**Spring Security**

**Five Spring Security Concepts**

1. Authentication : It is a process of knowing and identifying the user that want to access.
2. Authorization : It is a process to allow authority to perform actions in the application.
3. Principal : It represents the currently logged in user. Basically, it represents information about logged in user.
4. Granted Authority : Action that can be assigned to the principal.
5. Roles : Group Of Authorities that are assigned together.

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

**Adding Spring Security to Spring Boot**.

We need to add a maven dependency called Spring-boot-starter-security in POM.xml

Spring Security Default Behaviour. What happens if we add dependency only and do not configure it.

1. Add Mandatory authentication for URL’s.
2. Add login form.
3. Handles login error.
4. Create a user and set default password.

**How to skip creating default user and default password.**

In application.properties add following properties

1. Spring.security.user.name = username
2. Spring.security.user.password = password

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

**Configuring Spring Boot Security Authentication.**

We can configure authentication by affecting/using Authentication Manager.  
Authentication Manager has a method called ‘authenticate ()’ that either return a successful authentication or it throws an exception.  
We can affect Authentication Manager by using *AuthenticationManagerBuilde*r.  
We can configure using a method named “configure”. “configure” method accepts AuthenticationManagerBuilder as argument.  
This configure method is present in WebSecurityConfigurerAdapter class and this class must be inherited by our own class.  
Also, our own class must be annotated with @EnableWebSecurity.

@EnableWebSecurity tell Spring boot that this class is related to Web Security

For Example:

@EnableWebSecurity  
public class SecurityConfiguration extends *WebSecurityConfigurerAdapter* {  
 @Override  
 protected void configure (AuthenticationManagerBuilder auth) throws Exception {  
 // set your configuration on the auth object  
 // currently implementing in memory authentication.  
 auth.inMemoryAuthentication().withUser(“username”).password(“password”).roles(“role”);  
 }  
}

Note:  
Since Passwords cannot be a simple text, thus we need to have a password encoder

For Example  
@Bean  
public PasswordEncoder getPasswordEncoder() { }

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

**Configure Spring Boot Security Authorization**

As Authentication is completed, we can add authorization.

Here we can assume that we have 3 Rest API’s. One is available publicly and any body can use it. Second one can used by user and admin. Third one can be used only by admins.

Here also we need to override “configure” method, but this time we will pass argument of type **HTTPSecurity**.

http

.authorizeRequests()

.antMatchers("/admin").hasRole("ADMIN")

.antMatchers("/user").hasAnyRole("ADMIN","USER")

.antMatchers("/").permitAll()

.and()

.formLogin();

Note: Put most restrictive role at top.

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

**How Spring Security Authentication Works**

Just by adding *spring-boot-starter-security* to our application, the framework start intercepting the requests with the help of **Filters**.

Filters has the opportunity to do any processing or manipulate the request before it goes to servlet, it can even stop the request.

*Note:- Spring, Spring boot & Spring Security internally uses Servlet.*

Filters : They are the components that intercepts incoming requests, processes the requests and generate appropriate response based on the processes. After processing the filter calls chain.doFilter(request,response) to pass the request to next filter in filter chain.

Filter Chain: It is series of filters through which request and response passes.

Once the request passes through the authentication filter, the credentials of the user are stored in the **Authentication object**. Now, what actually is responsible for authentication is **AuthenticationProvider** A spring app can have multiple authentication providers, one may be using OAuth, others may be using LDAP. To manage all of them, there is an **AuthenticationManager**.

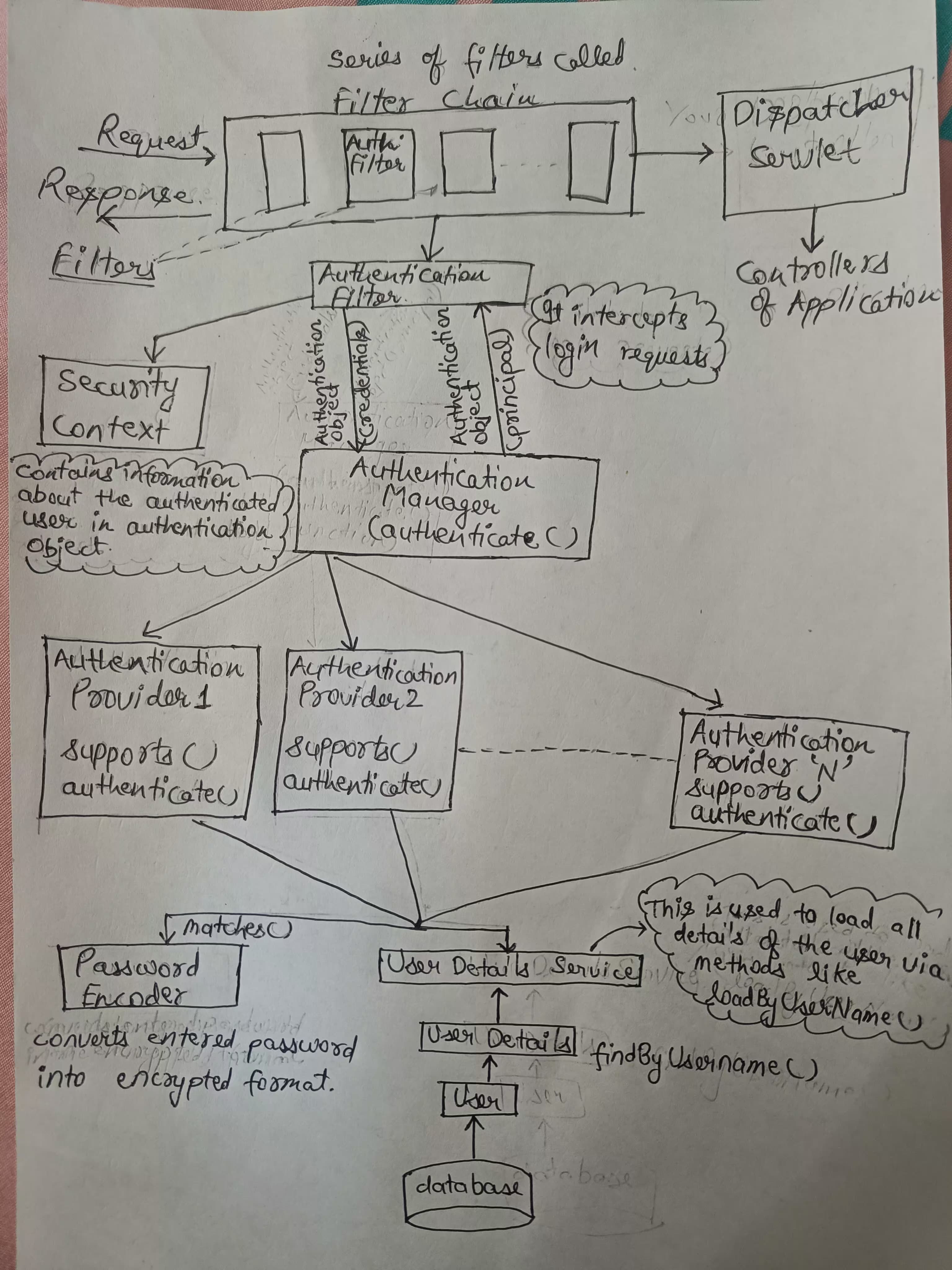
The authentication manager finds the appropriate authentication provider by calling the supports() method of each authentication provider. The supports() method returns a boolean value. If true is returned, then the authentication manager calls its authenticate() method.

After the credentials are passed to the authentication provider, it looks for the existing user in the system by **UserDetailsService**. It returns a **UserDetails** instance which the authentication provider verifies and authenticates. If success, the **Authentication** object is returned with the **Principal** and **Authorities** otherwise **AuthenticationException** is thrown which bubbles up all the way up to the filter where we can catch the exception.

Once the authentication Filter get the authentication object with the principal, The filter saves the authentication object (as a result of successful authentication) into security context so that it can be used in authorization for identifying current principal and more.

For Subsequent Requests, we have another filter whose job is to manages user session. This filter takes in authenticated principal from security context and associates it with user session. This is the reason why we don’t have to authenticate with every request.

Article Source : <https://www.linkedin.com/pulse/how-does-spring-security-works-internally-ayush-jain>



**Authentication Providers**

These are the component that does the verification of credentials provided by user during login process. In spring security we can configure multiple Authentication Providers.

**Some important Authentication Providers**

* DaoAuthenticationProvider : This is the default authentication provider.
* In-Memory AuthenticationProvider
* Ldap Authentication Provider
* PreAuthenticatedAuthenticationProvider : This is for single Sign on.
* OAuth2AuthenticationProvider

Note: All the Authentication Providers must implement AuthenticationProvider Interface.

Any AuthenticationProvider Interface has 2 methods:

* authenticate() : This methods takes an Authentication Object and returns a fully populated Authentication Object (which now have roles, userDetails, etc.)
* supports()

**User Management In Spring Security**

**User Details**

The UserDetails interface is a core component in Spring Security that **represents a user** in the application.  
It provides necessary information about the user, such as username, password,and authorities (roles)

Structure of UserDetails Interface

1. *String getUsername()*
2. *String getPassword()*
3. *Collection<? extends GrantedAuthority> getAuthorities()*
4. *boolean isAccountNonExpi red ( )*
5. *boolean isAccountNonLocked()*
6. *boolean isCredentialsNonExpi red()*
7. *boolean isEnabled()*

**User**

User class implements UserDetails interface.  
It is used to create an Object of Type UserDetails with predefined username, password, and authorities.

**UserDetailsService**

The UserDetailsService interface is responsible for retrieving user-related data.  
It has a single method(loadsUserByUsername) that loads a user based on the username and returns a UserDetails object.

**UserDetailsManager**

The UserDetailsManager interface extends UserDetailsService and provides additional methods for managing user accounts.

Provides additional capabilities for managing user accounts, such as creating, updating, and deleting users, as well as changing passwords and checking for user existence.

*void createUser(UserDetails user)  
void updateUser(UserDetai1s user)  
void deleteUser(String username)  
void changePassword(String oldPassword,String newPassword)  
boolean user Exists (String username)*

**JdbcUserDetailsManager**

JdbcUserDetailsManager is implementation of the UserDetailsManager interface that manages user details using a JDBC-based data source

It provides methods to create, update, delete, and query user accounts, and it interacts with the database using SQL queries.

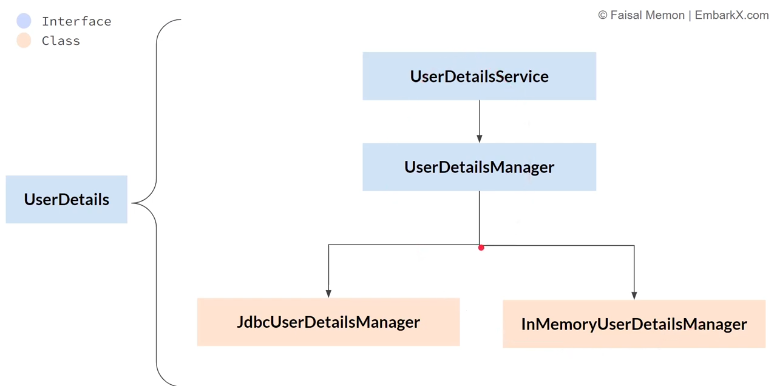
*void createUser(UserDetaiIs user)  
void updateUser(UserDetaiIs user)  
void deleteUser(String username)  
void changePassword(String oldPassword, String newPassword)  
boolean user Exists (String username)  
UserDetails loadUserByUsername(String username)*

**InMemoryUserDetailsManager**

InMemoryUserDetailsManager is another implementation of the UserDetailsManager interface .

It manages user details entirely in memory, which means the user data is stored in memory and is not persistent across application restarts.

*void createUser(UserDetaiIs user)  
void updateUser(UserDetails user)  
void deleteUser(String username)  
void changePassword(String oldPassword, String newPassword)  
boolean user Exists (String username)  
UserDetails loadUserByUsername(String username)*



**METHOD LEVEL SECURITY**

* @PreAuthorize: Checks the given expression before entering the method.  
  *@PreAuthorize( " hasRole( ‘ADMIN’ ) or hasRole(‘USER’)" )  
   public void deleteDocument(Long documentld) {  
   // Method implementation  
  }*
* @Secured: Simpler alternative to @PreAuthorize, used to specify roles directly  
  *@Secured ( " ROLE\_ADMIN" )  
  public void createAccount(Account account) {  
   // Method implementation  
  }*

*@Secured ({“ROLE\_USER” ," ROLE\_ADMIN " } )*

*public Account getAccount(Long accountld) {*

*// Method implementation  
}*

* @RolesAllowed : Specifies roles allowed to invoke the method.

*@RolesAllowed ( “ROLE\_MANAGER” )*

*public void process0rder(0rder order) {*

*// Method implementation  
}*

*@RolesAllowed ( { “ROLE\_USER” , “ROLE\_MANAGER”} )*

*public Order get0rder(Long orderld) {*

*// Method implementation  
}*

* @PostAuthorize

*@PostAuthorize( " return0bject.owner == authentication.name")*

*public Report getReport(Long reportld){*

*// Method implementation*

*return report;*

*} // here owner is coming from report*

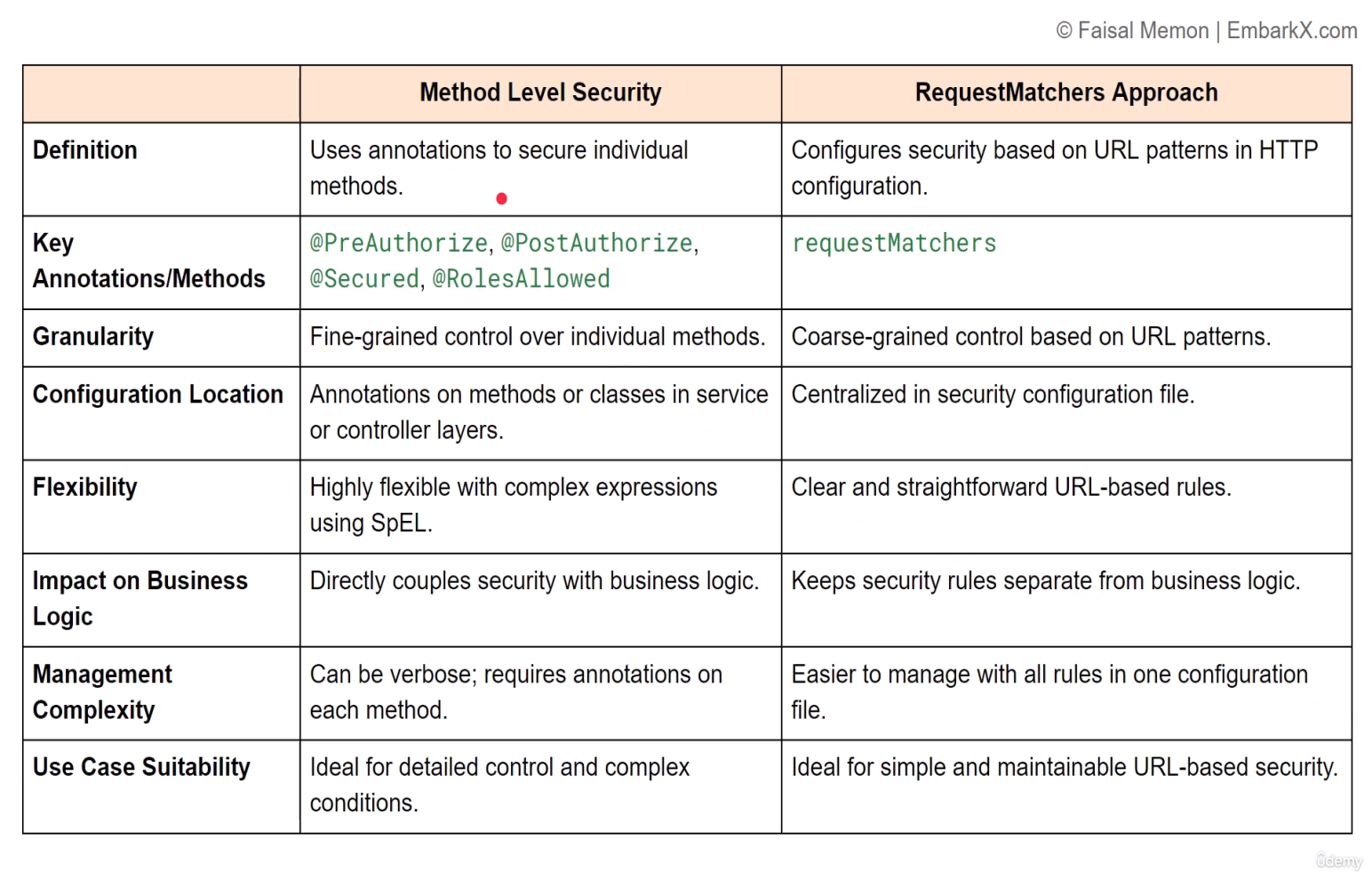
* @PreFilter and @PostFilter   
  They all are used for managing access or basically authorization

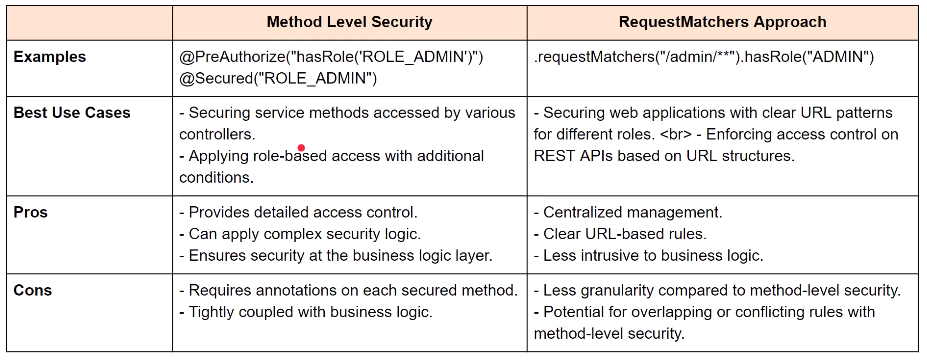
The above mentioned annotations will work only after @EnableMethodSecurity is enabled.

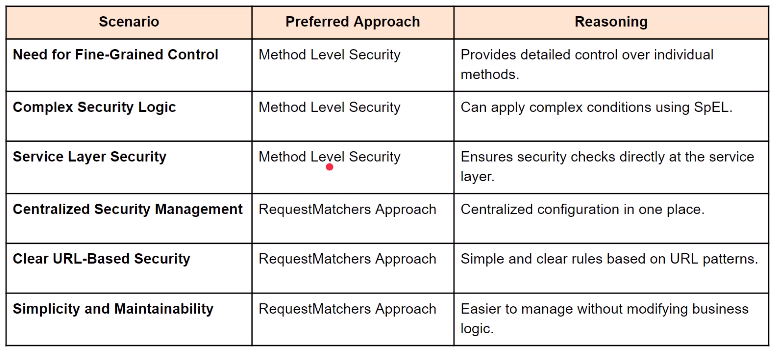
**Different ways of Implementing Authorization**

@EnableMethodSecurity(prePostEnabled = true, jsr250Enabled = true, securedEnabled = true)

*There are 2 ways of implementing authorization using method level security and other way is by using Request Matchers.*

**

**

**

**Hashing**

It is a process of converting a message/password/or any sensitive data into non readable format. It is also accompanied by Salting.

(Password + A random string called Salt) followed by Hashing gives more secure password. This process is also called salting.

We can use Password Encoder to encode the password.

Adding Custom Filters

Scenerios where custom Filters will be used:

1. Custom Authentication and Authorization: Custom filters can implement specialized authentication logic, such as token-based authentication, custom headers, or multi-factor authentication (MFA)
2. Rate Limiting : To prevent abuse of API endpoints, custom filters can implement rate limiting logic
3. IP Whitelisting and Blacklisting: Restrict access of API endpoints to certain IP addresses
4. Geo-Blocking : Restrict access of API endpoints to certain locations
5. Compliance and Logging : Custom filters can be used to implement detailed logging mechanisms for compliance purposes
6. Integration with External Systems : Custom filters can integrate with external systems or third-party services
7. Handling Cross-Cutting Concerns : Custom filters can handle cross-cutting concerns such as logging, transaction management, or modifying requests and responses

Default Filter Chain: It is the chain for filters which is present by default. It is handles by Java class called FilterChainProxy

What is FilterChainProxy?

FilterChainProxy is responsible for managing the filter chain, which is the chain of filters and delegating the request to the appropriate filters. There are different phases like initialization (where spring security would construct filter chain based on configuration defined by us). It contains a method called doFilter() which acts as entry point to FilterChainProxy. Inside doFilter() method we have doFIlterInternal() method which get the list of filters using getFilter() method.

Lifecycle Of Filter

1. Initialization: It is done with the help of method called init(). Init method is used to perform any necessary step or resource allocation.
2. Request Processing : It is handled by doFilter() method. This method is called every time a request/response pair is passed through the filter chain. It performs main filtering task.
3. Cleanup : It is handled by destroy method. This method is called once, when the filter is being removed from service. It is used to release any allocated resource.

All these Methods are present in filter.java interface.

Inbuilt Classes for Filter Implementation.

1. OncePerRequestFilter : This class ensures that this filter is executed only once. We must override doFilterInternal() method.
2. GenericFilterBean : This class provides a simple way to create filters without directly implementing filter interface. This class requires you to implement doFilter() method.

**Methods for adding filters into FilterChain**

1. addFilterBefore()
2. addFilterAfter()

Here is an example of Creating and Using Custom Filter

*@Component  
public class RequestValidationFilter extends OncePerRequestFilter {  
@Override  
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {*

*String header = request.getHeader("X-Valid-Request");  
 if (header == null || !header.equals("true")) {  
 response.sendError(HttpServletResponse.SC\_BAD\_REQUEST, "Invalid request");  
 return;  
 }  
 filterChain.doFilter(request, response);  
 }  
}*

*In SecurityConfig file, add this line 🡪 http.addFilterBefore(new RequestValidationFilter(), UsernamePasswordAuthenticationFilter.class);*

CSRF

CSRF (Cross-Site Request Forgery) is a type of malicious exploit of a website where unauthorized commands are transmitted from a user.

JWTUtils

Contains utility methods for generating a token from a username, validating a JWT, and extracting the username from a token.

AuthTokenFiIter

Filters incoming requests tocheck for a valid JWT in the header, setting the authentication context if the token is valid.

Extracts JWT from request header, validates it, and configures the Spring Security context with user details if the token is valid.

SecurityConfig

Configures Spring Security filters and rules for the application  
Sets up the security filter chain, permitting or denying access based on paths and roles.  
It also configures session management to stateless, which is crucial for JWT usage.

**Some Important Classes/ Interfaces in Spring Security**

1. SpringBootWebSecurityConfiguration : This class has default configuration that is defined by Spring Security. If no other spring security configuration is present this configuration will be picked up by Spring Security.
2. UserDetails
3. User
4. UserDetailsService
5. UserDetailsManager

**HOW SPRING SECURITY WORKS**

1. Create Rest API.
2. For Configuring Spring Security create **SpringSecurityConfiguration** File which extends **WebSecurityConfigurerAdapter** class.
3. In this SpringSecurityConfiguration file we need to override
4. protected void **configure(AuthenticationManagerBuilder auth)** method for authentication
5. protected void **configure(HttpSecurity http)** method for authorization
6. Authorization Example

http

.authorizeRequests()

.antMatchers("/admin").hasRole("ADMIN")

.antMatchers("/user").hasAnyRole("ADMIN","USER")

.antMatchers("/").permitAll()

.and()

.formLogin();

1. Authentication Example
2. @Autowired

MyUserDetailsService myUserDetailsService;

**auth.userDetailsService(myUserDetailsService);**

1. Add **@EnableWebSecurity Annotation**, for this SecurityConfiguration.
2. In Authentication we are using MyUserDetailsService class, which implements **UserDetailsService** **Interface.**
3. We need to mandatorily override loadUserByUserName() method.

CSRF -> Cross Site Request Forgery. It is an attack which tricks a web browser into executing an unwanted action in an application to which a user is logged in.

=====================================================================================

**Project Insights**

Create a Project from Scratch wherein we will use Spring Boot + Spring Security with JPA authentication and MySQL to develop the application.

List Of Dependencies:

1. spring-boot-starter-web (Spring Web Starter)
2. spring-boot-starter-security (Spring Security)
3. spring-boot-starter-data-jpa (Spring Data JPA)
4. mysql-connector-java (MySql Driver)

Create 3 Rest API’s, one can be assessed by any one, one can be accessed by User & Admin, and last one can be assessed by Admin only.

Now Create another class (e.g. ConfigureClientSecurity). In this class we will write code for authentication and authorization. In Authentication we will use userDetailsService. In authorization we will use role based authorization using antMatchers().

Authorization is quite simple to implement, same as done above in example in How Spring Security Works.

In Authentication, we need to create another class (e.g. MyUserDetailsService) which will implements UserDetailsService. Upon implemeting we need to implement loadUserByUsername(String username) method. Also we need to create another class (e.g. MyUserDetails) which will implements UserDetails. In this class as well we need to provide implementation for mandatory methods.

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

JWT (JSON Web Token)

A JWT contains three parts separated by a dot (.). The first two parts (the "header" and "payload") are Base64-URL encoded JSON, and the third is a cryptographic signature.

First Part is a Header. Header consists of 2 parts: type of token which is “jwt” and signing algorithm such as HMAC SHA256 or RSA.  
For Ex: 🡪 { "alg": "HS256", "typ": "JWT"}. This JSON is then encoded.  
  
The second part of the token is the payload, which contains the claims and additional data. Claims are statements about the user. There are three types of claims: registered, public, and private claims.

* Registered claims: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.
* Public claims: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.
* Private claims: These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

An example payload could be:

{"sub": "1234567890", "name": "John Doe", "admin": true }

The payload is then Base64Url encoded to form the second part of the JSON Web Token.

The Third part is Signature. The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify the sender of the JWT.

Project Implementing JWT.

1. Create REST API.
2. Create Rest Service (MyUserDetailsService) which implements UserDetailsService. Also override loadUserByUsername() method
3. Create SecurityConfigurer class which we will do authentication by overriding configure method.
4. Add few maven dependencies for JWT like, jjwt jaxb-api. These dependencies are used for generating and handling jwt.
5. Now Create a Util package which contains a JWTUtils class. This class contains all the the methods related to jwt. These methods are copied from

<https://github.com/koushikkothagal/spring-security-jwt/blob/master/src/main/java/io/javabrains/springsecurityjwt/util/JwtUtil.java>.

1. We need to add Jwtfilter class which extends OncePerRequestFilter. In that we need to override doFilterInternal() method where in we need to provide implementation for the scenerio of handling jwt token.
2. In Configuration we need to provide state management and doFilterChaining()

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

**OAuth**

OAuth is an open-standard authorization protocol or framework that describes how unrelated servers and services can safely allow authenticated access to their assets without actually sharing the initial, related, single logon credential. In authentication parlance, this is known as secure, third-party, user-agent, delegated authorization.

OAuth is meant for Authorization.  
OAuth was meant for a service to authorize another service.

OAuth is an authorization mechanism where services can authorize against each other on your behalf once you’ve given them permission. It is often referred to as delegated access for this reason. It is also an open standard — as it obviously needs to be — because multiple services over the internet need to talk to each other. So there is a specification that all these services need to follow so that they understand each other. There is a certain flow that needs to happen for this whole process to work — the OAuth flow.

**Some Important Filters**

1. SecurityContextPersistenceFilter : Manages the SecurityContext for each request.
2. WebAsyncManagerlntegrationFilter : Integrates the SecurityContext with Spring's WebAsyncManager for asynchronous web requests.
3. HeaderWriterFilter : Adds security-related HTTP headers to the response, such as

X-Content-Type-Options, X-Frame-Options, and X-XSS-Protection.

1. CorsFilter : Handles Cross-Origin Resource Sharing (CORS) by allowing or denying requests from different origins based on configured policies.
2. CsrfFilter : Enforces Cross-Site Request Forgery (CSRF) protection by generating and validating CSRF tokens for each request.
3. LogoutFilter : Manages the logout process by invalidating the session, clearing cookies, and redirecting the user to a configured logout success URL.
4. UsernamePasswordAuthenticationFilter : Processes authentication requests for username and password credentials. It handles the form-based login process.
5. DefaultLoginPageGeneratingFilter : Generates a default login page if no custom login page is provided.
6. DefaultLogoutPageGeneratingFilter: Generates a default logout page if no custom logout page is provided.

Stateless V/S Stateful

In Stateless, user’s data is not stored onto server where as in Stateful, user’s data is stored onto server. No data is presented in cookies in stateless.

Request Matchers : Permits/denies certain requests end points. They provide URL based restrictions.  
.requestMatcher(/pattern/).permitAll()

.requestMatcher(/pattern/).denyAll()

Difference Between 403 v/s 401 error status code.

@Bean

public UserDetailsService userDetailsService() {

InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

if(!manager.userExists("user1")){

manager.createUser(

User.withUsername("user1")

.password("{noop}password1")

.roles("User")

.build()

);

}

if(!manager.userExists("admin")){

manager.createUser(

User.withUsername("admin")

.password("{noop}adminPass")

.roles("Admin")

.build()

);

}

return manager;

}