

# JAVA FULL STACK DEVELOPMENT PROGRAM

Session 23: Spring security

# OUTLINE

- Spring Boot Actuator
- Spring Security

# APPLICATION MONITOR

- Imagine you are going to work on a new Spring application, how can you find all the endpoints it provided?
- Imagine you deployed your Spring application to production environment, when some features fail to work, how can you figure out what is the root cause?
- Both cases bring us an important idea: Application Monitoring. By monitoring our application, we can see all the details including configuration, auto configuration, beans definition and environment setup

# SPRING BOOT

- What if we want to monitor our application when it is in production?
- Actuator — An actuator is a manufacturing term that refers to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change
- Spring Boot Actuator
  - Spring Boot includes a number of additional features to help you monitor and manage your application when you push it to production
  - You can choose to manage and monitor your application by using HTTP endpoints or with JMX(Java Management Extension which I showed you when talking about monitor the JNDI).
  - Auditing, health, and metrics gathering can also be automatically applied to your application
  - Here we will explore the Spring Boot 2 Actuator (Spring boot 1 Actuator is tied to MVC, therefore to the Servlet API)



# SPRING BOOT ACTUATOR

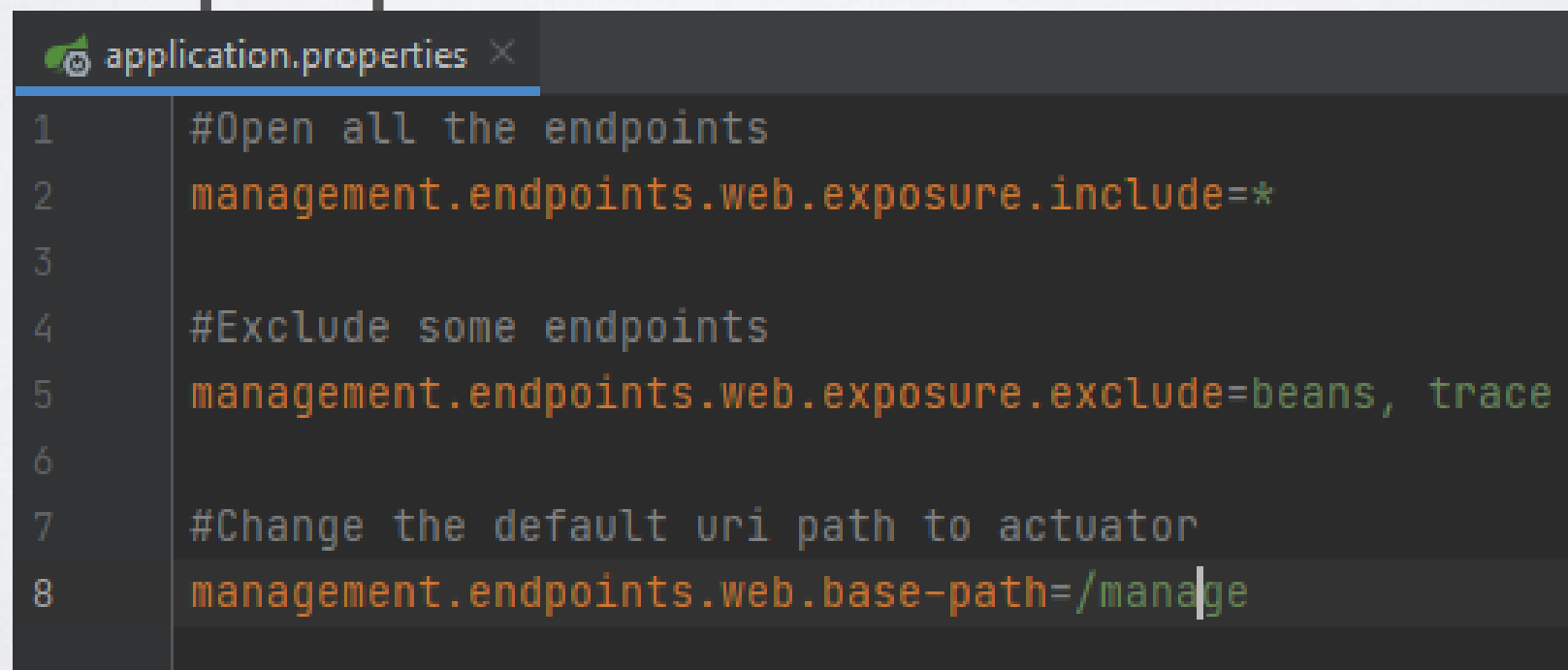
- To get started, we just need to add the *spring-boot-actuator* dependency to our package manager

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>
```

- Once added, we have a list of predefined endpoint (some commonly used are listed here):
  - /health – summarises the health status of our application
  - /info – returns general information. It might be custom data, build information or details about the latest commit
  - /metrics – details metrics of our application. This might include generic metrics as well as custom ones
  - /beans: show all the beans
  - /mappings: show all the URL path

# SPRING BOOT ACTUATOR

- By default actuator only opens two endpoints to user: /actuator/health & /actuator/info
- We can also open other endpoints in application.properties file

A screenshot of a code editor window titled 'application.properties'. The window shows a list of configuration properties for the Spring Boot Actuator. The properties are: 1. '#Open all the endpoints' followed by 'management.endpoints.web.exposure.include=\*'. 2. '#Exclude some endpoints' followed by 'management.endpoints.web.exposure.exclude=beans, trace'. 3. '#Change the default uri path to actuator' followed by 'management.endpoints.web.base-path=/manage'. The text is color-coded: comments are grey, property names are orange, and values are green. A cursor is visible at the end of the last line.

```
1 #Open all the endpoints
2 management.endpoints.web.exposure.include=*
3
4 #Exclude some endpoints
5 management.endpoints.web.exposure.exclude=beans, trace
6
7 #Change the default uri path to actuator
8 management.endpoints.web.base-path=/manage
```

# SPRING BOOT ACTUATOR

- If we enabled the Spring Security, as explained in last session, all endpoint without configuration will by default be secured.
- In order to visit our endpoint without any authentication, we have to exclude in security configuration.
- Spring Boot 2 Actuator now shares the security config with the regular App security rules

```
.authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry  
.antMatchers( ...antPatterns: "/all", "/generalException", "/businessException", "/auth/signin").permitAll()  
.antMatchers( ...antPatterns: "/actuator/**").permitAll()  
.anyRequest().authenticated()  
.and() HttpSecurity  
.apply(new JwtConfig(jwtTokenProvider));
```

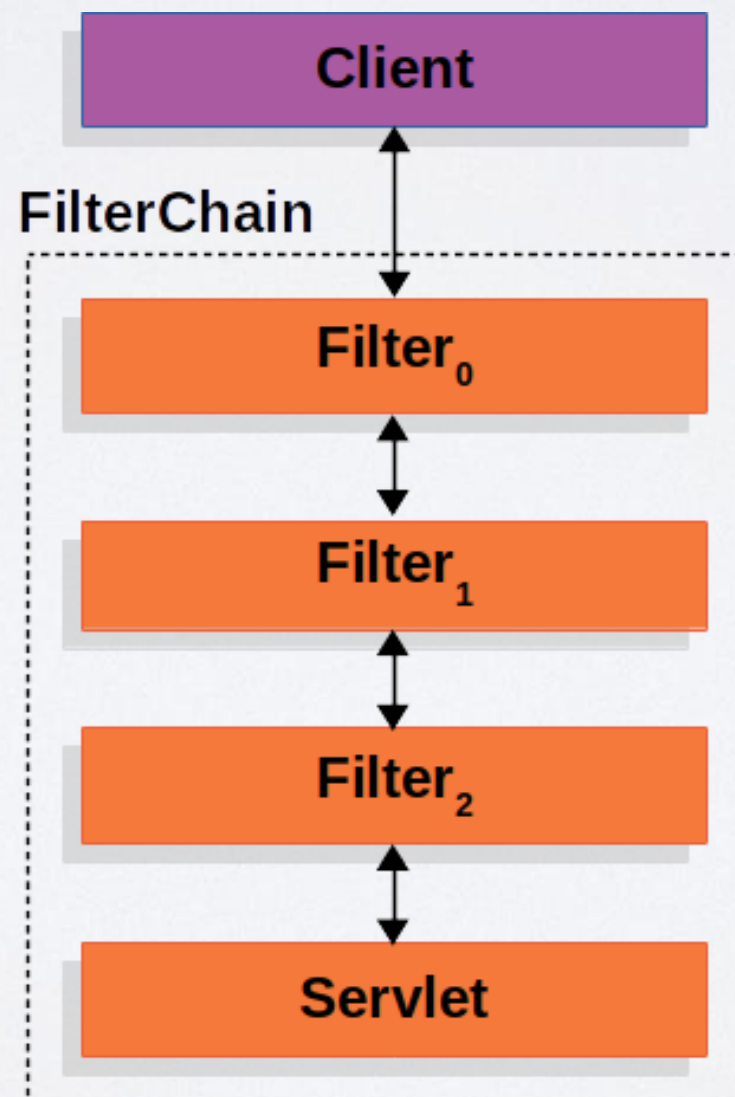
# SPRING SECURITY

- Spring Security is another module provided by Spring team
- 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.
- More concretely, you do not need to use Spring in your Servlet-based application to take advantage of Spring Security.



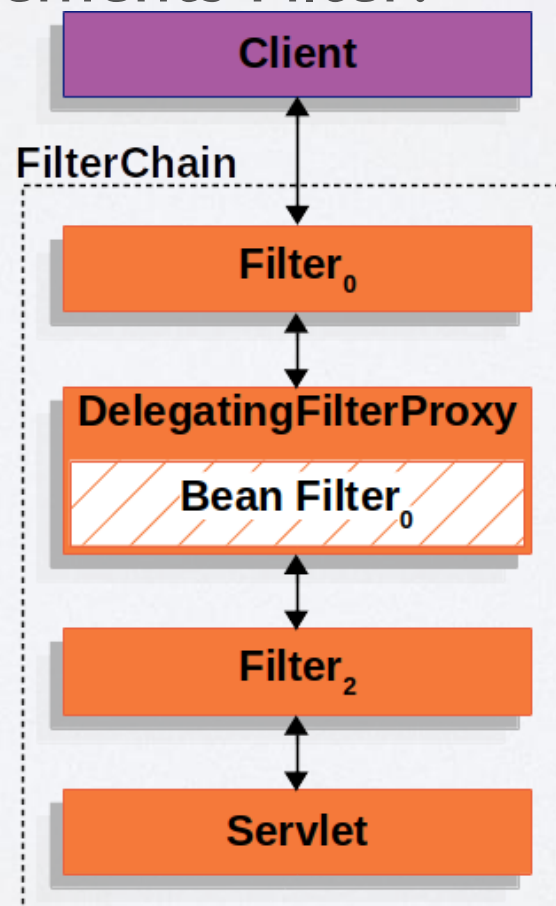
# Filter(s)

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



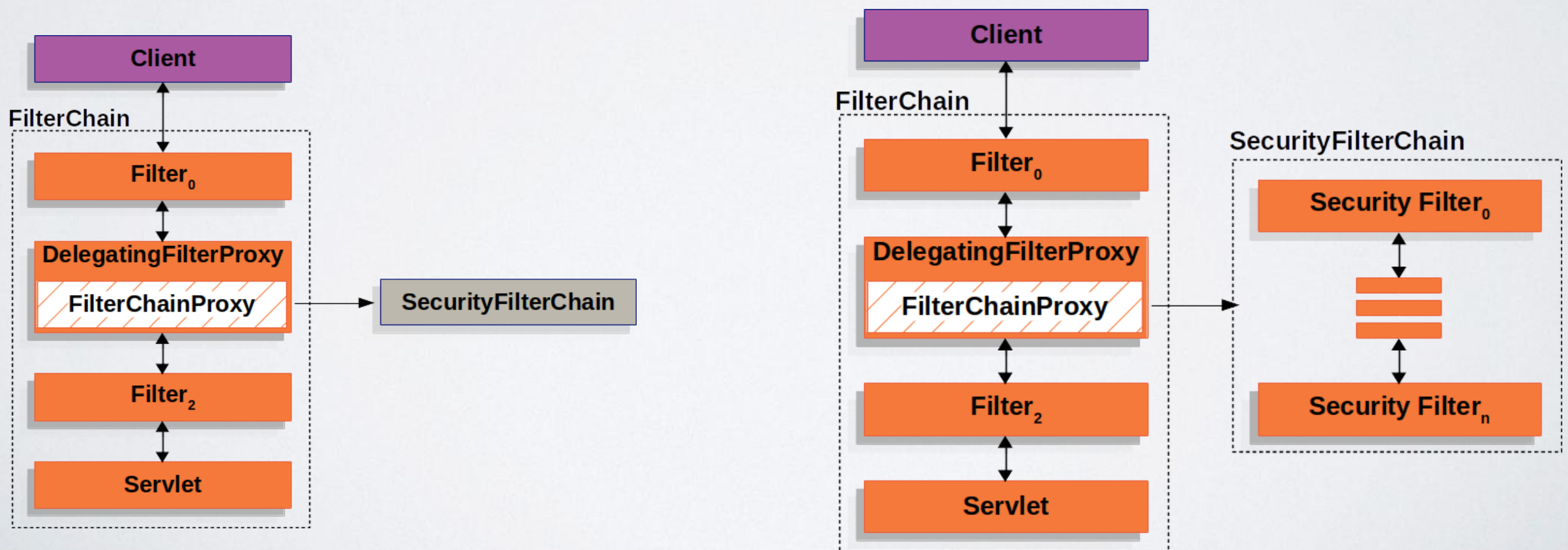
# DelegatingFilterProxy

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



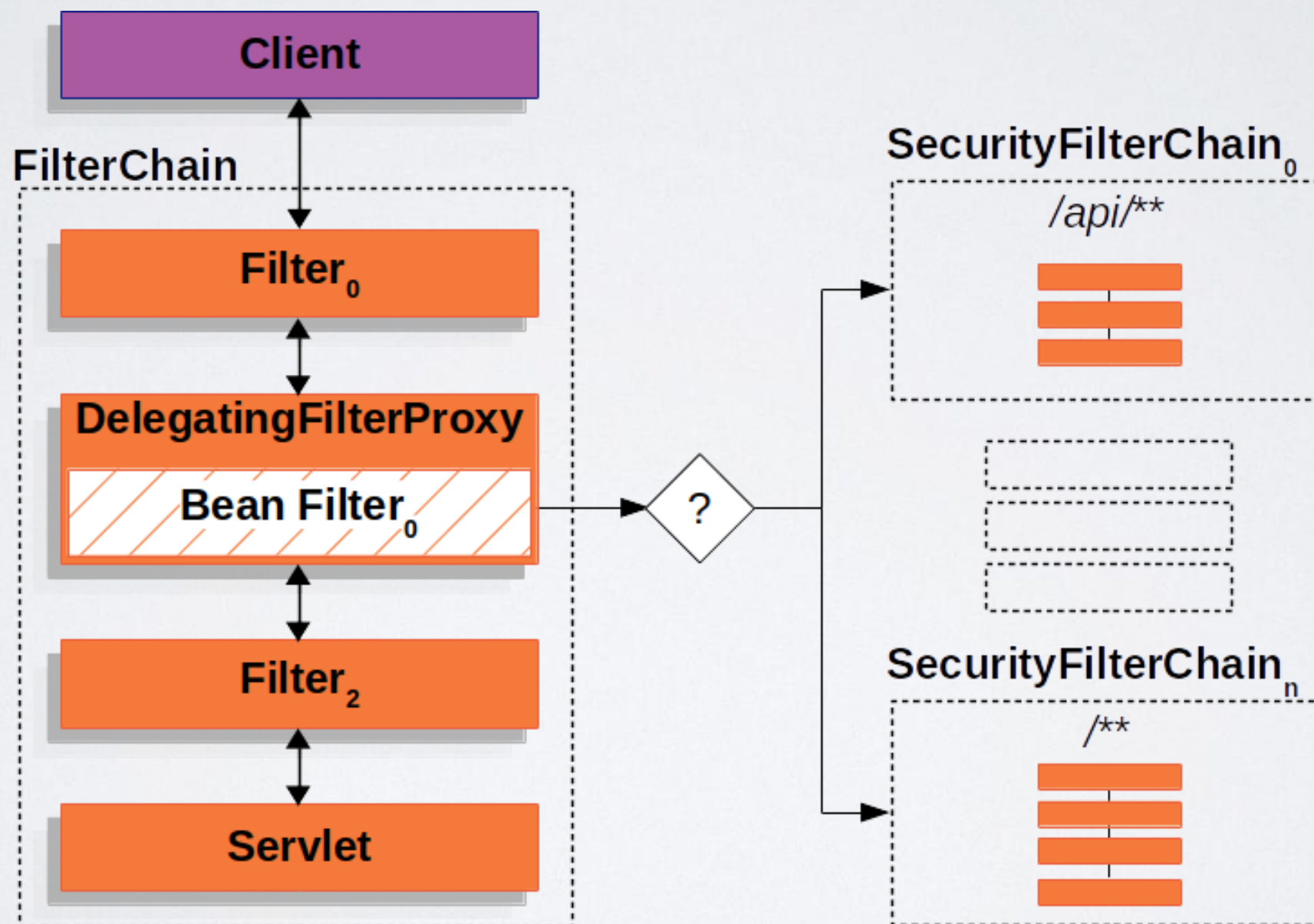
# FilterChainProxy

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





# Multiple SecurityFilterChain



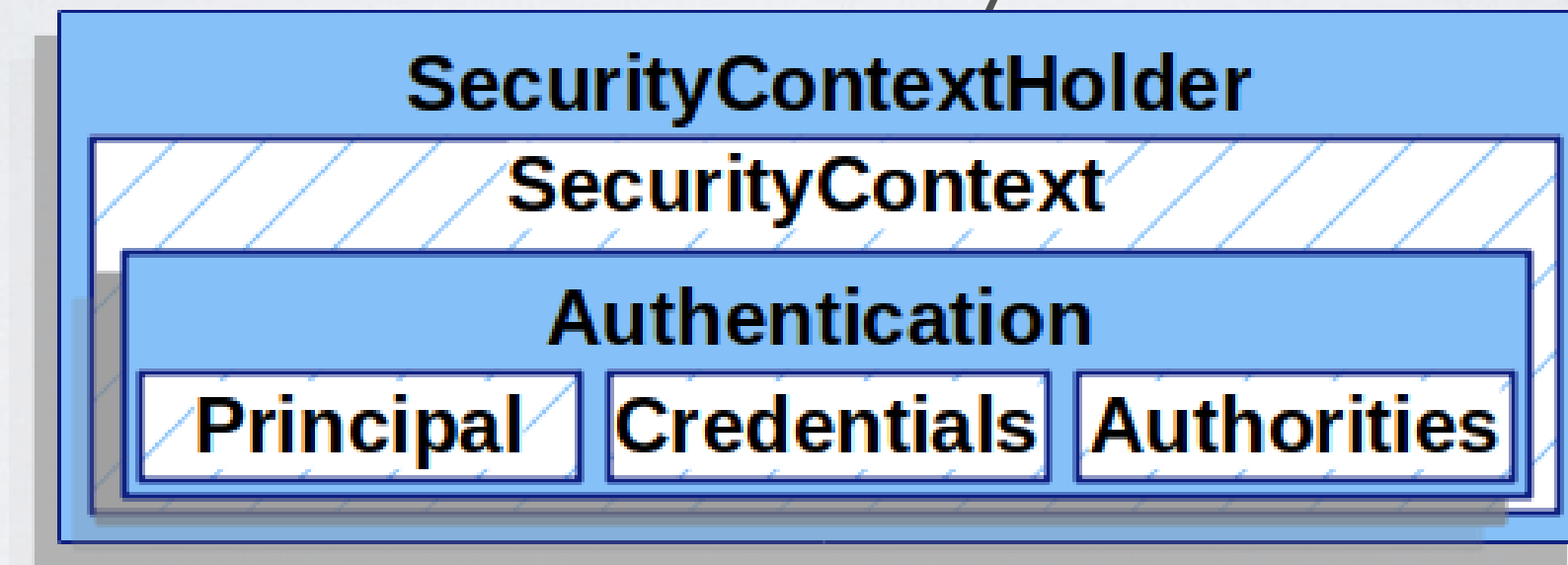


# SPRING SECURITY

- Spring Security **Authentication** Core Components:
  - 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.)
  - AuthenticationManager - the API that defines how Spring Security's Filters perform authentication.

# SecurityContextHolder

At the heart of Spring Security's authentication model is the SecurityContextHolder. It contains the SecurityContext.



SecurityContextHolder — to provide access to the SecurityContext

- SecurityContext — to hold the Authentication and possibly request-specific security information.
- Authentication — to represent the principal in a Spring Security-specific manner.

# SecurityContextHolder

```
SecurityContext context = SecurityContextHolder.createEmptyContext(); 1  
Authentication authentication =  
    new TestingAuthenticationToken("username", "password", "ROLE_USER"); 2  
context.setAuthentication(authentication);  
  
SecurityContextHolder.setContext(context); 3
```

1. We start by creating an empty SecurityContext.
2. Next we create a new Authentication object.
3. Finally, we set the SecurityContext on the SecurityContextHolder. Spring Security will use this information for authorization.

# SecurityContextHolder

If you wish to obtain information about the authenticated principal, you can do so by accessing the SecurityContextHolder.

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



# SecurityContext & Authentication

- **SecurityContext**
  - The SecurityContext is obtained from the SecurityContextHolder. The SecurityContext contains an Authentication object.
- **Authentication**
  - The Authentication 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.
    - Represents the currently authenticated user. The current Authentication can be obtained 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 will be cleared after the user is authenticated to ensure it is not leaked.
    - **authorities** - the GrantedAuthoritys are high level permissions the user is granted. A few examples are roles or scopes.

# GrantedAuthority

- GrantedAuthority(s) are high level permissions the user is granted. A few examples are roles or scopes.
- GrantedAuthority(s) can be obtained 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`. These roles are later on configured for web authorization, method authorization and domain object authorization.

# 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 (i.e. Spring Security's Filters) that invoked the AuthenticationManager.
- If you are not integrating with Spring Security's Filters you can set the SecurityContextHolder directly and are not required to use an AuthenticationManager.



# UserDetails & UserDetailsService

- **UserDetails**
  - UserDetails is returned by the UserDetailsService. The DaoAuthenticationProvider validates the UserDetails and then returns an Authentication that has a principal that is the UserDetails returned by the configured UserDetailsService.
- **UserDetailsService**
  - UserDetailsService is used by DaoAuthenticationProvider for retrieving a username, password, and other attributes for authenticating with a username and password. Spring Security provides in-memory and JDBC implementations of UserDetailsService.



# PasswordEncoder

Spring Security's servlet support stores passwords securely by integrating with PasswordEncoder. Customizing the PasswordEncoder implementation used by Spring Security can be done by exposing a PasswordEncoder Bean.

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
}

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

@Bean
@Override
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}
```

# SPRING SECURITY

- The most fundamental object is `SecurityContextHolder`.
- This is where we store details of the present security context of the application, which includes details of the principal currently using the application

```
Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();

if (principal instanceof UserDetails) {
    String username = ((UserDetails)principal).getUsername();
} else {
    String username = principal.toString();
}
```

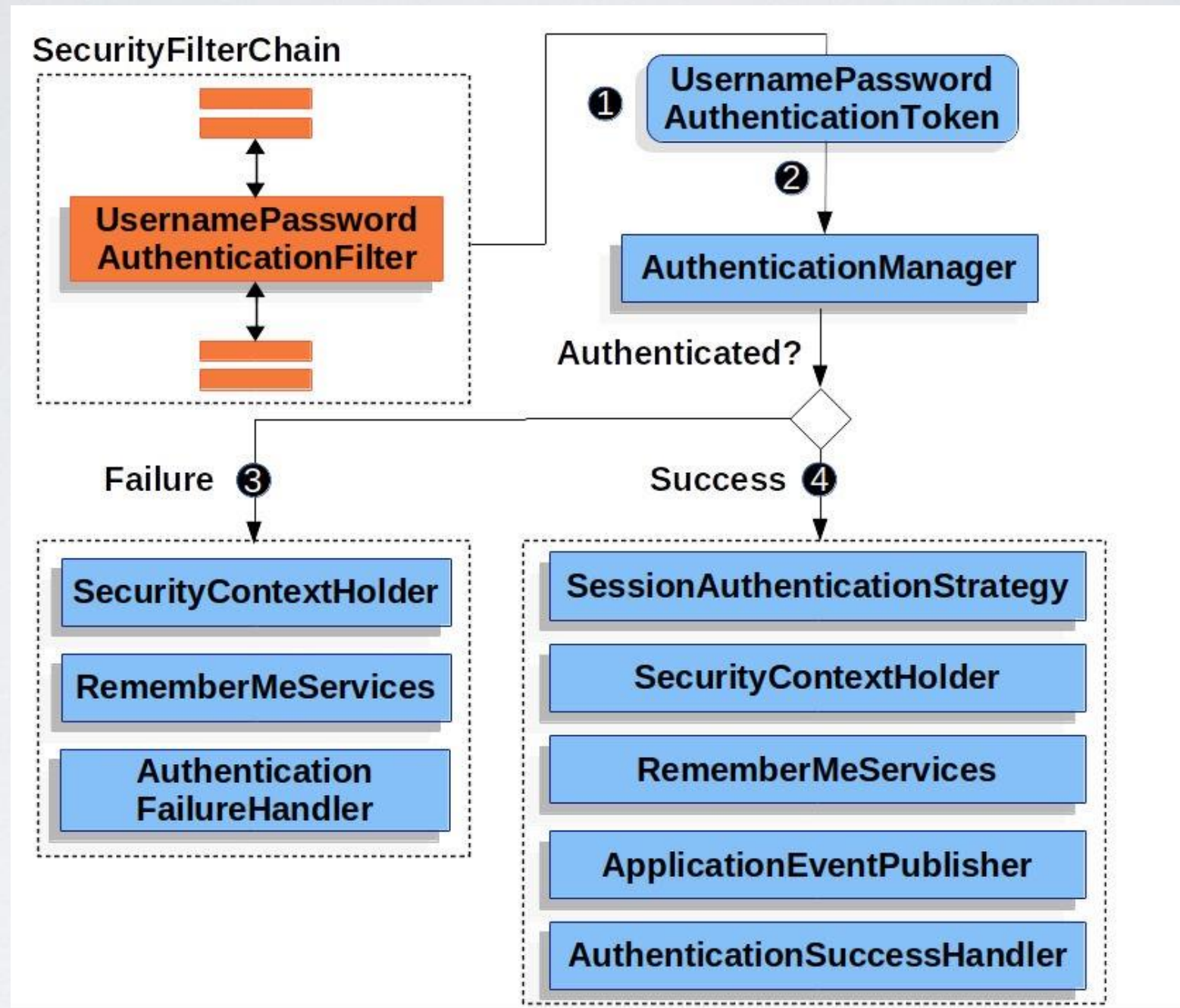
- Inside the `SecurityContextHolder` we store details of the principal currently interacting with the application. Spring Security uses an `Authentication` object to represent this information.
- You won't normally need to create an `Authentication` object yourself, but it is fairly common for users to query the `Authentication` object

# SPRING SECURITY

- The principal is just an Object. Most of the time this can be cast into a UserDetails object.
- UserDetails is a core interface in Spring Security.
- On successful authentication, UserDetails is used to build the Authentication object that is stored in the SecurityContextHolder
- UserDetailsService is purely a DAO for user data and performs no other function other than to supply that data to other components within the framework
  - UserDetailsService is only responsible for reading data, but not the authentication process.



# Workflow



Notice: Before **UsernamePasswordAuthenticationFilter**, we add our customized JWT filter



# SPRING SECURITY

## EXPRESSIONS

- Ant pattern — Apache Ant provide a directory tree to represent the path
- There are some rules:
  - A single star (\*) matches zero or more characters *within a path name*
  - A double star (\*\*) matches zero or more characters *across directory levels*
  - A question mark (?) matches exactly one character within a path name

# SPRING SECURITY EXPRESSIONS

- For example, we have following directory structure

1. bar.txt
2. src/bar.c
3. src/baz.c
4. src/test/bartest.c

- For following Patterns, what files do they match?

- \*.c
- src/\*.c
- \*/\*.c
- \*\*/\*.c
- \*\*/bar.\*

# SPRING SECURITY EXPRESSIONS

- Web Security Expression
  - Hasrole, Hasanyrole
  - Hasauthority, Hasanyauthority
  - Permitall, Denyall
  - Isanonymous, Isrememberme, Isauthenticated, Isfullyauthenticated
  - Principal, Authentication
  - hasPermission

**ANY QUESTIONS?**