in-memory, JDBC, and caching implementations of UserDetailsService.

Implementing CustomUserDetails

```
public class UserDetailsServiceImp implements UserDetailsService {
 @Override
 public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {


   User user = findUserbyUername(username);
```

```
    UserBuilder builder = null;
    if (user != null) {
      builder = org.springframework.security.core.userdetails.User.withUsername(username);
      builder.password(new BCryptPasswordEncoder().encode(user.getPassword()));
      builder.roles(user.getRoles());
    } else {
      throw new UsernameNotFoundException("User not found.");
    }

    return builder.build();
  }

  private User findUserbyUername(String username) {
    if(username.equalsIgnoreCase("admin")) {
      return new User(username, "admin123", "ADMIN");
    }
    return null;
  }
}
```

3. Password Encoder Bean

This bean provides a mechanism to encode passwords for secure storage and comparison during authentication. Plain text passwords are hashed using an encoder before storing them in the database. During login, the entered password is hashed and compared with the stored hash.

Spring Security, provides several password encoders (or hashers) that you can use depending on your security needs. Here's a list of the most common ones:

Common Encoders

BCryptPasswordEncoder

Uses the BCrypt hashing function, which is a widely recommended option for securing passwords because it is slow and resistant to brute-force attacks.

```
@Bean
public PasswordEncoder passwordEncoder() {
   return new BCryptPasswordEncoder();
}
```

Strengths: Adjustable work factor (strength), making it more secure over time.

Usage: Most commonly used and recommended for most applications.

`NoOpPasswordEncoder`

A placeholder encoder that does not perform any hashing and stores the password as plain text.

Strengths: Simple and useful for testing or demo purposes only.

Security Risk: Not recommended for production** because it does not secure passwords at all.

```
PasswordEncoder encoder = NoOpPasswordEncoder.getInstance();
```

`Pbkdf2PasswordEncoder’

Uses the PBKDF2 (Password-Based Key Derivation Function 2) hashing algorithm.

Strengths: Highly configurable, allowing you to set the iteration count, key length, and salt size.

Usage: Suitable for systems that require compliance with specific standards like NIST or FIPS.

PasswordEncoder encoder = new Pbkdf2PasswordEncoder();

`SCryptPasswordEncoder`

Uses the SCrypt hashing function, designed to be memory-intensive to resist hardware brute-force attacks.

Strengths: More resistant to specialized hardware attacks (like FPGA, ASIC) compared to other algorithms like BCrypt.

Usage: Recommended when you need higher security against brute-force attacks.

PasswordEncoder encoder = new SCryptPasswordEncoder();

`Argon2PasswordEncoder`

Uses the Argon2 hashing function, which won the Password Hashing Competition in 2015.

Strengths: Highly configurable with options for memory usage, parallelism, and iterations, providing a strong defence against attacks.

Usage: Ideal for high-security applications needing modern protection.

PasswordEncoder encoder = new Argon2PasswordEncoder();

4. Authentication Provider Bean

This bean is used to authenticate users by checking their credentials against the data source. An interface that provides authentication logic. You implement this to define how user credentials are validated (e.g., by checking a database). In a typical Spring Boot setup, you might not need to explicitly define this bean if you are using `DaoAuthenticationProvider`, which is automatically configured if you provide a `UserDetailsService` and `PasswordEncoder`.

```
@Bean
public AuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
```

```
    provider.setUserDetailsService(userDetailsService());
    provider.setPasswordEncoder(passwordEncoder());
    return provider;
}
```

Spring Security provides a variety of options for performing authentication. These options follow a simple contract: **an *AuthenticationProvider* processes an *Authentication* request*,*** and a fully authenticated object with full credentials is returned.

The standard and most common implementation is the *DaoAuthenticationProvider,* which retrieves the user details from the *UserDetailsService*. This User Details Service **only has access to the username** in order to retrieve the full user entity, which is enough for most scenarios.

More custom scenarios will still need to access the full *Authentication* request to be able to perform the authentication process. For example, when authenticating against some external, third-party service (such as Crowd), **both the *username* and *password* from the authentication request will be necessary**.

Implementating Custom AuthenticationProvider

```
@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {


    @Override
    public Authentication authenticate(final Authentication authentication) throws
AuthenticationException {
        final String name = authentication.getName();
        final String password = authentication.getCredentials().toString();
        if (!"admin".equals(name) || !"system".equals(password)) {
            return null;
        }
        return authenticateAgainstThirdPartyAndGetAuthentication(name, password);
```

```
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

Here, we have a generic method that returns an *Authentication* object. Its implementation can vary based on how we want to authenticate. As an example, we can write an example of a fixed credentials method:

```
private static UsernamePasswordAuthenticationToken
authenticateAgainstThirdPartyAndGetAuthentication(String name, String password) {
    final List<GrantedAuthority> grantedAuths = new ArrayList<>();
    grantedAuths.add(new SimpleGrantedAuthority("ROLE_USER"));
    final UserDetails principal = new User(name, password, grantedAuths);
    return new UsernamePasswordAuthenticationToken(principal, password, grantedAuths);
}
```

It is worth noting that we also add an authority to our *UserDetails* object

Register the Auth Provider

After we've defined the Authentication Provider, we need to specify it in the XML Security Configuration using the available namespace support:

```
<http use-expressions="true">
    <intercept-url pattern="/**" access="isAuthenticated()"/>
    <http-basic/>
</http>
<authentication-manager>
```

```
    <authentication-provider
      ref="customAuthenticationProvider" />
</authentication-manager>
```

Using the Authentication provider in security configuration

```
@Configuration
@EnableWebSecurity
@ComponentScan("com.baeldung.security")
public class SecurityConfig {
    @Autowired
    private CustomAuthenticationProvider authProvider;

    @Bean
    public AuthenticationManager authManager(HttpSecurity http) throws Exception {
        AuthenticationManagerBuilder authenticationManagerBuilder =
            http.getSharedObject(AuthenticationManagerBuilder.class);
        authenticationManagerBuilder.authenticationProvider(authProvider);
        return authenticationManagerBuilder.build();
    }

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

```
    }
}
```

5. DaoAuthenticationProvider

DaoAuthenticationProvider is an AuthenticationProvider implementation that uses a UserDetailsService and PasswordEncoder to authenticate a username and password.

The following figure explains the workings of the AuthenticationManager in figures from the Reading the Username & Password section.

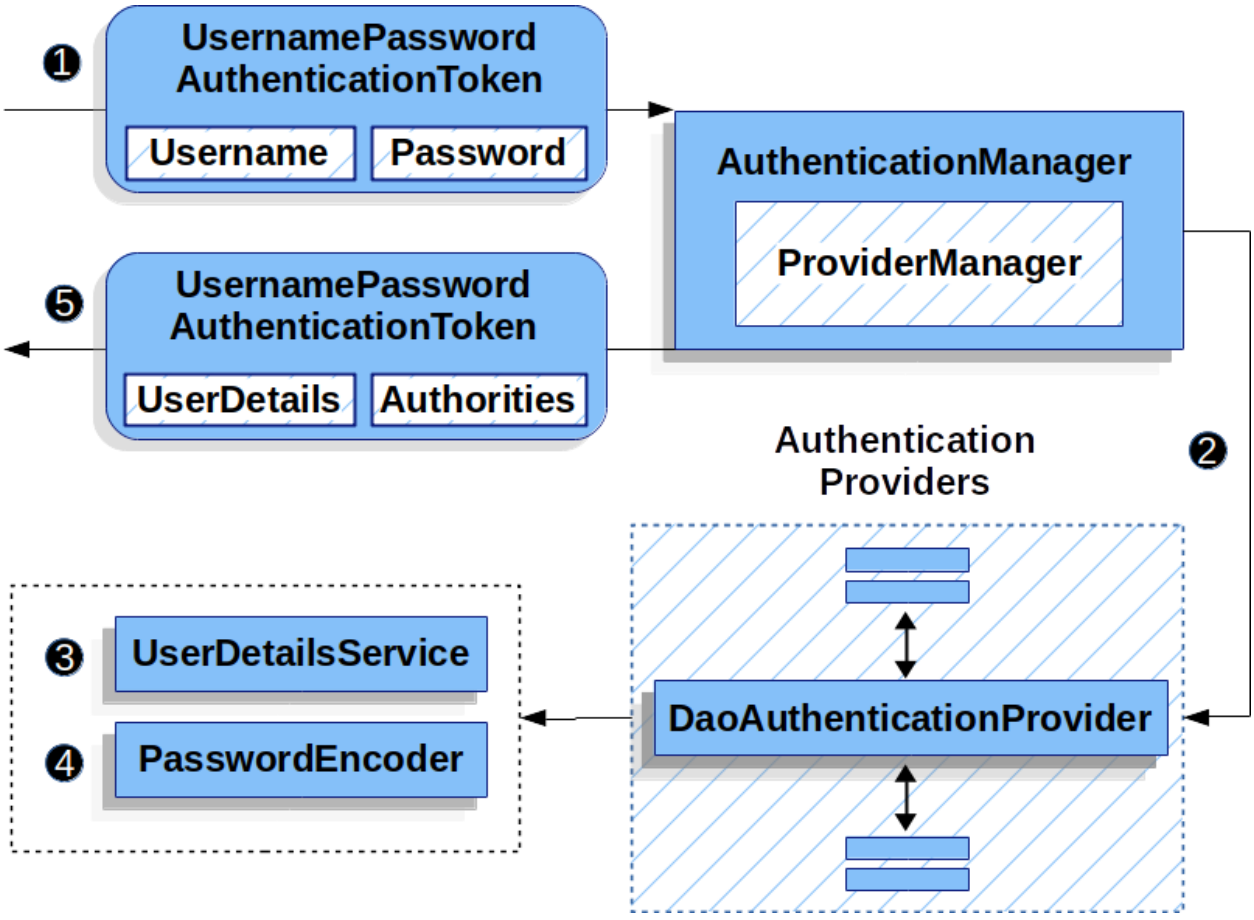


Figure 1. DaoAuthenticationProvider Usage

1 The authentication passes a UsernamePasswordAuthenticationToken to the AuthenticationManager, which is implemented by ProviderManager.

❷The ProviderManager is configured to use an AuthenticationProvider of type DaoAuthenticationProvider.

❸DaoAuthenticationProvider looks up the UserDetails from the UserDetailsService.

❹DaoAuthenticationProvider uses the PasswordEncoder to validate the password on the UserDetails returned in the previous step.

❺When authentication is successful, the Authentication that is returned is of type UsernamePasswordAuthenticationToken and has a principal that is the UserDetails returned by the configured UserDetailsService. Ultimately, the returned UsernamePasswordAuthenticationToken is set on the SecurityContextHolder by the authentication Filter.

Summary

These 4 components(SecurityFilterChain, UserDetailsService, PasswordEncoder and Authentication Provider) work together to ensure that your Spring Boot application securely handles authentication and authorization.

SecurityFilterChain`: Defines security rules for HTTP requests.

`UserDetailsService`: Fetches user details from a data source(By default spring security uses username field on users table to find users. It also creates a user with username = user and password generated in console every time application runs).

`Password Encoder`: Handles password encoding and decoding.

Authentication Provider`: Manages user authentication (By default spring security uses DaoAuthenticationProvider)

References

1. Spring Security. (n.d.). *Password storage and encoding*. Spring.io. Retrieved August 25, 2024, from https://docs.spring.io/spring-

security/reference/servlet/authentication/passwords/index.html#servlet-authentication-unpwd-input

2. Spring Security. (n.d.). *DAO authentication provider*. Spring.io. Retrieved August 25, 2024, from https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/dao-authentication-provider.html

3. Spring Security. (n.d.). *AuthenticationProvider architecture*. Spring.io. Retrieved August 25, 2024, from https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html#servlet-authentication-authenticationprovider.

4. Spring Security. (n.d.). *Spring Security servlet architecture*. Spring.io. Retrieved August 25, 2024, from https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-architecture.

5. Spring Security. (n.d.). *Spring Security servlet architecture overview*. Spring.io. Retrieved August 25, 2024, from https://docs.spring.io/spring-security/reference/servlet/architecture.html.

6. GeeksforGeeks. (n.d.). *Spring Security authentication providers*. GeeksforGeeks. Retrieved August 25, 2024, from https://www.geeksforgeeks.org/spring-security-authentication-providers/.