

6-AuthN vs AuthZ Explained



Kindly explain the above point with detailed explanation related to Authorities and Authorization



The image compares **Authentication (AuthN)** and **Authorization (AuthZ)** with key differences. Below is a detailed explanation of the points with an emphasis on the relationship between authorities and authorization:

Authentication (AuthN)

1. Identity Verification:

- Authentication confirms the **identity** of a user trying to access the system.
- For example, you enter your username and password when logging into an app. The system checks whether these credentials are valid.

2. Authentication Happens First:

- Before any resource access is granted, the system must verify who you are.
- Without authentication, the system cannot proceed to authorization.

3. Relies on Login Details:

- Typical methods of authentication include passwords, biometrics (e.g., fingerprint), or multi-factor authentication (MFA).
- These methods prove that you are the person you claim to be.

4. **401 Error for Failure:**

- If authentication fails (e.g., wrong username/password), the system responds with **401 Unauthorized**. This means the system cannot determine who you are.

5. **Example:**

- A bank customer logging into an online banking application proves their identity by providing valid login credentials.
-

Authorization (AuthZ)

1. **Authority Verification:**

- After the system identifies the user through authentication, **authorization checks what resources the user is allowed to access**.
- The system validates the **user's roles, permissions, or authorities** against the requested action/resource.

2. **Happens After Authentication:**

- Authorization always occurs after authentication. Without knowing who you are, the system cannot determine what actions you are permitted to perform.

3. **Requires Privileges or Roles:**

- Authorization depends on the roles/privileges assigned to the user.
- For instance, an "Admin" role might have access to manage users, while a "User" role might only view their own account.

4. **403 Error for Failure:**

- If a user does not have sufficient permissions to perform an action, the system responds with **403 Forbidden**. This means the user is authenticated but lacks authorization for the specific resource or action.

5. **Example:**

- After logging into the banking application, the customer's role (e.g., "Customer") determines what actions they can perform, like checking account balances or transferring money. A bank employee might have a different role allowing access to customer details but restricting actions like fund transfers.
-

Role of Authorities in Authorization

- **Authorities** are specific permissions or entitlements assigned to users or roles in the system.

- For example:
 - A "Customer" may have the authority to **view their account details** but not to **view other customers' details**.
 - An "Admin" might have broad authorities, such as creating, deleting, or modifying user accounts.
 - **Authorization Process with Authorities:**
 - Once a user logs in, their roles and associated authorities are validated.
 - The system matches these authorities with the resource or action being requested to decide whether to permit or deny access.
 - **Practical Implementation:**
 - In software, authorities are often mapped to specific endpoints or actions.
 - For example, in a REST API:
 - Endpoint `/user/create`: Requires "Admin" authority.
 - Endpoint `/user/profile`: Requires "User" authority.
-

Summary of Interaction Between Authentication and Authorization

1. **Authentication (AuthN):** Establishes **who you are**.
 - Focuses on validating the user's identity.
2. **Authorization (AuthZ):** Determines **what you can do**.
 - Focuses on validating the user's privileges, roles, and authorities to perform actions or access resources.

If you'd like, I can provide additional real-world examples or technical insights into how authorities and roles are implemented in systems!



HOW AUTHORITIES STORED ?

INSIDE SPRING SECURITY

**eazy
bytes**

- Authorities/Roles information in Spring Security is stored inside **GrantedAuthority**. There is only one method inside **GrantedAuthority** which return the name of the authority or role.
- **SimpleGrantedAuthority** is the default implementation class of **GrantedAuthority** interface inside Spring Security framework.

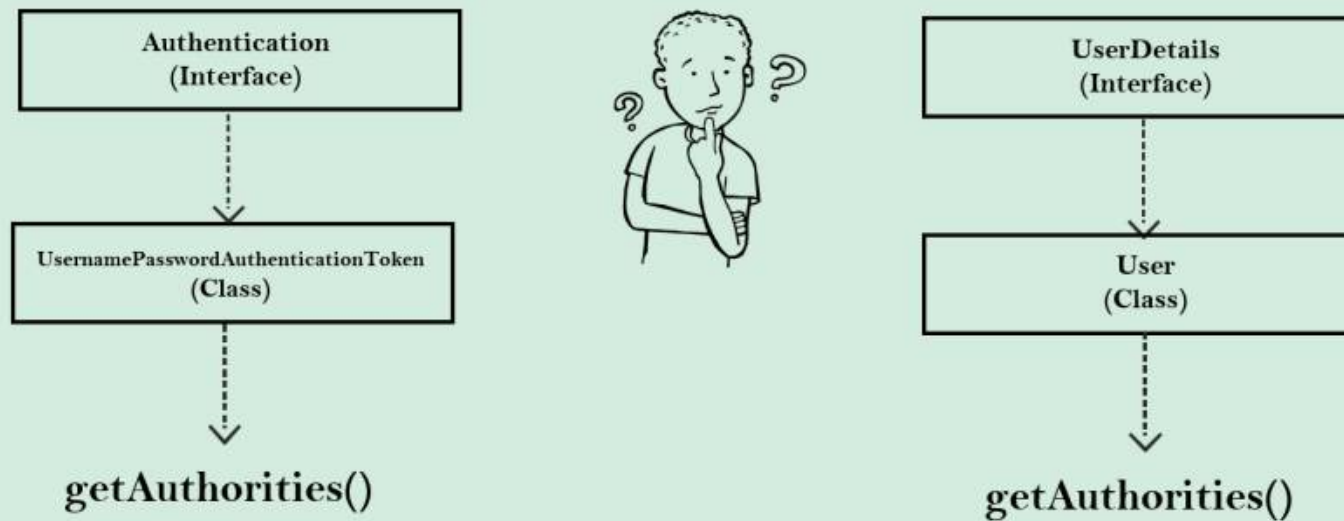
```
public interface GrantedAuthority {  
  
    String getAuthority();  
  
}
```

```
public final class SimpleGrantedAuthority implements GrantedAuthority {  
  
    2 usages  
    private final String role;  
  
    public SimpleGrantedAuthority(String role) {  
        this.role = role;  
    }  
  
    @Override  
    public String getAuthority() {  
        return this.role;  
    }  
  
}
```

HOW AUTHORITIES STORED ?

INSIDE SPRING SECURITY

How does Authorities information stored inside the objects of UserDetails & Authentication interfaces which plays a vital role during authentication of the user ?



CONFIGURING AUTHORITIES

INSIDE SPRING SECURITY

**eazy
bytes**

In Spring Security the authorities requirements can be configured using the following ways,

hasAuthority() — Accepts a single authority for which the endpoint will be configured and user will be validated against the single authority mentioned. Only users having the same authority configured can invoke the endpoint.

hasAnyAuthority() — Accepts multiple authorities for which the endpoint will be configured and user will be validated against the authorities mentioned. Only users having any of the authority configured can invoke the endpoint.

access() — Using Spring Expression Language (SpEL) it provides you unlimited possibilities for configuring authorities which are not possible with the above methods. We can use operators like OR, AND inside access() method.

Kindly explain the above image with detailed explanation and along with example



The provided images illustrate concepts related to **authorities** in Spring Security. Let me explain each one in detail:

1. How Authorities Are Stored in Spring Security

- **Storage Mechanism:**
 - In Spring Security, authorities (roles/privileges) are represented by the `GrantedAuthority` interface.
 - The primary method in this interface is `getAuthority()`, which returns the name of the authority (e.g., "ROLE_ADMIN", "ROLE_USER").
- **Implementation:**
 - `SimpleGrantedAuthority` is the default implementation of `GrantedAuthority`.
 - It uses a `String role` to represent the authority name.

Example Code:

```
java

public final class SimpleGrantedAuthority implements GrantedAuthority {
    private final String role;

    public SimpleGrantedAuthority(String role) {
        this.role = role;
    }

    @Override
    public String getAuthority() {
        return this.role;
    }
}
```

- When a user is authenticated, their authorities are typically encapsulated as a collection of `GrantedAuthority` objects.

2. How Authorities Are Stored Inside Authentication and UserDetails

- **Key Interfaces and Classes:**
 - **Authentication Interface:**
 - Represents the token for authentication.
 - Contains the `getAuthorities()` method, which retrieves the user's authorities.
 - **UserDetails Interface:**
 - Provides core user information, including roles/authorities, using the `getAuthorities()` method.

- **Classes:**

- `UsernamePasswordAuthenticationToken``:
 - Stores the authentication information, including authorities, once a user is authenticated.
- `User` Class`:
 - A default implementation of `UserDetails``, which stores user information like username, password, and authorities.

- **Flow:**

- During authentication, authorities are fetched and stored in the `Authentication`` object.
- These authorities are used later for authorization checks.

Example:

```
java

User user = new User("admin", "password", Arrays.asList(new SimpleGrantedAuthority("ROLE_ADMIN")));
UsernamePasswordAuthenticationToken authToken =
    new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
```

3. Configuring Authorities in Spring Security

- Spring Security allows configuring authorities for endpoints or resources using different methods:

- a. `hasAuthority(String authority)``:

- Checks if the user has a specific authority to access an endpoint.
- Example:

```
java

http.authorizeRequests()
    .antMatchers("/admin").hasAuthority("ROLE_ADMIN");
```

- b. `hasAnyAuthority(String... authorities)``:

- Checks if the user has any one of the specified authorities.
- Example:


```
java
```

```
http.authorizeRequests()
    .antMatchers("/dashboard").hasAnyAuthority("ROLE_ADMIN", "ROLE_USER");
```

C. `access(String expression)`:

- Allows more complex expressions using Spring Expression Language (SpEL).
- Example:

```
java
```

```
http.authorizeRequests()
    .antMatchers("/settings").access("hasAuthority('ROLE_ADMIN') and hasIpAddress('192.168.1.1')");
```

Example Use Case

1. User Authentication:

- A user logs in with the username "admin" and password.
- Spring Security validates the credentials and assigns authorities (e.g., `ROLE_ADMIN`).

2. Authorization:

- The user requests access to `/admin`.
- Spring Security checks if the user has `ROLE_ADMIN` using `hasAuthority()`.

3. Implementation:

```
java
```

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/admin").hasAuthority("ROLE_ADMIN")
    }
}
```

```
        .antMatchers("/user").hasAnyAuthority("ROLE_USER", "ROLE_ADMIN")
        .anyRequest().authenticated()
        .and().formLogin();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("admin").password("{noop}password").authorities("ROLE_ADMIN")
            .and()
            .withUser("user").password("{noop}password").authorities("ROLE_USER");
    }
}
```

4. Outcome:

- "Admin" can access `/admin` and `/user`.
- "User" can only access `/user`.

If you need further clarification or additional examples, feel free to ask!



AUTHORITY vs ROLE

INSIDE SPRING SECURITY

AUTHORITY

- Authority is like an individual privilege or an action
- Restricting access in a fine-grained manner
- Ex: VIEWACCOUNT, VIEWCARDS etc.

ROLE

- ROLE is a group of privileges/actions
- Restricting access in a coarse-grained manner
- Ex: ROLE_ADMIN, ROLE_USER

- The names of the authorities/roles are arbitrary in nature and these names can be customized as per the business requirement
- Roles are also represented using the same contract `GrantedAuthority` in Spring Security.
- When defining a role, its name should start with the `ROLE_` prefix. This prefix specifies the difference between a role and an authority.

CONFIGURING AUTHORITIES

INSIDE SPRING SECURITY



In Spring Security the ROLES requirements can be configured using the following ways,

hasRole() — Accepts a single role name for which the endpoint will be configured and user will be validated against the single role mentioned. Only users having the same role configured can invoke the endpoint.

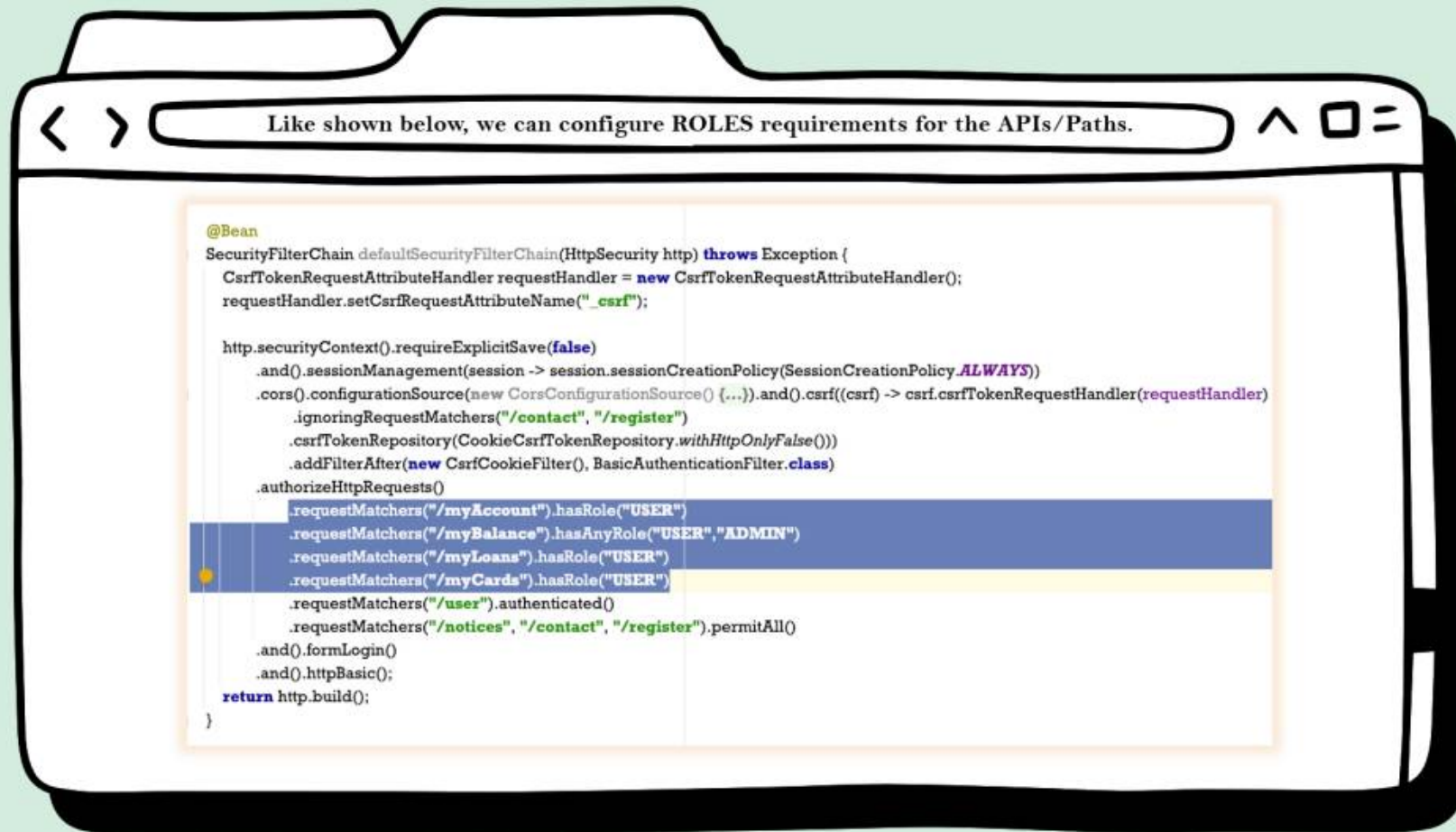
hasAnyRole() — Accepts multiple roles for which the endpoint will be configured and user will be validated against the roles mentioned. Only users having any of the role configured can call the endpoint.

access() — Using Spring Expression Language (SpEL) it provides you unlimited possibilities for configuring roles which are not possible with the above methods. We can use operators like OR, AND inside access() method.

Note :

- ROLE_ prefix only to be used while configuring the role in DB. But when we configure the roles, we do it only by its name.
- access() method can be used not only for configuring authorization based on authority or role but also with any special requirements that we have. For example we can configure access based on the country of the user or current time/date.

CONFIGURING ROLES INSIDE SPRING SECURITY

eazy
bytes

Kindly explain the above image with detailed explanation and along with example



Explanation of the Concepts: Authority vs Role and Role Configuration in Spring Security

1. Authority vs Role in Spring Security

Authority:

- Represents **fine-grained privileges or actions** in the system.
- Typically maps to specific permissions, such as **VIEW_ACCOUNT** or **VIEW_CARDS**.
- Used for **restricting access** to a more granular level.
- Example:
 - A user might have an authority `VIEW_ACCOUNT` to access their bank account details but not edit them.

Role:

- Represents **coarse-grained groupings** of privileges/actions.
- A role is a higher-level abstraction and can include multiple authorities.
- Used for broader access control.
- Example:
 - A role like `ROLE_ADMIN` might encompass authorities such as `VIEW_ACCOUNT`, `EDIT_ACCOUNT`, and `DELETE_ACCOUNT`.

Key Notes:

- Roles are internally handled as authorities in Spring Security with the prefix `ROLE_`.
- Customizing roles/authorities depends on the application's business logic.

2. Configuring Authorities/Roles in Spring Security

Methods for Role Configuration:

1. `hasRole(String role)`:

- Checks if the user has a specific role.
- The role name should exclude the `ROLE_` prefix when used here.
- Example:

```
java

http.authorizeRequests()
    .antMatchers("/admin").hasRole("ADMIN");
```

2. `hasAnyRole(String... roles)`:

- Checks if the user has **any one** of the specified roles.
- Example:

```
java

http.authorizeRequests()
    .antMatchers("/dashboard").hasAnyRole("ADMIN", "USER");
```

3. `access(String expression)`:

- Allows using **Spring Expression Language (SpEL)** for complex conditions.
- Example:

```
java

http.authorizeRequests()
    .antMatchers("/settings").access("hasRole('ADMIN') and hasIpAddress('192.168.1.1')");
```

Note:

- While storing roles in the database, avoid using the `ROLE_` prefix.
- When defining roles programmatically, always use `ROLE_` to differentiate roles from authorities.

3. Configuring Roles in Code

Below is an example of configuring roles for APIs in Spring Security:

Example Code:

```
java

@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.securityContext().requireExplicitSave(false)
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/myAccount").hasRole("USER")
            .requestMatchers("/myBalance").hasAnyRole("USER", "ADMIN")
            .requestMatchers("/myLoans").hasRole("USER")
            .requestMatchers("/myCards").hasRole("USER")
            .requestMatchers("/user").authenticated()
            .requestMatchers("/notices", "/contact", "/register").permitAll()
        )
        .formLogin()
        .and().httpBasic();

    return http.build();
}
```

Explanation of the Code:

1. API Restrictions:

- `/myAccount`: Accessible only by users with the `USER` role.
- `/myBalance`: Accessible by users with either `USER` or `ADMIN` roles.
- `/myLoans` and `/myCards`: Accessible only by `USER` role.
- `/user`: Requires authentication but not specific roles.
- `/notices`, `/contact`, `/register`: Open to all users without authentication.

2. Flow:

- When a user attempts to access an API, Spring Security checks their roles and allows or denies access based on the rules configured.

Key Takeaways

- **Authorities** are granular and focus on specific actions.
- **Roles** are broad and encompass multiple authorities.
- Spring Security provides flexible mechanisms (`hasRole`, `hasAnyRole`, and `access`) for configuring roles/authorities.

- Using SpEL in ``access()`` gives additional flexibility for complex requirements like IP-based restrictions.

Let me know if you'd like further clarification or more examples!



FILTERS IN SPRING SECURITY

**eazy
bytes**

- ✓ Lot of times we will have situations where we need to perform some house keeping activities during the authentication and authorization flow. Few such examples are,
 - Input validation
 - Tracing, Auditing and reporting
 - Logging of input like IP Address etc.
 - Encryption and Decryption
 - Multi factor authentication using OTP
- ✓ All such requirements can be handled using HTTP Filters inside Spring Security. Filters are servlet concepts which are leveraged in Spring Security as well.

- ✓ We already saw some in built filters of Spring security framework like UsernamePasswordAuthenticationFilter, BasicAuthenticationFilter, DefaultLoginPageGeneratingFilter etc. in the previous sections.
- ✓ A filter is a component which receives requests, process its logic and handover to the next filter in the chain.
- ✓ Spring Security is based on a chain of servlet filters. Each filter has a specific responsibility and depending on the configuration, filters are added or removed. We can add our custom filters as well based on the need.

NOT RECOMMENDED
FOR PRODUCTION

FILTERS IN SPRING SECURITY

✓ We can always check the registered filters inside Spring Security with the below configurations,

1. `@EnableWebSecurity(debug = true)` – We need to enable the debugging of the security details
2. Enable logging of the details by adding the below property in application.properties
`logging.level.org.springframework.security.web.FilterChainProxy=DEBUG`

Attached are the some of the internal filters of Spring Security that gets executed in the authentication flow,

```
Security filter chain: [
  DisableEncodeUrlFilter
  WebAsyncManagerIntegrationFilter
  SecurityContextHolderFilter
  HeaderWriterFilter
  CorsFilter
  CsrfFilter
  LogoutFilter
  UsernamePasswordAuthenticationFilter
  DefaultLoginPageGeneratingFilter
  DefaultLogoutPageGeneratingFilter
  BasicAuthenticationFilter
  RequestCacheAwareFilter
  SecurityContextHolderAwareRequestFilter
  AnonymousAuthenticationFilter
  SessionManagementFilter
  ExceptionTranslationFilter
  FilterSecurityInterceptor
]
```

Kindly explain the above image with detailed explanation



Explanation of Filters in Spring Security

Filters are an integral part of Spring Security, forming the backbone of its authentication and authorization mechanisms. Let's break down the details from the provided image.

1. Why Filters Are Used in Spring Security

Filters are used to perform housekeeping or pre/post-processing activities during the authentication and authorization flow. Some common use cases include:

- **Input validation:** Ensuring requests are well-formed before processing.
- **Tracing, auditing, and reporting:** Tracking user activities for compliance or debugging.
- **Logging:** Capturing details like IP addresses or request metadata.
- **Encryption and decryption:** Securing sensitive data in transit.
- **Multi-factor authentication:** Implementing additional security layers like OTPs.

These functionalities are implemented using **HTTP Filters**, which are based on servlet concepts in Java.

2. Built-In Filters in Spring Security

Spring Security comes with several pre-configured filters, including:

- **UsernamePasswordAuthenticationFilter:** Handles authentication for username and password.
- **BasicAuthenticationFilter:** Processes HTTP Basic Authentication headers.
- **CsrfFilter:** Protects against Cross-Site Request Forgery attacks.
- **SecurityContextHolderFilter:** Maintains the security context for requests.
- **DefaultLoginPageGeneratingFilter:** Generates a default login page if one is not provided.

Each filter performs a specific responsibility in the authentication and authorization chain.

3. Filter Chain Concept

- **Filter Chain:**
 - Spring Security uses a **chain of servlet filters**, where each filter processes requests and then forwards them to the next filter in the chain.
 - This modular approach allows adding or customizing filters without disrupting the entire authentication flow.
- **Custom Filters:**

- You can create your own custom filters and add them to the chain for specific requirements.
- Example:

```
java

@Component
public class CustomFilter extends OncePerRequestFilter {
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        // Custom logic
        filterChain.doFilter(request, response); // Pass control to the next filter
    }
}
```

4. Debugging Filters

To debug the filter chain and inspect how filters are applied, you can:

1. Use `@EnableWebSecurity(debug = true)` in your security configuration.
 - This enables debugging for the filter chain.
2. Add the following property to `application.properties`:

```
properties

logging.level.org.springframework.security.web.FilterChainProxy=DEBUG
```

- This logs detailed information about the filters and their processing.

5. Internal Filters in Spring Security

Below is a summary of the key filters in the chain:

- **DisableEncodeUrlFilter:** Disables URL encoding.

- **WebAsyncManagerIntegrationFilter**: Integrates Spring Security with asynchronous requests.
 - **CorsFilter**: Enforces Cross-Origin Resource Sharing policies.
 - **LogoutFilter**: Manages logout functionality.
 - **SessionManagementFilter**: Ensures session integrity and management.
 - **ExceptionTranslationFilter**: Translates exceptions into appropriate HTTP responses.
 - **FilterSecurityInterceptor**: Performs the final authorization check.
-

Example Filter Chain

Here's an example of a typical filter chain:

1. A request arrives at the **UsernamePasswordAuthenticationFilter**.
 2. If authenticated, it proceeds to the **CsrfFilter** for CSRF protection.
 3. It then moves to the **SecurityContextHolderFilter**, which ensures the security context is intact.
 4. Finally, the **FilterSecurityInterceptor** checks if the user has the necessary permissions.
-

Key Takeaways

- Filters are crucial for authentication, authorization, and other security-related processing in Spring Security.
- The modular filter chain design allows flexibility and customization.
- Built-in filters cover most common use cases, but custom filters can be added for specific requirements.
- Debugging the filter chain is essential for troubleshooting issues in the security flow.

Let me know if you'd like examples or further clarification on any of the filters!



IMPLEMENTING CUSTOM FILTERS

INSIDE SPRING SECURITY

✓ We can create our own filters by implementing the **Filter** interface from the **jakarta.servlet** package. Post that we need to override the `doFilter()` method to have our own custom logic. This method accepts 3 parameters the `ServletRequest`, `ServletResponse` and `FilterChain`.

- **ServletRequest**—It represents the HTTP request. We use the `ServletRequest` object to retrieve details about the request from the client.
- **ServletResponse**—It represents the HTTP response. We use the `ServletResponse` object to modify the response before sending it back to the client or further along the filter chain.
- **FilterChain**—The filter chain represents a collection of filters with a defined order in which they act. We use the `FilterChain` object to forward the request to the next filter in the chain.

✓ You can add a new filter to the spring security chain either before, after, or at the position of a known one. Each position of the filter is an index (a number), and you might find it also referred to as "the order."

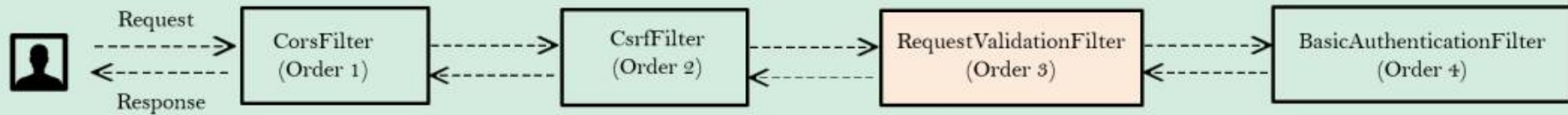
✓ Below are the methods available to configure a custom filter in the spring security flow,

- **`addFilterBefore(filter, class)`** – adds a filter before the position of the specified filter class
- **`addFilterAfter(filter, class)`** – adds a filter after the position of the specified filter class
- **`addFilterAt(filter, class)`** – adds a filter at the location of the specified filter class

ADD FILTER BEFORE IN SPRING SECURITY

**eazy
bytes**

`addFilterBefore(filter, class)` – It will add a filter before the position of the specified filter class.

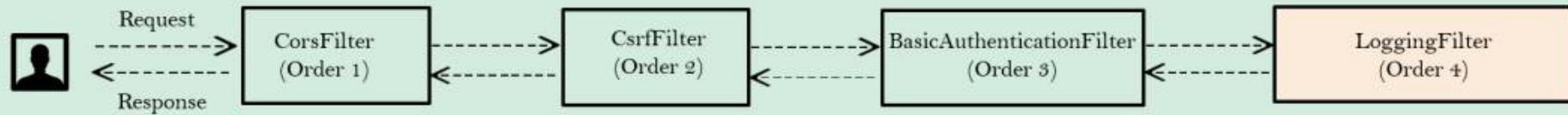


Here we add a filter just before authentication to write our own custom validation where the input email provided should not have the string 'test' inside it.

ADD FILTER AFTER IN SPRING SECURITY

**eazy
bytes**

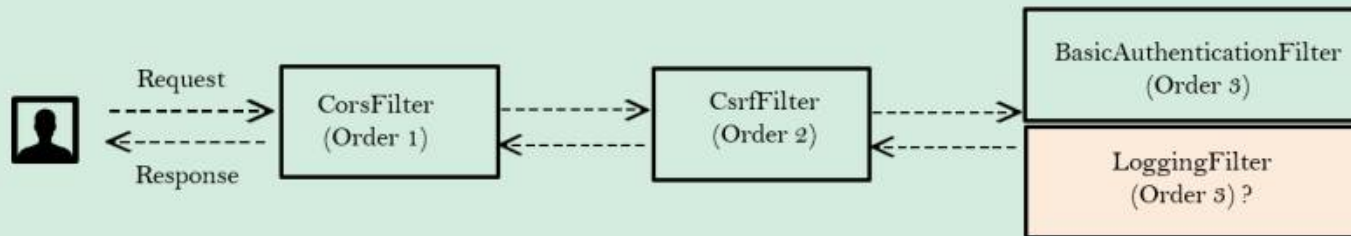
`addFilterAfter(filter, class)` – It will add a filter after the position of the specified filter class



Here we add a filter just after authentication to write a logger about successful authentication and authorities details of the logged in users.

ADD FILTER AT IN SPRING SECURITY

addFilterAt(filter, class) – Adds a filter at the location of the specified filter class. But the order of the execution can't be guaranteed. This will not replace the filters already present at the same order.



Since we will not have control on the order of the filters and it is random in nature we should avoid providing the filters at same order.

Kindly explain the above image with detailed explanation and along with example



Explanation of Implementing Custom Filters in Spring Security

1. Custom Filters Overview

Custom filters in Spring Security are implemented to handle specific requirements not covered by the built-in filters. The steps to implement a custom filter are as follows:

1. Implementing the Filter Interface:

- Use the `jakarta.servlet.Filter` interface.
- Override the `doFilter()` method, which processes the request, response, and forwards it along the filter chain.

2. Key Parameters in `doFilter()`:

- `ServletRequest`: Represents the incoming HTTP request.
- `ServletResponse`: Represents the outgoing HTTP response.
- `FilterChain`: Passes the request/response to the next filter in the chain.

Example:

```
java

@Component
public class CustomFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        // Custom logic here
        chain.doFilter(request, response); // Forward request/response to next filter
    }
}
```

2. Adding Custom Filters to the Filter Chain

Spring Security allows adding custom filters in various positions relative to existing filters:

1. `addFilterBefore(filter, class)`:

- Adds a custom filter **before** a specified filter class in the chain.
- Example:

```
java
```

```
http.addFilterBefore(new CustomFilter(), UsernamePasswordAuthenticationFilter.class);
```

- **Use Case:** Input validation before authentication.

2. `addFilterAfter(filter, class)`:

- Adds a custom filter **after** a specified filter class.
- Example:

```
java
```

```
http.addFilterAfter(new LoggingFilter(), BasicAuthenticationFilter.class);
```

- **Use Case:** Logging user details after authentication.

3. `addFilterAt(filter, class)`:

- Adds a custom filter at the **same position** as the specified filter class.
- The execution order may not be predictable.
- Example:

```
java
```

```
http.addFilterAt(new CustomFilter(), CsrfFilter.class);
```

- **Caution:** Avoid this approach due to potential conflicts.

3. Custom Filter Examples

Example 1: Adding a Filter Before

- **Scenario:** Validate email input before proceeding to authentication.
- **Implementation:**

```
java
```

```
public class RequestValidationFilter extends GenericFilterBean {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        String email = request.getParameter("email");
        if (email != null && email.contains("test")) {
            throw new ServletException("Invalid email input");
        }
        chain.doFilter(request, response);
    }
}

http.addFilterBefore(new RequestValidationFilter(), UsernamePasswordAuthenticationFilter.class);
```

Example 2: Adding a Filter After

- **Scenario:** Log user authentication details after successful login.
- **Implementation:**

```
java
```

```
public class LoggingFilter extends OncePerRequestFilter {
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        Authentication auth = SecurityContextHolder.getContext().getAuthentication();
        if (auth != null && auth.isAuthenticated()) {
            System.out.println("User: " + auth.getName() + " authenticated with roles: " + auth.getAuthorities());
        }
        filterChain.doFilter(request, response);
    }
}

http.addFilterAfter(new LoggingFilter(), BasicAuthenticationFilter.class);
```

Example 3: Adding a Filter at the Same Position

- **Scenario:** Encrypt user data at the same position as an existing filter.

- **Caution:** Execution order may be unpredictable.
- **Implementation:**

```
java

http.addFilterAt(new EncryptionFilter(), CsrfFilter.class);
```

4. Visual Representation of Filter Placement

Adding Filter Before

- Example: Validating input email (Custom filter is placed before `BasicAuthenticationFilter``).

Filter Order:

1. `CorsFilter``
2. `CsrfFilter``
3. `RequestValidationFilter` (Custom)`
4. `BasicAuthenticationFilter``

Adding Filter After

- Example: Logging user details (Custom filter is placed after `BasicAuthenticationFilter``).

Filter Order:

1. `CorsFilter``
2. `CsrfFilter``
3. `BasicAuthenticationFilter``
4. `LoggingFilter` (Custom)`

Adding Filter at the Same Position

- Example: Encrypting user data (Custom filter at the same position as `CsrfFilter``).

Filter Order:

- Execution order between `CustomFilter` and `CsrfFilter` is not guaranteed.

Key Considerations

- Use `addFilterBefore` or `addFilterAfter` for predictable filter order.
- Avoid `addFilterAt` unless absolutely necessary.
- Always call `filterChain.doFilter(request, response)` to ensure the chain continues.
- Test custom filters thoroughly to ensure they integrate seamlessly with the Spring Security flow.

Let me know if you need further details or more examples!



Kindly explain the above image with detailed explanation and along with example



The image describes two important filter classes in Spring Security: `GenericFilterBean` and `OncePerRequestFilter`. These filters play a crucial role in implementing custom logic during the request processing lifecycle. Below is a detailed explanation with examples:

1. GenericFilterBean

- **Overview:**

- `GenericFilterBean` is an abstract class in Spring Security that simplifies the creation of filters.
- It allows you to access initialization parameters and configurations defined in deployment descriptors (e.g., `web.xml` or Spring configuration).

- **Features:**

- Acts as a base class for filters.
- Provides access to Spring's dependency injection and lifecycle methods.
- Simplifies filter creation by handling boilerplate code.

- **Usage:**

- Ideal for implementing filters where initialization parameters and configurations are required.
- Provides access to the `ServletContext` and initialization parameters.

- **Implementation Example:**

```
java

@Component
public class MyGenericFilter extends GenericFilterBean {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        // Custom filter logic
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        System.out.println("GenericFilterBean: Request URI - " + httpRequest.getRequestURI());

        // Continue the filter chain
        chain.doFilter(request, response);
    }
}
```

- **Integration:** Add the filter to the Spring Security filter chain:

```
java

http.addFilterBefore(new MyGenericFilter(), UsernamePasswordAuthenticationFilter.class);
```

2. OncePerRequestFilter

- **Overview:**

- `OncePerRequestFilter` ensures that the filter is executed **only once per request**.
- This is useful when multiple filters in the chain might call the same filter due to request forwarding or including (e.g., with JSPs or dispatchers).

- **Features:**

- Prevents duplicate execution of the filter logic.
- Automatically manages re-entrance of filters for forwarded requests.

- **Usage:**

- Ideal for scenarios where the filter logic must run **exactly once per request**.
- Commonly used for tasks like logging, authentication, or validation.

- **Implementation Example:**

```
java

@Component
public class MyOncePerRequestFilter extends OncePerRequestFilter {
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        // Custom logic
        System.out.println("OncePerRequestFilter: Handling request URI - " + request.getRequestURI());

        // Proceed with the filter chain
        filterChain.doFilter(request, response);
    }
}
```

- **Integration:** Add the filter to the Spring Security filter chain:

```
java

http.addFilterAfter(new MyOncePerRequestFilter(), CsrfFilter.class);
```


Comparison: GenericFilterBean vs OncePerRequestFilter

Aspect	GenericFilterBean	OncePerRequestFilter
Execution	Executes for each dispatch (e.g., include, forward).	Executes only once per request.
Usage	When initialization parameters and servlet context are required.	When logic needs to run exactly once per request.
Ideal For	Basic filtering tasks with initialization parameters.	Logging, authentication, or validation logic.
Boilerplate Handling	Minimal boilerplate handling.	Handles re-entrance of filters automatically.

Key Considerations

- Use `OncePerRequestFilter` for tasks like logging or validation to avoid redundant execution during forwarded requests.
- Use `GenericFilterBean` when you need access to initialization parameters or the `ServletContext`.
- Always ensure `filterChain.doFilter(request, response)` is called to continue the filter chain.

If you need more examples or clarifications, feel free to ask!