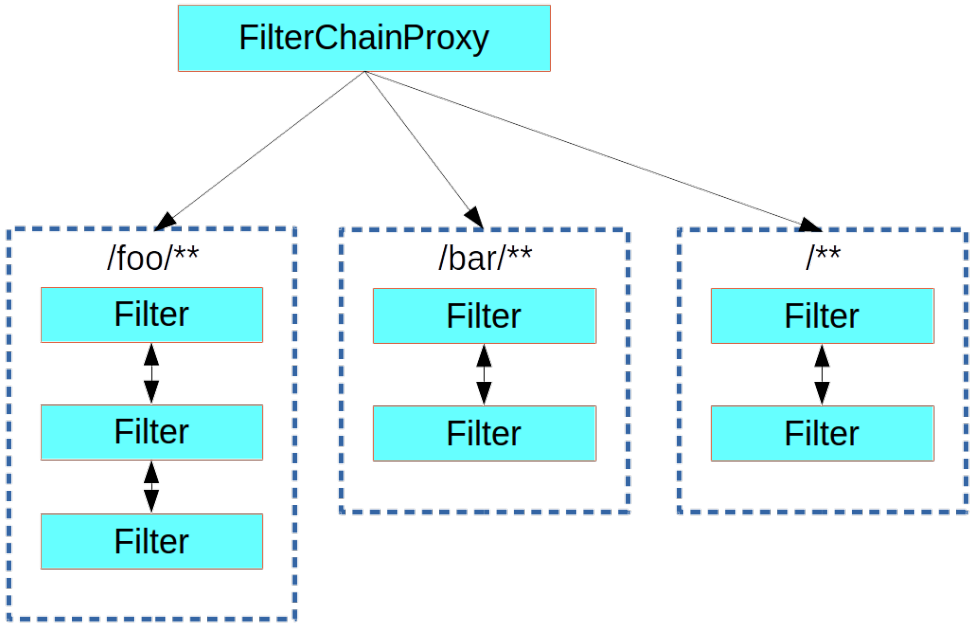
Customizations Outline

- Dispatching Requests to the First Chain That Matches
- Configuration of Filter Chains
- Creating and Customizing Filter Chains - I
- Creating and Customizing Filter Chains – II
- Request Matching for Dispatch and Authorization
- Combining Application Security Rules with Actuator Rules

Dispatching Requests to the First Chain That Matches

- There can be **multiple filter chains** all managed by **Spring Security** in the same top level FilterChainProxy and all are unknown to the container.
- The Spring Security filter contains a **list of filter chains** and ***dispatches a request to the first chain that matches it***.
- The **most important feature** of this dispatch process is that ***only one chain ever handles a request***.



- The **dispatch** happening based on matching the request path
  - **/foo/**** matches before **/****.

Configuration of Filter Chains

- A **vanilla Spring Boot application** with no custom security configuration has a several (call it n) filter chains, where usually **n=6**.
- The first (n-1) chains are there just to **ignore** static resource **patterns**, like /css/** and /images/**, and the error view: /error.
- The last chain matches the **catch-all** path (/**) and is more active, containing logic for authentication, authorization, exception handling, session handling, header writing, and so on.
- There are **a total of** 11 filters in this chain by default.
    - Users **don't have to** concern themselves with which filters are used and when.

- All filters internal to Spring Security are **unknown to the container** is important, especially in a Spring Boot application, where, by default, all @Beans of type Filter are **registered automatically with the container**.
- If you want to add a **custom filter** to the security chain, you need to either
    - **not make** it be a @Bean or
    - wrap it in a FilterRegistrationBean that explicitly disables the container registration.

Creating and Customizing Filter Chains - I

- The **default** fallback filter chain in a Spring Boot application (the one with the /** request matcher) has a **predefined order** of SecurityProperties.BASIC_AUTH_ORDER.
  - Order applied to the SecurityFilterChain that is used to configure **basic authentication** for application endpoints.
- The **actual order** can be interpreted as prioritization, with the first object (with the **lowest order value**) having the **highest priority**.

```
package org.springframework.boot.autoconfigure.security;

public class SecurityProperties {
    . . . // intentionally skipped

    public static final int BASIC_AUTH_ORDER = Ordered.LOWEST_PRECEDENCE - 5;

    public static final int IGNORED_ORDER = Ordered.HIGHEST_PRECEDENCE;

    . . . // intentionally skipped
}
```

- SecurityProperties.IGNORED_ORDER is applied to the WebSecurityCustomizer that ignores standard **static resource paths**.

Creating and Customizing Filter Chains - II

- You can switch it off completely by setting security.basic.enabled = false, or
- You can use it as a fallback and define other rules **with a lower order**.
  - Add a @Bean of type WebSecurityConfigurer and
  - Decorate the class with @Order:

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/match1/**")
        ...;
    }
}
```

- This bean causes **Spring Security** to add a new filter chain and order it before the **fallback**.

Request Matching for Dispatch and Authorization

- A **security filter chain** (or, equivalently, a WebSecurityConfigurerAdapter) has a **request matcher** that is used to decide whether to apply it to an **HTTP request**.
- Once the decision is made to apply a particular filter chain, no others are applied.
- However, within a filter chain, you can have more **fine-grained control of authorization** by setting additional matchers in the HttpSecurity configurer, as follows:

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/match1/**").authorizeRequests()    // this one is a request matcher for the whole filter chain
            .antMatchers("/match1/user").hasRole("USER")     // this one is only to choose the access rule to apply
            .antMatchers("/match1/spam").hasRole("SPAM")
            .anyRequest().isAuthenticated();
    }
}
```

Combining Application Security Rules with Actuator Rules

- If you use the **Spring Boot Actuator** for management endpoints, you probably want them to be **secure**, and, by default, **they are**.
- In fact, as soon as you add the **Actuator** to a secure application, **you get an additional filter chain** that applies only to the actuator endpoints.
  - It is defined with a **request matcher** that matches only actuator endpoints and
  - it has an order of ManagementServerProperties.**BASIC_AUTH_ORDER**, which is 5 fewer than the default SecurityProperties fallback filter, so it is consulted before the fallback.

```
@Configuration
@Order(ManagementServerProperties.BASIC_AUTH_ORDER + 1)
public class ApplicationConfigurerAdapter extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/foo/**")
        . . .;
    }
}
```

Applying Custom Application Security Rules to the Actuator Endpoints:

- Add a filter chain that
  - is ordered earlier than the actuator one and
  - has a request matcher that includes all actuator endpoints.

Default Security Settings for the Actuator Endpoints

- Add your own filter
  - **later than the actuator one**,
  - but **earlier than the fallback**,

    i.e., ManagementServerProperties.**BASIC_AUTH_ORDER** + 1