**Spring Security:-**

The delegating filter proxy is a generic bean that provides a link between web.xml and application-Context.xml.

uses filters to implement several security related cross-cutting concerns like authentication and authorization.

filters need to be declared in the web.xml so that the Servlet container can invoke them before passing the request to the actual Servlet class.  
  
 **there are some rules and restrictions which you need to follow like**  
  
1) You must declare delegating filter proxy to your web.xml as a filter.  
  
2) The target bean must implement the javax.servlet.Filter interface.  
  
3) The target bean must have the same name as that in the filter-name element.  
  
4) You can also specify a "targetBeanName" filter init-param in web.xml to specify the name of the target bean in the Spring application context.

**. Who manages the life-cycle of filter bean in Spring?**  
As explained in the previous example, by default **Spring container** manages the life-cycle of filter beans in Spring i.e. the beans which implements Filter interface and handle request delegated by delegating Spring proxy, but you can ask Servlet container to manage their life-cycle by declaring "targetFilterLifecycle" filter init-param as "true" on web.xml while declaring the DelegatingFilterProxy filter as shown below:

<filter>

<filter-name>springSecurityFilterChain</filter-name>

<filter-class>

org.springframework.web.filter.DelegatingFilterProxy

</filter-class>

</filter>

<filter-mapping>

<filter-name>springSecurityFilterChain</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

**What is the security filter chain in Spring Security?**  
The Spring Security framework uses a chain of filters to apply **various security concerns** like **intercepting the request, detecting (absence of) authentication, redirecting to the authentication entry point, or pass the request to authorization service, and eventually let the request either hit the servlet or throw a security exception (unauthenticated or unauthorized).**

**What are some predefined filters used by Spring Security? What are their functions and in which order they occurred?**  
The Spring security filter chain is a very complex and flexible chain of filters. These filter access services such as UserDetailsService and AuthenticationManager to perform their task. Their orders are also important as you may want to check for authentication before authorization.  
  
Here are some of the important filters from Spring's security filter chain, in the order they occur in:  
  
**SecurityContextPersistenceFilter -**This filter restores Authentication from the JSESSIONID cookie.  
  
**UsernamePasswordAuthenticationFilter -**This filter performs authentication.  
  
**ExceptionTranslationFilter -**This filter catch security exceptions from FilterSecurityInterceptor.  
  
**FilterSecurityInterceptor -**This filter may throw authentication and authorization exceptions.  
  
  
 **Can you add custom filters in Spring security's filter chain?**  
Yes, you can add or replace individual filters with your own logic in Spring's security filter chain. Even though Spring Security provides a number of filters by default, and most of the time, these are enough. You may need to implement new functionality depending upon your project's requirement and this can be done by creating a new filter to use in the chain.

**How to implement a custom filter in Spring Security?**  
You can implement a custom filter in Spring security by implementing the org.springframework.web.filter.GenericFilterBean class. The GenericFilterBean is a simple javax.servlet.Filter implementation which is Spring aware. You can override doFilter(ServletRequest req, ServletResponse res, FilterChain chain) to implement your own logic.

**What are authentication and authorization? Which must come first?**  
Authentication is a process to verify that the user is the one who he claims to be. It is generally implemented using a username and password. If a user enters the correct username and password then authentication is successful, otherwise, authentication failed.   
  
Authorization provides access control. For example, only the admin can see some pages in a web application. To implement that, the admin must have some admin-related permissions or roles.   
  
If a user becomes admin then those permissions are added to this profile. If you have access to a page it means you are authorized to that page or resources. Obviously, authorization comes after authentication because access can only be provided to genuine users. 

**1. Why do you need the intercept-url?**  
The intercept-url is needed to secure URLs in your Java web application using Spring security. It also defines some sort of authorization, I mean roll or access a user needs to see a page or URL. Most of the web applications using Spring Security only has a couple of **intercept-urls** because they only have very basic security requirements.

You need to have unauthenticated or anonymous access to the login and login-error screens and usually some aspect of the public site, which can be intercepted in a few URL patterns. Then there's often an admin section for admin stuff like creating roles, users, or permissions, and then everything else is ROLE\_USER.  
  
Here is one of the examples of **basic Spring security using intercept URL:**

<http realm="Contacts Realm" use-expressions="false">

<intercept-url pattern="/index.jsp" access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>

<intercept-url pattern="/login.jsp\*" access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>

<intercept-url pattern="/admin/\*" access="ROLE\_ADMIN"/>

<intercept-url pattern="/trade/\*" access="ROLE\_TRADER"/>

<intercept-url pattern="/\*\*" access="ROLE\_USER,ROLE\_ADMIN,ROLE\_TRADER"/>

<http-basic/>

</http>

You can see that index.jsp and admin.jsp is allowed to be accessed without authentication. Anything which has admin in URL required ROLE\_ADMIN access and any URL with trade in it requires ROLE\_TRADER access.

Spring also allows expression-based access control from Spring 3.0 but it is not mandatory, though it gives you more power to implement complex access mechanisms. If you want to learn more about authentication and authorization on spring-security then you can also checkout [**OAuth 2.0 in the Spring Boot Applications**](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Foauth2-in-spring-boot-applications%2F) course on Udemy.

**What do @Secured and @RolesAllowed do? What is the difference between them?**  
Th @Secured annotation is used to define a list of security configuration attributes for business methods. You can specify the security requirements[roles/permission etc] on a method using the @Secured annotation and then only the user with those roles/permissions can invoke that method.

If anyone tries to invoke a method and does not possess the required roles/permissions, an AccessDeniedException will be thrown.

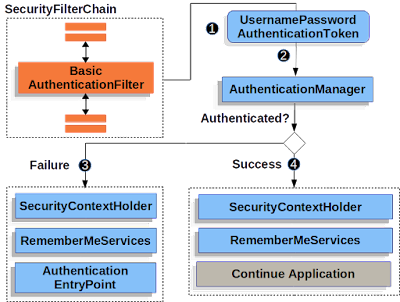
here is a simple example of @Secured annotation: 

@Secured({"ROLE\_ADMIN"})

public String showTrades() {

return "secure/trades";

}

[](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Foauth2-in-spring-boot-applications%2F)

The @Secured is coming from previous versions of Spring. It has a limitation in that it does not support Spring EL expressions.   
  
It's better to replace @Secured annotation with @PreAuthorize annotation which supports Spring EL. For example, the above code can be written using @PreAuthorize as follows: 

@PreAuthorize("hasRole('ROLEADMIN')")

public String showTrades() {

return "secure/trades";

}

On the other hand, @RolesAllowed is a [JSR 250 annotation](https://jcp.org/en/jsr/detail?id=250), which specifies the security roles permitted to access a method or a couple of methods in an application.

If you are not familiar with JSR 250, it's the objective to define common annotations for the Java platform, and if you want to learn more about essential Spring security annotations then you can pick a good course mention in these [best spring security courses](https://medium.com/javarevisited/top-10-courses-to-learn-spring-security-and-oauth2-with-spring-boot-for-java-developers-8f0222d6066d?source=---------5-----------------------) to start with.

**12. Why do you need method security?**  
Security is hard, you often need multiple levels of security to improve the chances to block circumvention attempts. Since method level security is directly coded inside the class, after the AOP augmentation, when you call the method, you'll always call the security check before.  
  
Method level security is useful for two main reasons:

1. It provides  another layer of security (in addition to other layers)  
  
2. In cases where it's more convenient or logical to have permissions at the method level consider a case where different users can perform the same "direct" actions (so client security isn't relevant). but in some cases, their action may trigger behavior you wish to limit - in this case, method level security may be a relevant solution.

### Spring Security Password Encoding questions

Now, let's see some Spring security interview questions on password security and protection related topics like hashing and salting, which is also important for Java developers.   
  
  
**23. Does Spring Security support password hashing? What is salting?**  
One of the common problems of storing passwords on databases is their security. You just can't store passwords as plain text into your database because then anyone who has access to the [database](https://javarevisited.blogspot.com/2018/05/top-5-sql-and-database-courses-to-learn-online.html)would have access to the password of every user. To solve this problem, encrypted passwords are stored in a database and this is known as password hashing.  
  
In cryptography, a salt is random data that is combined with a password before password hashing. This makes a dictionary attack more difficult. This process is known as salting. The hashed version of the password is then stored in a database along with salt.  
  
Btw, some hashing algorithms are not suitable for password hashing, if salt is too small or predictable it's possible to recover passwords by matching random words with salt then comparing the hashed version of output with the data stored in the database.  
  
Yes, [Spring Security](https://javarevisited.blogspot.com/2017/05/how-to-enable-spring-security-in-java-web-application.html) includes password hashing out of the box. Since version 3.1, Spring Security automatically takes care of salting too. You can use PasswordEncoder implementation to implement password hashing in Spring security.  
  
  
**24. What is PasswordEncoder?**  
The PasswordEncoder is an interface in Spring security that provides password encoding or password hashing. It has two methods encode() to encode the raw password and matches() to verify the encoded password obtained from the database matches the submitted raw password after it too is encoded using the same salt and same hashing algorithm.

**25. What are some implementations of PasswordEncoder in Spring Security?**  
Spring security provides several implementations based upon different hashing algorithms, which you can use in your application. The two important implementations of the new PasswordEncoder interface are BCryptPasswordEncoder and the confusingly named StandardPasswordEncoder based on SHA-256. The BCrypt implementation is the recommended one. There is also a [NoOpPasswordEncoder](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/NoOpPasswordEncoder.html" \t "_blank) which does no encoding. It's intended for unit testing only.

Cros origin

# **Enabling Cross Origin Requests for a RESTful Web Service**

The @RequestMapping annotation ensures that HTTP requests to /greeting are mapped to the greeting() method.

@CrossOrigin(origins = "http://localhost:8080")

You can also add the @CrossOrigin annotation at the controller class level as well, to enable CORS on all handler methods of this class.

### Global CORS configuration

In addition (or as an alternative) to fine-grained annotation-based configuration, you can define some global CORS configuration as well. This is similar to using a Filter but can be declared within Spring MVC and combined with fine-grained @CrossOrigin configuration. By default, all origins and GET, HEAD, and POST methods are allowed.

public WebMvcConfigurer corsConfigurer() {

return new WebMvcConfigurer() {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/greeting-javaconfig").allowedOrigins("http://localhost:8080");

}

};

}COPY

You can easily change any properties (such as allowedOrigins in the example), as well as apply this CORS configuration to a specific path pattern.

@SpringBootApplication is a convenience annotation that adds all of the following:

* @Configuration: Tags the class as a source of bean definitions for the application context.
* @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.
* @ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application. Did you notice that there was not a single line of XML? There is no web.xml file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

# Token-Based Authentication System Flow

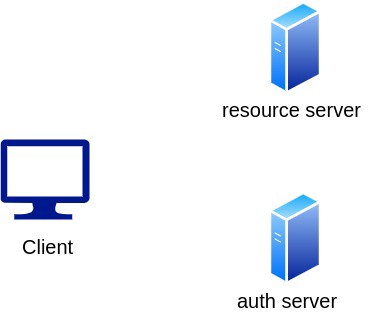
# What is token-based authentication?

*The general concept behind a token-based authentication system is simple. Allow users to enter their username and password in order to obtain a token which allows them to fetch a specific resource — without using their username and password. Once their token has been obtained, the user can offer the token — which offers access to a specific resource for a time period — to the remote site.*

[**JSON Web Token**](https://jwt.io/)**(JWT)**to helps the system authenticate client and user. You can read the specification of JWT [**here**](https://tools.ietf.org/html/rfc7519).

# Use Case

We want to create a system that has a dedicated authentication service, so if we have many services, we won’t have to store client credentials on each service.



use case illustration

The authentication server will generate access token and refresh token in order to replace the user’s credentials to be authenticated on another server.