**SPRING SECURITY**

**What are authentication and authorization? Which must come first?**

**Authentication** is the process of verifying the validity of the principal’s credentials. Answers question of: Who are you?

Authorization is the process of making a decision whether an authenticated user is allowed to perform a certain action within the application. Answers question of: What are you allowed to do? It can be happen at both:

1) Web request level

2) Method invocation level

Authentication is the first step of authorization so always comes first.

**Spring Security supports authentications:** Basic, Form, OAuth, X.509, Cookies, Single-Sign-On

**How credentials are stored:** LDAP, RDBMS, properties files, DAOs, Beans

**Most common user roles:** ADMIN, MEMBER, GUEST

**Is security a cross cutting concern? How is it implemented internally?**

The cross-cutting concern is a concern which is applicable throughout the application and it affects the entire application. For example: logging, security and transactions.

Yes, security is a cross-cutting concern.

Spring Security tackles security from two angles:

1) To secure web requests and restrict access at the URL level, Spring Security is based entirely on standard servlet ﬁlters.

2) It doesn’t use servlets or any other servlet-based frameworks (such as Spring MVC) internally, so it has no strong links to any particular web technology.

3) It deals in HttpServletRequest s and HttpServletResponse s and doesn’t care whether the requests come from a browser, a web service client, or AJAX.

4) Spring Security’s web infrastructure should only be used by delegating to an instance of FilterChainProxy .

5) Spring Security can also secure method invocations using Spring AOP, proxying objects and applying advice to ensure that the user has the proper authority to invoke secured methods.

**Spring Security Configuration**

1) Declare the security filter for the application

2) Define the Spring Security context

3) Configure authentication and authorization

**Spring Security is implemented for --**

1) Spring Security WebMvc

2) Securing RESTful-WS

3) Spring Security in Sprint Boot

If library spring-boot-starter-security is in the classpath, Spring Boot automatically secures all HTTP endpoints with basic authentication. This will add the spring-security-core, spring-security-config, and springsecurity-web dependencies to your project.

**What does the \*\* pattern in an antMatcher or mvcMatcher do?**

\*\* pattern in an antMatcher or mvcMatcher matches zero or more 'directories' in a path.

antMatcher(String antPattern) - Allows configuring the HttpSecurity to only be invoked when matching the provided Ant-style pattern.

For example - /admin/\*\* matches any path starting with /admin.

mvcMatcher(String mvcPattern) - Allows configuring the HttpSecurity to only be invoked when matching the provided Spring MVC pattern.

***For Example -***

antMatchers("/secured") matches only the exact /secured URL

mvcMatchers("/secured") matches /secured as well as /secured/, /secured.html, /secured.xyz

Note that the matchers are always evaluated in the order they are defined. Most specific patterns must come first and most general last. Therefore, the following is invalid because the first matcher matches every request and will never get to the second mapping:

http.authorizeRequests().antMatchers("/\*\*").hasRole("USER").antMatchers("/admin/\*\*").hasRole("ADMIN")

**Why is the usage of mvcMatcher recommended over antMatcher?**

1) Generally mvcMatcher is more secure than an antMatcher.

2) MvcMatcher() uses Spring MVC’s HandlerMappingIntrospector to match the path and extract variables.

3) They both implement RequestMatcher interface.

4) MvcMatcher can also restrict the URLs by HTTP method.

**Does Spring Security support password hashing? What is salting?**

Spring Security uses PasswordEncoder for encoding passwords. This interface has a Md5PasswordEncoder that allows for obtaining hashes of the password - that will be persisted.

Salting is appending a random string to the hash to prevent hackers from matching with a hash from the standard dictionary of hashes.

Spring Security uses PasswordEncoder for encoding passwords. This interface has a Md5PasswordEncoder that allows for obtaining hashes of the password – that will be persisted. The problem is that there are “dictionaries” of hashes available on the Internet and some hacker may just match the hash with a record from those dictionaries and gain unauthorized (from system’s point of view authorized) access. To avoid that you can add some “salt” to the pass before it is hashed. Perfectly that salt (which is some appended string) is some random value – a simpler implementation is to use the user id.

There is a implementation of PasswordEncoder – BCryptPasswordEncoder that generates the salt automatically and thus you don’t have to bother about this.

**Why do you need method security? What type of object is typically secured at the method level (think of its purpose not its Java type) ?**

The abstract design of any application with Method level security keeps front-end independent (or loosely coupled) from back-end.

Due to this disconnect between each other the back-end security framework can't just assume that web level security is implemented flawlessly and hence it becomes important to implement method level security on the applications.

If we secure only the web layer there may be a way to access service layer in case we expose some REST endpoints. That’s why usually services are secured at method level.

To apply security to lower layers of an application, Spring Security uses AOP. The respective bean is wrapped in a proxy that before calling the target method, first checks the credentials of the user and calls the method only if the user is authorized to call it.

When securedEnabled is true, a pointcut is created such that the Spring Security aspects will wrap bean methods that are annotated with @Secured.

Typically methods in service classes (@Service annotation) are secured at method level.

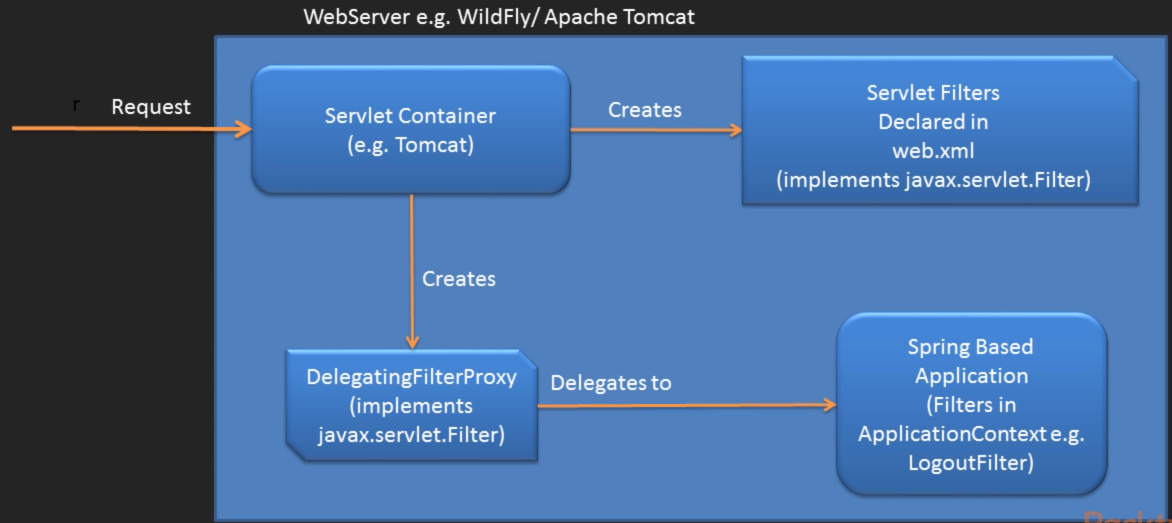
**What is Spring Security?**

Spring security is one of the most important modules of the Spring framework. It enables the developers to integrate the security features easily and in a managed way. Spring Security is a separate module of the Spring framework that focuses on providing authentication and authorization methods in Java applications. It also takes care of most of the common security vulnerabilities such as CSRF attacks.

To use Spring Security in web applications, you can get started with a simple annotation: ***@EnableWebSecurity.***

* Security Framework for Java EE based applications.
* Spring Security can be used even without Spring Framework as an underlying technology for developing an application. However, it is advisable to use Spring security with Spring Framework.
* Spring Security utilizes power of - Spring Dependency Injection.
* Spring security has power of Authentication and Authorization. Error prone boilerplate code like session management is taken care by Spring security itself.
* Spring security can integrate with wide range of APIs - LDAP, Single sign-on, Kerberos, Mule ESB and so on.

**What is DelegatingFilterProxy ?**



Spring security infrastructure is based on Servlet Filters. A Servlet filter is an object that can intercept HTTP requests targeted at your web application. In order to create a servlet filter you must implement the javax.servlet. Filter interface and register in web.xml. The servlet container(webserver) is responsible for managing life cycle of servlets. Servlet filters can be used for login ,logout, authentication etc.

DelegatingFilterProxy is also implementation of the javax.servlet.Filter interface provided by Spring Security framework. DelegatingFilterProxy is kind of bridge between servlet container and filters in ApplicationContext (e.g. LogoutFilter). In Spring Security, the filter classes are also Spring beans defined in the application context and thus able to take advantage of Spring’s rich dependency-injection facilities and lifecycle interfaces.

**What is Spring Security Filter Chain (FilterChainProxy)?**

In Spring Security you have a lot of filters for web application and these filters are Spring Beans. Each Spring security filter bean that requires in your application you have to declare in your application context file and as we know that filters would be applied to the application only when they would be declared in web.xml. Now DelegatingFilterProxy comes into picture for delegating the request to filter which declared into application context file by adding a corresponding DelegatingFilterProxy entry to web.xml for each filter and we have to make sure about ordered, it should be defined correctly, but this would be cumbersome and would clutter up the web.xml file quickly if you have a lot of filters. FilterChainProxy lets us add a single entry to web.xml and deal entirely with the application context file for managing our web security beans.

DelegatingFilterProxy delegates to filters in Spring ApplicationContext using FilterChainProxy. FilterChainProxy will take care of all the filters created in ApplicationContext.

The springSecurityFilterChain is a mandatory name that refers to a bean with the same name in the Spring root application context.

This chain of filters has the following key responsibilities:

1) Driving authentication

2) Enforcing authorization

3) Managing logout

4) Maintaining SecurityContext in HttpSession

**A simple security configuration**

Any bean in the Spring application context that implements WebSecurityConfigurer can contribute to Spring Security configuration, but it’s often most convenient for the configuration class to extend WebSecurityConfigurerAdapter.

@Configuration

@EnableWebSecurity //enables web security, use @EnableWebMvcSecurity if applicable

public class SecurityConfig extends WebSecurityConfigurerAdapter {

}

**Overriding WebSecurityConfigurerAdapter’s configure() methods --**

1) Configure(WebSecurity) Override to configure Spring Security’s filter chain.

2) Configure(HttpSecurity) Override to configure how requests are secured by interceptors.

3) Configure(AuthenticationManagerBuilder) Override to configure user-details services.

------------------------------------------------------------------------------------------------------------------------------------------

**How DelegatingFilterProxy and FilterChainProxy are created in Java configuration?**

In Java config, web.xml is not there. DelegatingFilterProxy and FilterChainProxy are created just by extending AbstractSecurityWebApplicationInitializer. In other words, by just extending AbstractSecurityWebApplicationInitializer, Spring security filter chain is created.

**How to create a custom filter chain ?**

If you want to create a custom filter chain and register it before Spring Security Filter chain, override beforeSpringSecurityFilterChain method of class AbstractSecurityWebApplicationInitializer.

If you want to create a custom filter chain and register it after Spring Security Filter chain, override afterSpringSecurityFilterChain method of class AbstractSecurityWebApplicationInitializer.

To create a custom filter chain, extend class FilterChainProxy

**What is concept of Filter Ordering?**

Default ordering of Filters ---

1. ChannelProcessingFilter
2. SecurityContextPersistenceFilter
3. ConcurrentSessionFilter
4. AuthenticationFilter - UsernamePasswordAuthenticationFilter, BasicAuthenticationFilter, CasAuthenticationFilter
5. SecurityContextHolderAwareRequestFilter
6. JaasAPIIntegrationFilter
7. RememberMeAuthenticationFilter
8. AnonymousAuthenticationFilter
9. ExceptionTranslationFilter
10. FilterSecurityInterceptor

**What are Core Security Filters ?**

Core security Filters are mandatory filters which are always executed in a Spring Security application. Below are Core Security Filters in Spring ---

1. SecurityContextPersistenceFilter
2. UsernamePasswordAuthenticationFilter
3. ExceptionTranslationFilter
4. FilterSecurityInterceptor

**What is ChannelProcessingFilter ?**

ChannelProcessingFilter is for channel security because it might need to redirect to a different protocol(http/https). The most common usage is to ensure that a request takes place over HTTPS, where the ChannelDecisionManagerImpl is configured with a SecureChannelProcessor and an InsecureChannelProcessor.

**What is SecurityContextPersistenceFilter ?**

SecurityContextPersistenceFilter is to setup and manage SecurityContext in the SecurityContextHolder. This filter is executed only once per request.

A SecurityContext can be set up in the SecurityContextHolder at the beginning of a web request, and any changes to the SecurityContext can be copied to the HttpSession when the web request ends (ready for use with the next web request).

It populates the SecurityContextHolder with information obtained from the configured SecurityContextRepository prior to the request and stores it back in the repository once the request has completed and clearing the context holder.

This filter MUST be executed BEFORE any authentication processing mechanisms. Authentication processing mechanisms (e.g. BASIC, CAS processing filters etc) expect the SecurityContextHolder to contain a valid SecurityContext by the time they execute.

**What is ConcurrentSessionFilter ?**

ConcurrentSessionFilter performs two functions. First, it calls SessionRegistry.refreshLastRequest(String) for each request so that registered sessions always have a correct "last update" date/time. Second, it retrieves a SessionInformation from the SessionRegistry for each request and checks if the session has been marked as expired. If it has been marked as expired, the configured logout handlers will be called (as happens with LogoutFilter), typically to invalidate the session.

**What is AbstractAuthenticationProcessingFilter ?**

AbstractAuthenticationProcessingFilter is an authentication processing mechanism i.e. an abstract processor of browser-based HTTP-based authentication requests. UsernamePasswordAuthenticationFilter, CasAuthenticationFilter, BasicAuthenticationFilter etc. extend AbstractAuthenticationProcessingFilter class so that the SecurityContextHolder can be modified to contain a valid Authentication request token.

***Authentication Process***

The filter requires that you set the authenticationManager property. An AuthenticationManager is required to process the authentication request tokens created by implementing classes. This filter will intercept a request and attempt to perform authentication from that request. Authentication is performed by the attemptAuthentication method, which must be implemented by subclasses.

***Authentication Success***

If authentication is successful, the resulting Authentication object will be placed into the SecurityContext for the current thread, which is guaranteed to have already been created by an earlier filter.

See the successfulAuthentication(HttpServletRequest, HttpServletResponse, FilterChain, Authentication) method for more information.

***Authentication Failure***

If authentication fails, it will delegate to the configured AuthenticationFailureHandler to allow the failure information to be conveyed to the client. The default implementation is SimpleUrlAuthenticationFailureHandler , which sends a 401 error code to the client.

**What is SecurityContextHolderAwareRequestFilter?**

A Filter which populates the ServletRequest with a request wrapper which implements the servlet API security methods. Use it to install a Spring Security aware HttpServletRequestWrapper into your servlet container.

**What is JaasApiIntegrationFilter?**

A Filter which attempts to obtain a JAAS Subject and continue the FilterChain running as that Subject.

By using this Filter in conjunction with Spring's JaasAuthenticationProvider both Spring's SecurityContext and a JAAS Subject can be populated simultaneously. This is useful when integrating with code that requires a JAAS Subject to be populated.

**What is JaasApiIntegrationFilter?**

A Filter which attempts to obtain a JAAS Subject and continue the FilterChain running as that Subject.

By using this Filter in conjunction with Spring's JaasAuthenticationProvider both Spring's SecurityContext and a JAAS Subject can be populated simultaneously. This is useful when integrating with code that requires a JAAS Subject to be populated.

**What is RememberMeAuthenticationFilter?**

Detects if there is no Authentication object in the SecurityContext, and populates the context with a remember-me authentication token if a RememberMeServices implementation so requests. If no earlier authentication processing mechanism updated the SecurityContextHolder, and the request presents a cookie that enables remember-me services to take place, a suitable remembered Authentication object will be put there.

**What is AnonymousAuthenticationFilter?**

If no earlier authentication processing mechanism updated the SecurityContextHolder, an anonymous Authentication object will be put there.

**What is ExceptionTranslationFilter?**

ExceptionTranslationFilter is used to handles any AccessDeniedException and AuthenticationException thrown within the filter chain.

If an AuthenticationException is detected, the filter will launch the authenticationEntryPoint. This allows common handling of authentication failures originating from any subclass of AbstractSecurityInterceptor.

If an AccessDeniedException is detected, the filter will determine whether or not the user is an anonymous user. If they are an anonymous user, the authenticationEntryPoint will be launched. If they are not an anonymous user, the filter will delegate to the AccessDeniedHandler.

To use this filter, it is necessary to specify the following properties:

1) AuthenticationEntryPoint indicates the handler that should commence the authentication process if an AuthenticationException is detected. Note that this may also switch the current protocol from http to https for an SSL login.

2) requestCache determines the strategy used to save a request during the authentication process in order that it may be retrieved and reused once the user has authenticated. The default implementation is HttpSessionRequestCache.

**What is FilterSecurityInterceptor?**

FilterSecurityInterceptor is used to protect web URIs and raise exceptions when access is denied.

**How to Create a custom filter?**

Spring Security maintains a filter chain internally where each of the filters has a particular responsibility and custom filters are added or removed from the configuration depending on which services are required.

To create a custom filter class, we must extend GenericFilterBean class provided by Spring security. While creating a custom filter, more specific filter should come higher in order and general filter should be lower in order.

Below functions are available to add filters in default filter chain of Spring Security framework.

***http.addFilter*** - This method ensures that the ordering of the Filters is automatically taken care of (default ordering).

***http.addFilterBefore*** - Allows adding a Filter before one of the known Filter classes.

***http.addFilterAfter*** - Allows adding a Filter after one of the known Filter classes.

***http.addFilterAt(customFilter, UsernamePasswordAuthenticationFilter.class)*** - Adds the Filter at the default location of UsernamePasswordAuthenticationFilter. That means ordering of customFilter and UsernamePasswordAuthenticationFilter is not guaranteed and can be random.

Apart from GenericFilterBean Class, Package org.springframework.web.filter provides other generic filter base classes allowing for bean-style configuration like - AbstractRequestLoggingFilter, DelegatingFilterProxy, RequestContextFilter, OncePerRequestFilter etc.

For "pre-authenticated" scenarios, where Spring Security assumes the incoming request has already been authenticated by some externally configured system like Pre-Authentication Token generation scenarios should extend AbstractPreAuthenticatedProcessingFilter or RequestHeaderAuthenticationFilter from Package org.springframework.security.web.authentication.preauth.

**What are different Spring Security modules (jar files)?**

1) Core - spring-security-core.jar

* org.springframework.security.core
* org.springframework.security.access
* org.springframework.security.authentication
* org.springframework.security.provisioning

2) Remoting - spring-security-remoting.jar

* org.springframework.security.remoting

3) Web - spring-security-web.jar

* org.springframework.security.web

4) Config - spring-security-config.jar

* org.springframework.security.config

5) LDAP - spring-security-ldap.jar

* org.springframework.security.ldap

6) OAuth 2.0 Core - spring-security-oauth2-core.jar

* org.springframework.security.oauth2.core

7) OAuth 2.0 Client - spring-security-oauth2-client.jar

* org.springframework.security.oauth2.client

8) OAuth 2.0 JOSE - spring-security-oauth2-jose.jar - contains Spring Security’s support for the JOSE (Javascript Object Signing and Encryption) framework. The JOSE framework is intended to provide a method to securely transfer claims between parties.

* org.springframework.security.oauth2.jwt
* org.springframework.security.oauth2.jose

9) ACL - spring-security-acl.jar

* org.springframework.security.acls

10) CAS - spring-security-cas.jar

* org.springframework.security.cas

11) OpenID - spring-security-openid.jar

* org.springframework.security.openid

12) Test - spring-security-test.jar

**What are core components of Spring Security?**

1. Authentication - An Interface to represent current user I.e. principal with username, password and authorities
2. UsernamePasswordAuthenticationToken - A Class which implements Authentication interface and creates an Authentication instance
3. AuthenticationManager - An Interface with userDetailsService() and Authentication authenticate(Authentication authentication) throws AuthenticationException methods
4. GrantedAuthority - An Interface representing an authority granted to an Authentication object.
5. SecurityContextHolder - Holds SecurityContext
6. SecurityContext - Holds principal and Authentication objects. Each request is executed by a separate thread and hence each request has separate SecurityContext.
7. Principal Object – Currently logged in user OR user trying to authenticate

SecurityContextHolder.getContext().getAuthentication().getPrincipal()

1. Credentials Object - The credentials that prove the principal is correct. This is usually a password, but could be anything relevant to the AuthenticationManager.
2. UserDetails – An interface to map User bean from database to principal object
3. UserDetailsService - Core interface which loads user-specific data. It is used throughout the framework as a user DAO to provide UserDetails object

UserDetails loadUserByUsername(String username) throws UsernameNotFoundException

**What is a security context?**

There are several ways to determine who the user is. These are a few of the most common ways:

1) Inject a Principal object into the controller method.

2) Inject an Authentication object into the controller method.

3) Use SecurityContextHolder to get at the security context.

4) Use an @AuthenticationPrincipal annotated method.

5) SecurityContext holds security information about the current thread of execution.

6) This information includes details about the principal.

7) Context is held in the SecurityContextHolder.

8) By default the SecurityContextHolder uses a ThreadLocal to store these details, which means that the security context is always available to methods in the same thread of execution, even if the security context is not explicitly passed around as an argument to those methods.

Obtaining information about the current user --

***Authentication authentication = SecurityContextHolder***

***.getContext()***

***.getAuthentication();***

***User user = (User) authentication.getPrincipal();***

**--------------------------------------------------------------------------------------------------------------------**

**How Authentication request is authenticated in Spring Security?**

Authentication request is authenticated in Spring Security as shown in steps below -

1) // Get username and password from external user and form an Authentication request object (Either form login or command console)

Authentication request = new UsernamePasswordAuthenticationToken(name, password);

2) // Invoke authenticate() method with this request object

Authentication result = authenticationManager.authenticate(request);

3) // If Authentication is successful, then store Authentication result object in security context or else throw AuthenticationException - This step is taken care internally by Spring Security Framework

SecurityContextHolder.getContext().setAuthentication(result);

In case of formlogin, all above steps are taken care by Spring Security and authenticate() method is directly called

4) // Implement AuthenticationManager interface to get authenticationManager instance

class AuthenticationManagerImpl implements AuthenticationManager {

public UserDetailsService userDetailsService() {

5) // Return UserDetailsService ( InMemoryUserDetailsManager, JdbcUserDetailsManager, LdapUserDetailsManager) }

public Authentication authenticate(Authentication authentication) throws AuthenticationException {

6) // Input parameter 'authentication' is an Authentication request object from external user

7) // Get UserDetails from UserDetailsService

// UserDetails user = userDetailsService().loadUserByUsername(authentication.getName());

// Variable 'user' is a UserDetails object from system stored user details (either in memory, jdbc or ldap)

// If UserDetails object is not in the DB then it is treated as an anonymous user.

8) // UserDetails object (db) is compared against Authentication request object (external)

9) // If credentials match, return a fully authenticated Authentication object with principal, credentials and authorities

// new UsernamePasswordAuthenticationToken(Object principal, Object credentials, Collection<? extends GrantedAuthority>

10) // else throw BadCredentialsException

// throw new BadCredentialsException("Bad Credentials");

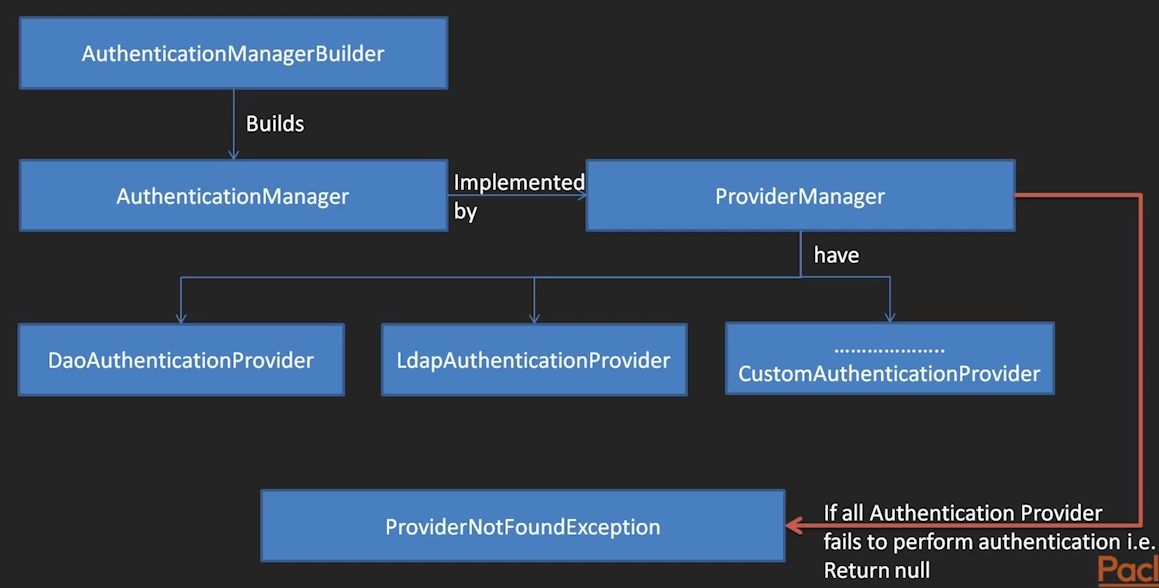
}

}

**How authentication works in a web application?**

1. An authentication object is created out of an incoming request.
2. SecurityContextPersistenceFilter handles the incoming request, takes Authentication object out of request and stores it in SecurityContext. SecurityContext is held by SecurityContextHolder in a ThreadLocal variable to segregate different requests (different principal, separate requests).
3. Request is further passed to FilterSecurityInterceptor filter which extends class AbstractSecurityInterceptor.
4. FilterSecurityInterceptor takes Authentication object out of SecurityContext and checks whether user is anonymous or authenticated. For anonymous user pass to AuthenticationEntryPoint.
5. If isAlwaysAuthicate is true in Authentication object, then it will again authenticate. If Authentication object is already authenticated and authentication is successful it will pass the request to process further.
6. ExceptionTranslationFilter uses AuthenticationEntryPoint and HttpSessionRequestCache.
7. If AuthenticationException occurs it will be passed to ExceptionTranslationFilter and further to AuthenticationEntryPoint.
8. In case of AuthenticationException, if you want to remove request object from cache, remove it from HttpSessionRequestCache.

**How AuthenticationManager, ProviderManager and AuthenticationProvider work together?**



1. AuthenticationManager and AuthenticationProvider are interfaces.
2. ProviderManager is a class which implements AuthenticationManager and has instances of different types of AuthenticationProviders like DaoAuthenticationProvider, LdapAuthenticationProviders etc. and even custom AuthenticationProvider.
3. Filters like UsernamePasswordAuthenticationFilter needs AuthenticationManager for authentication process and Authenticate() method of AuthenticationManager needs UserDetailsService to get UserDetails from DB/memory/LDAP.
4. AuthenticationManagerBuilder in overriden Configure() method builds AuthenticationManager internally. When AuthenticationManagerBuilder builds an AuthenticationManager, that will be ProviderManager. This is due to the fact that ProviderManager implements AuthenticationManager.
5. If you create custom AuthenticationManager then it will take control of authentication request, otherwise by default, ProviderManager acts as an AuthenticationManager.
6. In case of default scenario where ProviderManager acts as an AuthenticationManager, available AuthenticationProviders are executed in order. Available AuthenticationProviders work in a chain one after another. AuthenticationProvider returns either null (failure/AuthenticationException) of Authentication object (success/fully authenticated).
7. If all AuthenticationProviders return null i.e. If all AuthenticationProviders fail to perform authentication at the end of chain, ProviderNotFoundException is thrown. Authentication Scenarios like single-sign-on require custom AuthenticationProvider, otherwise all AuthenticationProviders will return null.
8. In most generic cases Authentication scenario is taken care by default AuthenticationProviders like DaoAuthenticationProvider, or LdapAuthenticationProvider.

**What are the ways to create a custom AuthenticationManager and a custom AuthenticationProvider in Spring Security?**

Below methods can be overriden when SecurityConfiguration class extends WebSecurityConfigurerAdapter.

@Override

protected AuthenticationManager authenticationManager() throws Exception {

return new customAuthenticationManager();

}

// customAuthenticationProvider is required for Authentication Scenarios like single-sign-on

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.authenticationProvider(customAuthenticationProvider);

}

**How AuthenticationProvider and UserDetailsService are used by default for authentication by Spring Security?**

.inMemoryAuthentication() configures Spring Security to use InMemoryUserDetailsManager which (indirectly) implements UserDetailsService interface, so it is a UserDetailsService itself.

DaoAuthenticationProvider is used as AuthenticationProvider implementation by default with inMemoryAuthentication().

.withUser("user").password("password").roles("USER") configures user known to InMemoryUserDetailsManager. This can be used to populate it with users you'd like to use for logging in.

One more thing: not all AuthenticationProviders use UserDetailsService to obtain user details. Actually, among the standard AuthenticationProvider implementations, only DaoAuthenticationProvider class uses UserDetailsService.

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

// Below statements provide UserDetailsService to security configuration

auth.inMemoryAuthentication().passwordEncoder(passwordEncoder()).withUser("user").password("password").roles("USER") .and().withUser("admin").password("password").roles("USER", "ADMIN");

}

// It did not work when PasswordEncoder bean was not provided in above method

@Bean

public PasswordEncoder passwordEncoder(){

return NoOpPasswordEncoder.getInstance();

}

**What is difference between configure() and configureGlobal() methods for security configuration?**

@EnableWebSecurity, @EnableWebMvcSecurity, @EnableGlobalMethodSecurity annotations are annotated with @EnableGlobalAuthentication.

***Option 1: configureGlobal()***

The @EnableGlobalAuthentication annotation (or other related annotations) signals that the annotated class can be used to configure a global instance of AuthenticationManagerBuilder. For example:

@Configuration

@EnableGlobalAuthentication

public class MyGlobalAuthenticationConfiguration {

// This is a config method. Please refer to next question for details about config method

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) {

auth.inMemoryAuthentication().withUser("user").password("password").roles("USER")

.and().withUser("admin").password("password").roles("USER", "ADMIN");

}

}

Method name configureGlobal() can be anything. The only constraints are :

* Annotate the method with @Autowired.
* The method MUST be in a class annotated with one of the following : @EnableWebSecurity, @EnableWebMvcSecurity, @EnableGlobalMethodSecurity, or @EnableGlobalAuthentication.
* The method must have an argument of type AuthenticationManagerBuilder. Configuring AuthenticationManagerBuilder in a class without the @EnableWebSecurity, @EnableWebMvcSecurity, @EnableGlobalMethodSecurity, or @EnableGlobalAuthentication annotations has unpredictable results.

***Option 2: configure()***

Overriding configure() method is a convenient approach in a subclass of WebSecurityConfigurerAdapter (or any @Configuration class implementing WebSecurityConfigurer) but it has the same effect as the other option.

@Configuration

@EnableWebSecurity

public class MyWebSecurityConfiguration extends WebSecurityConfigurerAdapter {

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.inMemoryAuthentication().withUser("user").password("password").roles("USER");

}

}

**How to choose the correct approach between Option 1 and Option 2?**

It's only a question of taste/programming-style because both approaches have the same effect.

The first option make sense when you want/need to keep your configuration in a single class, but your @Configuration class already extends some other class (and you don't want to implement the whole WebSecurityConfigurer interface). Spring provides many Adapter classes that you can extends to speed up the development of your Spring configuration.

As an example, let's take a commonly used Adapter : WebMvcConfigurerAdapter. You will start with a very simple configuration like this :

@EnableWebMvc

@Configuration

@ComponentScan({ "com.company.mypackage" })

public class SpringWebConfig extends WebMvcConfigurerAdapter {

}

What's important here : your class already extends an Adapter class, so you can't extends another one.

Now, you need to add security configuration. You have the choice between including it in your existing SpringWebConfig configuration class or create a new security specific configuration class. Here is a sample of both approaches:

***1) Single @Configuration class approach***

What's important to note here : SpringWebConfig extends WebMvcConfigurerAdapter +

@EnableWebSecurity

@EnableWebMvc

@Configuration

@ComponentScan({ "com.company.mypackage" })

@EnableWebSecurity

public class SpringWebConfig extends WebMvcConfigurerAdapter {

@Autowired

public void whatever(AuthenticationManagerBuilder auth) throws Exception {

auth.inMemoryAuthentication()

.withUser("user").password("password").roles("USER").and()

.withUser("admin").password("password").roles("USER", "ADMIN");

}

}

***2) Specific security @Configuration class:*** Keep your SpringWebConfig as it was and create a new @Configuration class

What's important to note here : MySecurityConfig extends WebSecurityConfigurerAdapter

@Configuration

@EnableWebSecurity

public class MySecurityConfig extends WebSecurityConfigurerAdapter {

@Override

public void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.inMemoryAuthentication()

.withUser("user").password("password").roles("USER").and()

.withUser("admin").password("password").roles("USER", "ADMIN");

}

}

**What is config method in Spring?**

Any method that is annotated by @autowired is a config method.

***Difference between a normal method and a config method in spring***

Config method will be automatically invoked when the bean instance created, after constructor but before @PostConstruct. The parameters of config method will be autowired from the application context.

The name of the method doen’t matter and parameter number doesn’t matter. In fact we often use @autowired before setter method, that’s just a special case of spring config method.

***Usage of config method***

Genetic config method is not widely use as field injection, setter injection or constructor injection, but config method is used in spring security. According to spring security official reference here, the first step to config spring security is to extend from WebSecurityConfigurerAdapter like below.

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

auth.inMemoryAuthentication().withUser("user").password("password").roles("USER");

}

}

This is a good example of using config method. config method will be automatically invoked when bean instantiated, that’s why the document said

The name of the configureGlobal method is not important because it just a config method, and will be automatically invoked with a AuthenticatonManagerBuilder bean from context. The purpose of this method is just to using AuthenticationManagerBuilder to setup authentication provider before real logic begins.

**What are different Authentication types in Spring Security?**

***inMemoryAuthentication()*** - This method also ensure that a UserDetailsService is available for the getDefaultUserDetailsService() method. Additional UserDetailsService's may override this UserDetailsService as the default.

***ldapAuthentication()*** - This method does NOT ensure that a UserDetailsService is available for the getDefaultUserDetailsService() method.

***jdbcAuthentication()*** - This method also ensure that a UserDetailsService is available for the getDefaultUserDetailsService() method. Additional UserDetailsService's may override this UserDetailsService as the default.

**How JDBC Authentication is implemented in overridden configure method of WebSecurityConfigurerAdapter class?**

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

auth.jdbcAuthentication().dataSource(dataSource)

.usersByUsernameQuery(

"select username,password,enabled from users where username=?")

.authoritiesByUsernameQuery(

"select username,role from user\_roles where username=?");

}

**What implementations of UserDetailsService are provided by Spring Security?**

UserDetailsService is a core interface which loads user-specific data. It is used throughout the framework as a user DAO and is the strategy used by the DaoAuthenticationProvider. The interface provides only one method .

UserDetails loadUserByUsername(String username) throws UsernameNotFoundException

Below are all Known Implementing Classes of UserDetailsService interface --

CachingUserDetailsService, InMemoryUserDetailsManager, JdbcDaoImpl, JdbcUserDetailsManager, LdapUserDetailsManager, LdapUserDetailsService

If it is a \*Manager implementation then it has additional functionality of create, update and delete users. However, \*Service implementation only searches user and returns UserDetails object.

**When do you need custom UserDetailsService implementation?**

Below are some of the scenarios where a custom UserDetailsService implementation is required --

1) If you need to use NoSQL database

2) If you need to change DB table structure in a way that JdbcDaoImpl or JdbcUserDetailsManager can not be used

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

auth.authenticationProvider(authProvider());

}

DaoAuthenticationProvider authProvider(){

DaoAuthenticationProvider provider=new DaoAuthenticationProvider();

provider.setUserDetailsService(new CustomUserDetailsService());

return provider;

}

}

public class CustomUserDetailsService implements UserDetailsService {

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

UserTo userTo = getUserToDetail(username);

User userDetail = new User(

userTo.getUsername(), userTo.getPassword(), userTo.isEnabled(), true, true, true, getAuthorities());

return userDetail;

}

public List<GrantedAuthority> getAuthorities() {

List<GrantedAuthority> authList = new ArrayList<GrantedAuthority>();

authList.add(new SimpleGrantedAuthority("ROLE\_USER"));

authList.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

return authList;

}

public UserTo getUserToDetail(String username) {

MongoOperations mongoOperation = (MongoOperations) mongoTemplate();

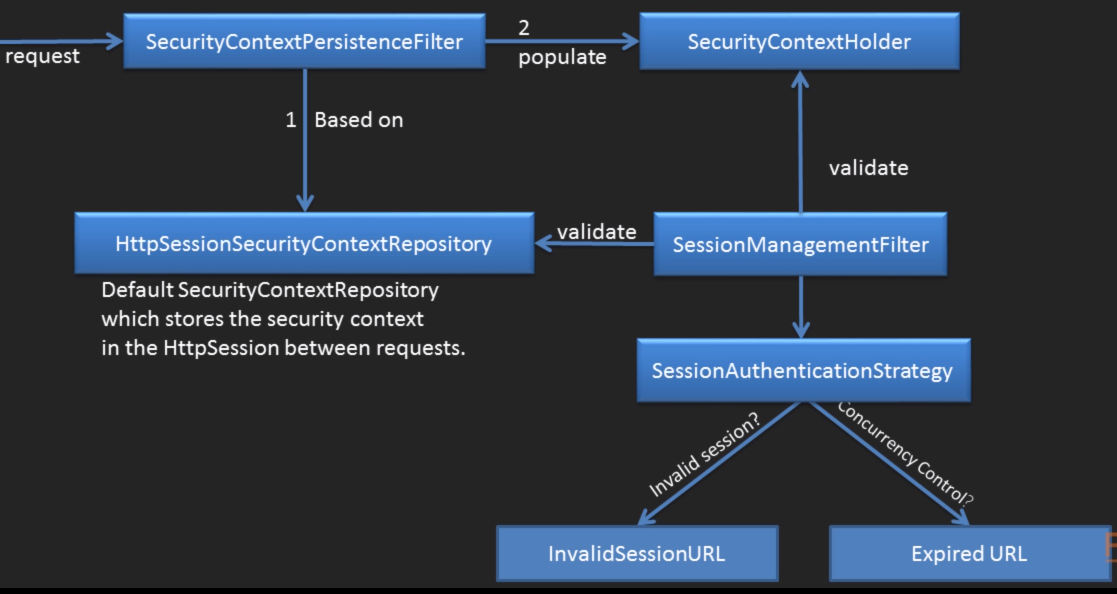
UserTo userTo = mongoOperation.findOne(new Query(Criteria.where("username").is(username)), UserTo.class);

return userTo;

}

}

**Explain session management workflow?**



1. Incoming web request is intercepted by SecurityContextPersistenceFilter.
2. HttpSessionSecurityContextRepository is the default SecurityContextRepository which stores the SecurityContext in the Http session between the requests.
3. SecurityContextPersistenceFilter populates the SecurityContextHolder with information obtained from the HttpSessionSecurityContextRepository.
4. If an Authentication is required for a web request, then sessionManagementFilter validates if SecurityContext in SecurityContextHolder matches SecurityContext in HttpSessionSecurityContextRepository.
5. a) Request is Authenticated - If HttpSessionSecurityContextRepository do not have SecurityContext(Application chose not to store SecurityContext in HttpSessionSecurityContextRepository) but SecurityContextHolder has it.

b) Request is Authenticated - If SecurityContext in HttpSessionSecurityContextRepository matches SecurityContext in SecurityContextHolder.

c) Request is Authenticated and Exception is raised - If SecurityContext in HttpSessionSecurityContextRepository do not match SecurityContext in SecurityContextHolder.

1. When request is authenticated, sessionManagementFilter will call SessionAuthenticationStrategy.
2. SessionAuthenticationStrategy can be used to tasks like Concurrency Control, fixation prevention, and invalid and expired URLs.

a) Check if session is invalidated or user logged out, then redirect to InvalidSessionURL

b) If concurrency is disabled or user exceeds maximum no. of concurrent sessions then invalidate the session and redirect to ExpiredURL

**What is difference between Session fixation and session hijacking?**

Session fixation and session hijacking are both attacks that have a common goal i.e. to gain access to a legitimate session of another user. But the attack vectors are different.

In a session fixation attack, the attacker already has access to a valid session and tries to force the victim to use this particular session. While in a session hijacking attack, the attacker tries to get the ID of a victim's session to use his/ her session.

**What are terms session creation policy, Concurrency Control, fixation prevention, and invalid and expired URLs?**

1) Session Creation Policy

There are 4 kinds of policies defined in Spring Security ---

ALWAYS - Always create a Http session

NEVER - Spring security will never create a new http session but will use it if it is already there.

IF\_REQUIRED - Spring security will only create an Http session if required

STATELESS - Spring security will never create an http session and it will never use it to obtain the SecurityContext

2) Concurrency Control - Should Concurrency be enabled or disabled OR set no. of maximum allowed sessions and expired url

3) InvalidSessionURL - Check if session is invalidated or user logged out, then redirect to InvalidSessionURL

4) Session fixation prevention - Session Authentication Strategy should be to change Session Id and retain other session attributes to prevent session fixation

***http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.IF\_REQUIRED)***

***.maximumSessions(1).expiredUrl("/login")***

***.and().invalidSessionUrl("/login") // If specified, logout().delete-cookies("JSESSIONID") should be specified. If user is logged out but ‘JSESSIONID’ cookie is not deleted then it will invalidate session during next login attempt.***

***.sessionFixation().changeSessionId();***

**What is session scope and how to setup a bean as session scoped?**

@Component

@Scope(value="session",proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class UserTo {

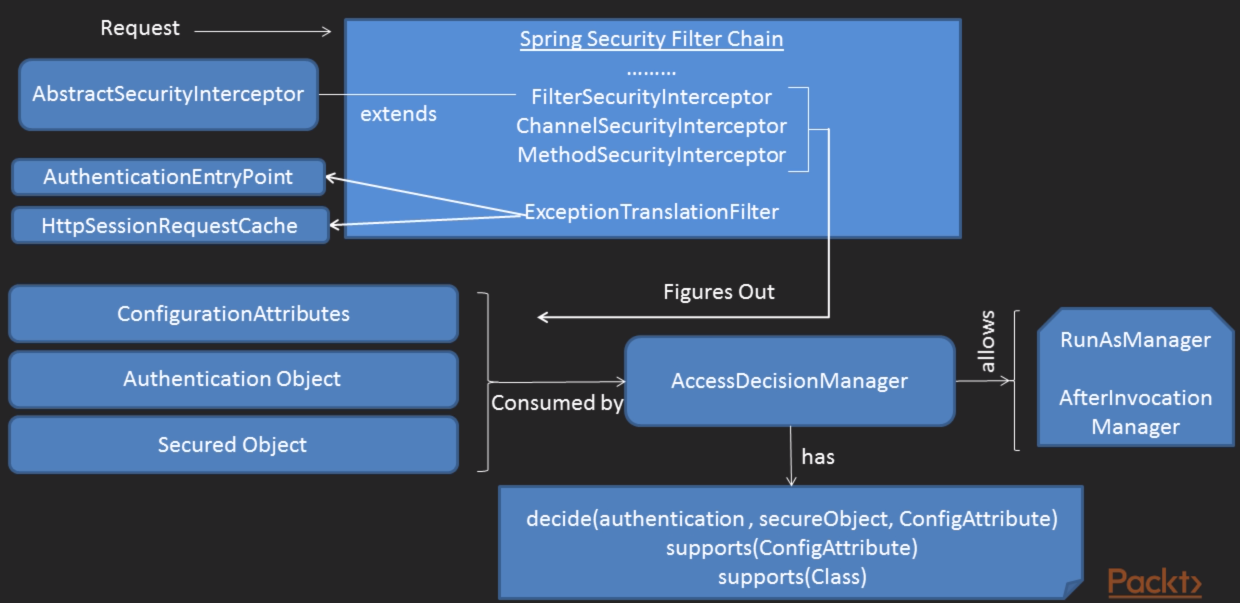
private String username;

private String password;

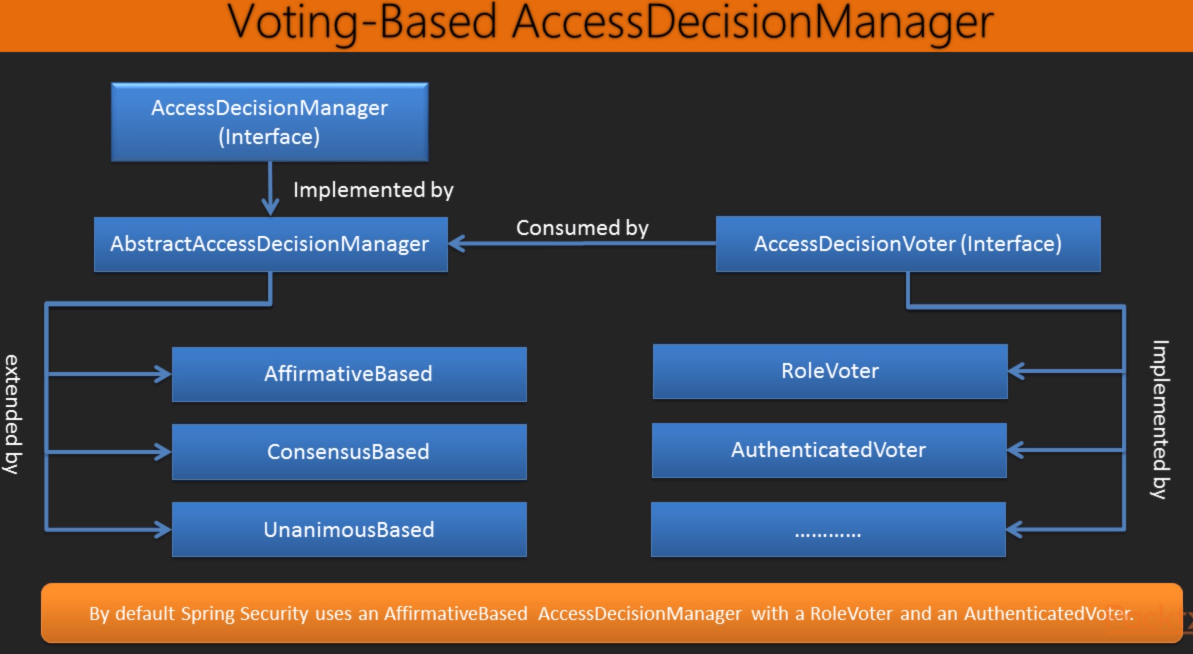
UserTo bean is a session scoped bean. If UserTo bean is autowired in another class (Controller) and session is not created then controller will give an exception as autowire fails! To avoid this, 'proxyMode=copedProxyMode.TARGET\_CLASS' attribute is used so that if session is not created and

bean is autowired somewhere then it will be autowired as a request scoped bean.

**Explain Spring Authorization architecture?**



1. AccessDecisionManager is an interface.
2. Spring security decides whether to allow or deny an access to a particular resource using voting based AccessDecisionManager.
3. Class AbstractAccessDecisionManager implements an interface AccessDecisionManager.
4. Class AbstractAccessDecisionManager is extended by AffirmativeBased, ConsensusBased and UnanimousBased classes.
5. Class AbstractAccessDecisionManager and it's subclasses need votes to make access decision for a particular resource.
6. These votes (ACCESS\_GRANTED, ACCESS\_DENIED, ACCESS\_ABSTAIN) are provided by RoleVoter and AutheticatedVoter classes.
7. RoleVoter and AutheticatedVoter classes implement AccessDecisionVoter interface.
8. By default, Spring Security uses an AffirmativeBased AccessDecisionManager with a RoleVoter and an AutheticatedVoter.



**What are AffirmativeBased, ConsensusBased and UnanimousBased implementations of AccessDecisionManager interface ?**

1. ***AffirmativeBased*** - Simple concrete implementation of AccessDecisionManager that grants access if any AccessDecisionVoter returns an affirmative response.
2. ***ConsensusBased*** - Simple concrete implementation of AccessDecisionManager that uses a consensus-based approach. "Consensus" here means majority-rule (ignoring abstains).
3. ***UnanimousBased*** - Simple concrete implementation of AccessDecisionManager that requires all voters to abstain or grant access.

**What are RoleVoter and AutheticatedVoter implementations of AccessDecisionVoter interface ?**

1. ***RoleVoter*** implementation of AccessDecisionVoter is Role-based which treats passed configuration attributes as simple role names(like ROLE\_USER, ROLE\_ADMIN etc.) and votes to grant access if the user has been assigned that role. In short, it will fetch authorities starting with 'ROLE\_' using getAuthority() and if one or more matches are found with the passed config attributes, it will vote to grant access or else will deny the access. If no config attributes beginning with 'ROLE\_' is passed, then the vote will abstain (ACCESS\_ABSTAIN).
2. ***AutheticatedVoter*** implementation of AccessDecisionVoter is used to make access decisions for anonymous, fully-authenticated and Remember-me authenticated users.

**What is password encoding in Spring Security?**

Password encoding is basically not storing the password in plaintext. There are a few encoding mechanism supported by Spring Security –

1. BCryptPasswordEncoder
2. LdapShaPasswordEncoder
3. Md4PasswordEncoder
4. MessageDigestPasswordEncoder
5. NoOpPasswordEncoder
6. Pbkdf2PasswordEncoder
7. SCryptPasswordEncoder
8. StandardPasswordEncoder

Password encoding in Spring typically involves below steps -

1) Define the Password Encoder in Security configuration

2) Encode the Password during Registration process before storing it in database.

3) Encode the Password on Authentication. we need to inject the password encoder bean we defined earlier into our authentication provider.

@Autowired

private UserDetailsService userDetailsService;

@Bean

public DaoAuthenticationProvider authProvider() {

DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();

authProvider.setUserDetailsService(userDetailsService);

authProvider.setPasswordEncoder(passwordEncoder());

return authProvider;

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

4) When using in-memory authentication, encode password before storing it in memory.

5) PasswordEncoder is a Service interface for encoding passwords. The preferred implementation is BCryptPasswordEncoder.

**What is a DelegatingPasswordEncoder?**

A password encoder that delegates to another PasswordEncoder based upon a prefixed identifier.

Migrating password encoders is a real-life problem and Spring Security 5 gives a quite handy way to easily handle it by supporting multiple passwordEncoders at once.

***Constructing an instance***

You can easily construct an instance using PasswordEncoderFactories. Alternatively, you may create your own custom instance. For example:

String idForEncode = "bcrypt";

Map encoders = new HashMap<>();

encoders.put(idForEncode, new BCryptPasswordEncoder());

encoders.put("noop", NoOpPasswordEncoder.getInstance());

encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());

encoders.put("scrypt", new SCryptPasswordEncoder());

encoders.put("sha256", new StandardPasswordEncoder());

PasswordEncoder passwordEncoder = new DelegatingPasswordEncoder(idForEncode, encoders);

And then @Bean needs to be declared using the DefaultPasswordEncoderFactories instead.

@Bean

public PasswordEncoder passwordEncoder() {

return DefaultPasswordEncoderFactories.createDelegatingPasswordEncoder();

}

***Password Storage Format***

The general format for a password is: {id}encodedPassword

Such that "id" is an identifier used to look up which PasswordEncoder should be used and "encodedPassword" is the original encoded password for the selected PasswordEncoder. The "id" must be at the beginning of the password, start with "{" and end with "}". If the "id" cannot be found, the "id" will be null. For example, the following might be a list of passwords encoded using different "id". All of the original passwords are "password".

{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG

{noop}password

{pbkdf2}5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca763d8dc{sha256}97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605abcbc0

If the password hash has no prefix, the delegation process uses a default encoder. Hence, by default, we get the StandardPasswordEncoder. The DelegatingPasswordEncoder class allows us to set a defaultPasswordEncoder. It means that whenever the framework tries and doesn’t find a prefix in the stored password, it will fall back to the default one to try to decode it.

Let’s assume we want to support:

* ***bcrypt as our new default*** -- Going forward if no prefix is provide in password, then it will be encoded/decoded as bcrypt
* ***scrypt as another supported encoding algorithm by our security config***
* ***SHA-256 as the currently used algorithm***. --- Default Password encoder for existing password entries in the database. Since password is already encoded in this format, this algorithm will be used for decoding them.

***The configuration for this set-up will look like this:***

@Bean

public PasswordEncoder delegatingPasswordEncoder() {

String encodingId = "bcrypt";

PasswordEncoder defaultEncoder = new StandardPasswordEncoder();

Map<String, PasswordEncoder> encoders = new HashMap<>();

encoders.put(encodingId, new BCryptPasswordEncoder());

encoders.put("scrypt", new SCryptPasswordEncoder());

DelegatingPasswordEncoder passworEncoder = new DelegatingPasswordEncoder(

encodingId, encoders);

passworEncoder.setDefaultPasswordEncoderForMatches(defaultEncoder);

return passworEncoder;

}

**What is Basic Authentication in Spring Security?**

Basic authentication is the simplest http authentication where the user agent(like chrome/IE browsers or curl) provides username/password with each request. Basic authentication uses the easily reversible Base64 encoding instead of encryption, making it non-secure unless used in conjunction with TLS (https). Basic authentication is often used with stateless clients which pass their credentials on each request. It's quite common to use it in combination with form-based authentication where an application is used through both a browser-based user interface and as a web-service.

BasicAuthenticationFilter and BasicAuthenticationEntryPoint is provided by Spring Security by default. BasicAuthenticationEntryPoint can be customized to add failure response.

You must provide realmName in basic Authentication otherwise it will not work. It can be an application name. When using basic/digest authentication on browsers like Chrome/IE, login form is provided by browser itself and not by application or Spring framework.

**What is Digest Authentication in Spring Security?**

Digest access authentication is one of the agreed-upon methods a web server can use to negotiate credentials, such as username or password, with a user's web browser. This can be used to confirm the identity of a user before sending sensitive information, such as online banking transaction history. It applies a hash function to the username and password before sending them over the network and hence it is more secure than the basic authentication. It is also immune to replay-attacks, as it uses a one-time number from the server (a nonce - “number once”).

Central to Digest Authentication is a "nonce". This is a value the server generates. Spring Security's nonce adopts the following format:

base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))

expirationTime: The date and time when the nonce expires, expressed in milliseconds

key: A private key to prevent modification of the nonce token

You can think of it like this:

1. Client makes request
2. Client gets back a nonce from the server and a 401 authentication request
3. After nounce is created, both client and server create a digest or hash using generate\_md5\_key function.
4. Client sends back the following response array (username, realm, generate\_md5\_key(nonce, username, realm, URI, password\_given\_by\_user\_to\_browser))
5. The server takes username and realm (plus it knows the URI the client is requesting) and it looks up the password for that username. Then it goes and does its own version of generate\_md5\_key(nonce, username, realm, URI, password\_I\_have\_for\_this\_user\_in\_my\_db) i.e digest
6. It compares the client sent digest/hash with server generated digest or hash, if they match the client is authenticated, otherwise not.
7. Nounce expiration time by defaullt is 5 minutes. When nounce expires but digest is valid, DigestAuthenticationEntryPoint sends "stale=true" in header with new nounce. User-agent tries with new nounce without invalidating current user session.

The configured UserDetailsService is needed because DigestAuthenticationFilter must have direct access to the clear text password of a user. It will not work with encrypted password unless it is MD5 (for this reason digest authentication is not recommended to use as MD5 is old encryption mechanism which is security-breach-prone). Digest Authentication will NOT work if you are using encoded passwords in your DAO. The DAO collaborator, along with the UserCache, are typically shared directly with a DaoAuthenticationProvider. The authenticationEntryPoint property must be DigestAuthenticationEntryPoint, so that DigestAuthenticationFilter can obtain the correct realmName and key for digest calculations.

**What is difference between authentication and full authentication in Spring Security?**

In full authentication, user must provide complete credentials while logging in. However, When we say just authentication, it can either be a full authentication by providing username/password on login form or it can be a Remember-me authentication where credentials are not provided again by the user but authentication happens based on the cookie stored in browser.

Apart from Remember-me it also can be an Anonymous authentication which gives users access to the public areas of your Website without prompting them for a username or password.

**What is anonymous authentication in Spring Security?**

Note that there is no real conceptual difference between a user who is "anonymously authenticated" and an unauthenticated user. Spring Security’s anonymous authentication just gives you a more convenient way to configure your access-control attributes. Calls to servlet API calls such as getCallerPrincipal, for example, will still return null even though there is actually an anonymous authentication object in the SecurityContextHolder.

There are other situations where anonymous authentication is useful, such as when an auditing interceptor queries the SecurityContextHolder to identify which principal was responsible for a given operation. Classes can be authored more robustly if they know the SecurityContextHolder always contains an Authentication object, and never null.

If you need to check if it is an anonymousUser then you can check whether Authentication object is AnonymousAuthenticationToken instance or not.

**What is Remember-Me authentication in Spring Security?**

Ideally Remember-Me cookie should be deleted once user explicitly logs out.

Critical functionalities like profile updates are not allowed with Remember-Me authentication because hacker can change profile details like password.

There are 2 types of Remember-Me authentication ---

***1) Hash-based token Remember-Me authentication***

When user logs in for the first time (full authentication), server generates a hash-based token and returns it to the user-agent in the form of a cookie. When user logs in next time for the same website, he will be logged in without providing credentials unless the cookie has expired.

base64(username + ":" + expirationTime + ":" + md5Hex(username + ":" + expirationTime + ":" + password + ":" + key))

***expirationTime:*** Remember-me token expiration time

***key:*** A private key for Remember-me token

In case of a full authentication only JSESSIONID cookie will be there on browser, but in case of Remember-me authentication, a remember-me cookie will be there along with JSESSIONID cookie.

If JSESSIONID cookie is deleted/expired in case of a full authentication user will be logged out. However, in case of Remember-me authentication, when JSESSIONID cookie is deleted/expired, user still will be able to login with the help of Remember-me cookie.

Hash-based token Remember-Me authentication has security vulnerability as a hacker can reuse existing Remember-me cookie from any other terminal.

***2) Persistent-token Remember-Me authentication***

To counter security vulnerability of Hash-based token Remember-Me authentication, a persistent token is used in Persistent-token Remember-Me authentication. Every time user logs in a new token is generated and a new cookie is sent to the user-agent. This way even if a hacker gets hold of old Remember-Me cookie, authentication would fail as cookie is not updated with new token.

Persistent-token Remember-Me authentication maintains a persistence table having username, series, token and last\_used timestamp columns. When authentication fails, Remember-Me cookie will be deleted and persistence table will be cleared.

**How to customize logout mechanism in Spring Security?**

In order to customize logout mechanism in Spring, below parameters need to be defined --

1) ***logoutUrl*** -- Controller endpoint for logout button/link. Default is /logout. logoutUrl must be mapped to a POST method in controller so that it is enforced to pass a CSRF token with logout request so that hacker is not able to logout user.

2) ***logoutSuccessUrl*** -- Where to redirect to on successful logout. Default is login page. This may be necessary to define in some online shopping applications where merchant wants to redirect user to advertisement on successful logout.

3) ***logoutSuccessHandler*** -- When you customize logoutSuccessHandler, logoutSuccessUrl customized above would be ignored. logoutSuccessHandler can be SimpleUrlLogoutSuccessHandler or HttpStatusReturningLogoutSuccessHandler.

SimpleUrlLogoutSuccessHandler implements LogoutSuccessHandler in order to redirect user to a url after successful logout.

HttpStatusReturningLogoutSuccessHandler implementation of LogoutSuccessHandler is useful in REST-type scenarios where a redirect upon a successful logout is not desired but instead a http status code is returned.

logoutSuccessHandler is called after a successful logout by the LogoutFilter. Custom logoutSuccessHandler may be required in special cases where an email needs to be sent to the logged out user with last used session information.

Override onLogoutSuccess() method and write custom logic before calling super.onLogoutSuccess() method.

Spring security takes care of below activities by default (below activities can be customized too)

a) Delete JSESSIONID cookie

b) Set invalidateHttpSession to 'true'

c) Adding LogoutHandler like SecurityContextLogoutHandler (can be customized by implementing LogoutHandler interface). Below activities can be customized by implementing custom LogoutHandler --

1) TokenBasedRemembermeServices/PersistentBasedRemembermeServices to clear Remember-me cookies

2) CookieClearingLogoutHandler

3) CsrfLogoutHandler

4) SecurityContextLogoutHandler (added by default at the end)

It is not required to customize LogoutHandler since Spring Security takes care of all required activities by default.