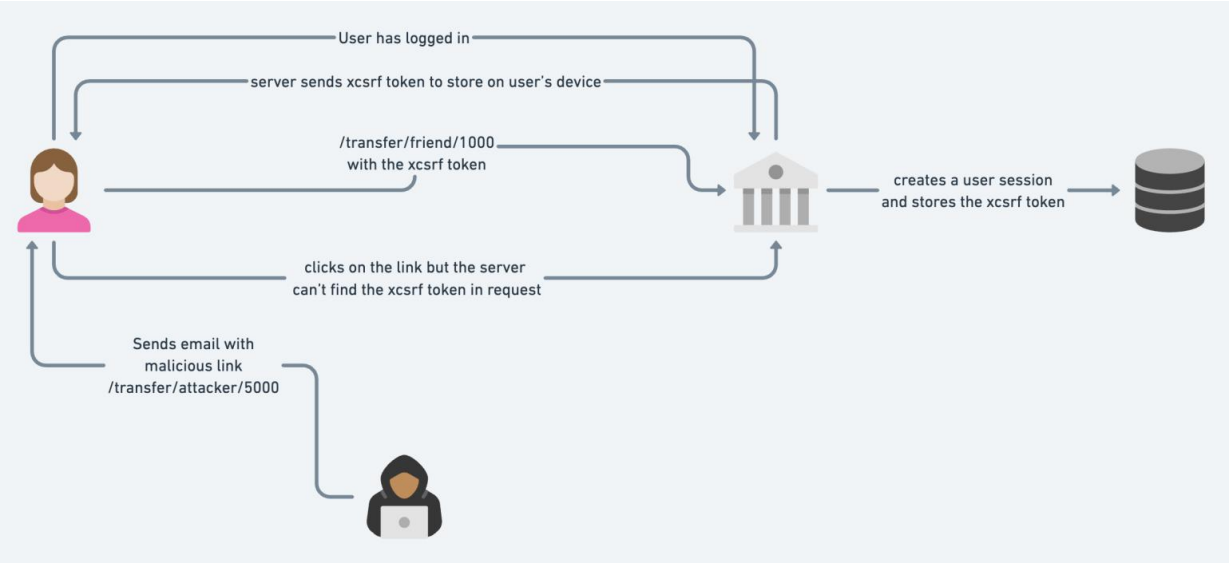
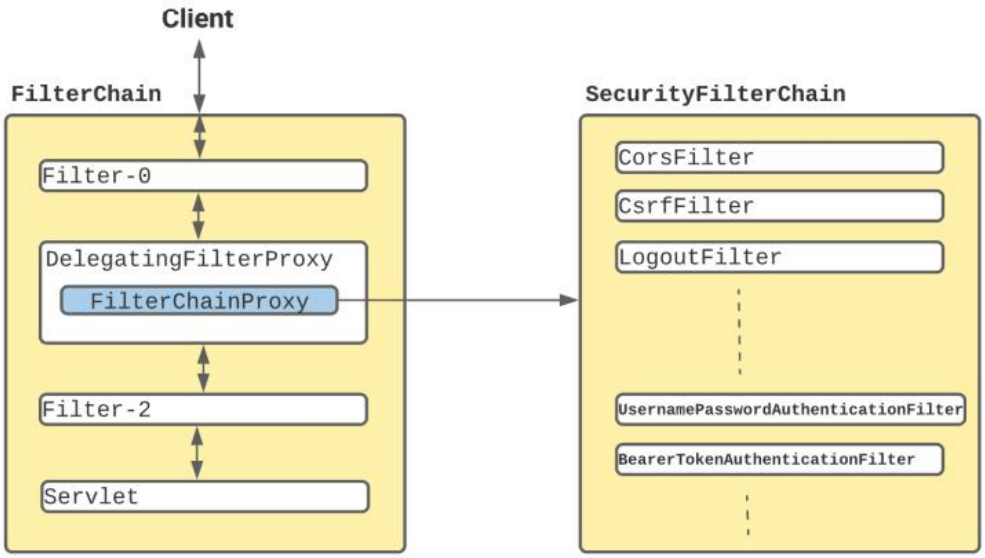
- **CSRF**
  - CSRF (Cross-Site Request Forgery) is an attack where a malicious website tricks a logged-in user's browser into sending an unauthorized request to a trusted website using the user's existing session/cookies.
  - So basically, **CSRF** attacks doesn't steal password; It uses user's cookies stored in the browser to steal the data/money.
  - So, to prevent CSRF, either you need to pass some unique token in the request headers or make the request bind with a particular website (means other website cannot send that request)
  - Someone can steal your cookies, but they don't have your headers.
- With **csrf** token, it can be prevented.
  - 

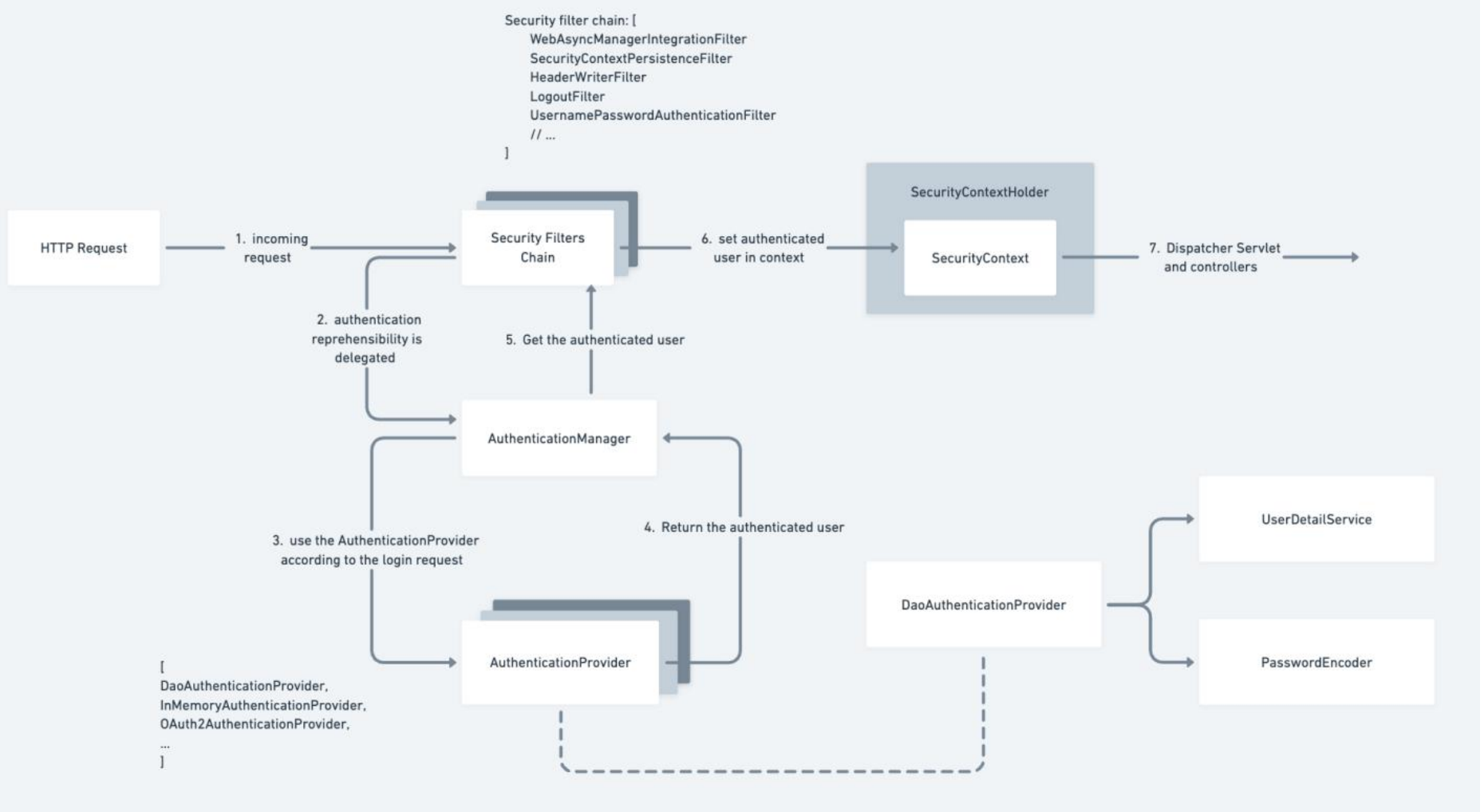
-

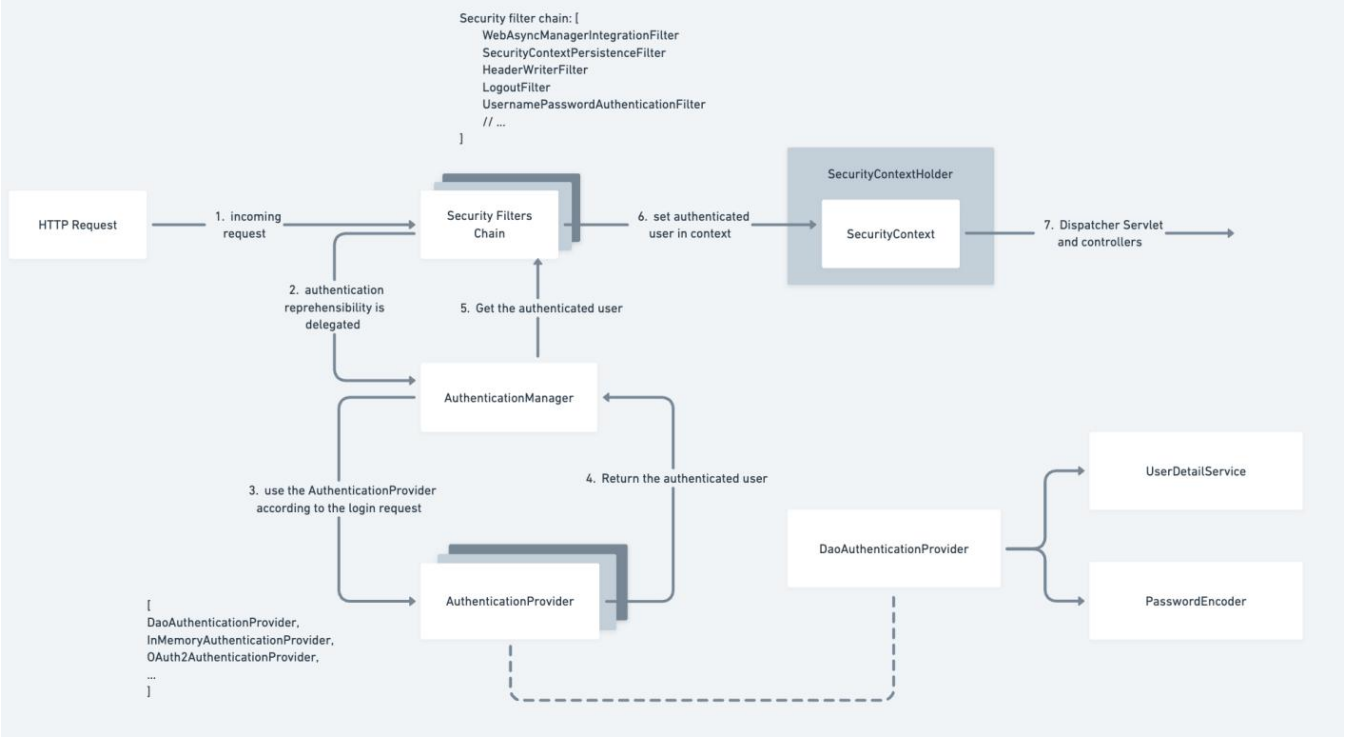
Internal Working Of Spring Security

- The dependency **spring-boot-starter-security** has to be added in pom.xml which groupId is **org.springframework.boot**
- After that spring-boot will auro-configure the security with sensible defaults defined in **WebSecurityConfiguration** class.

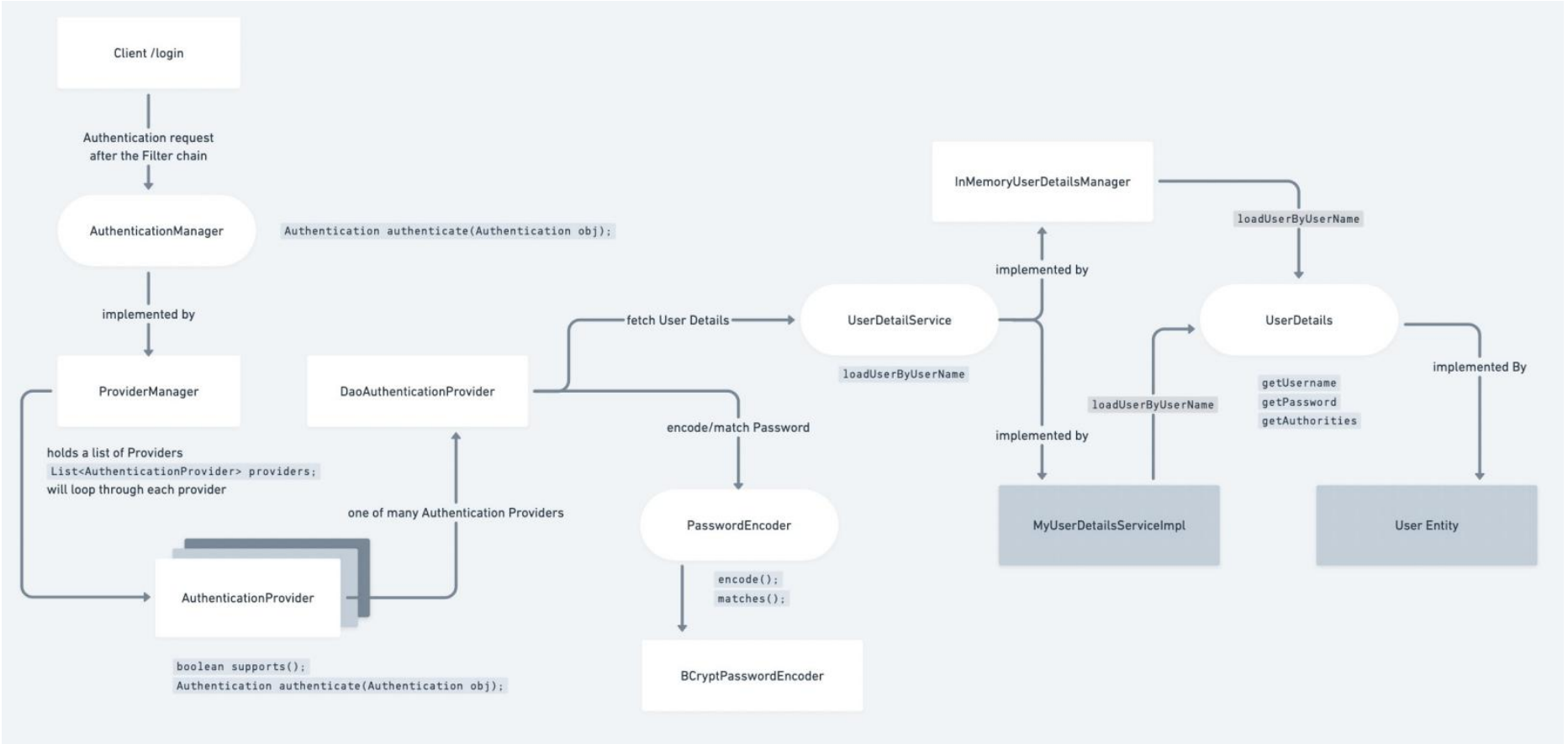


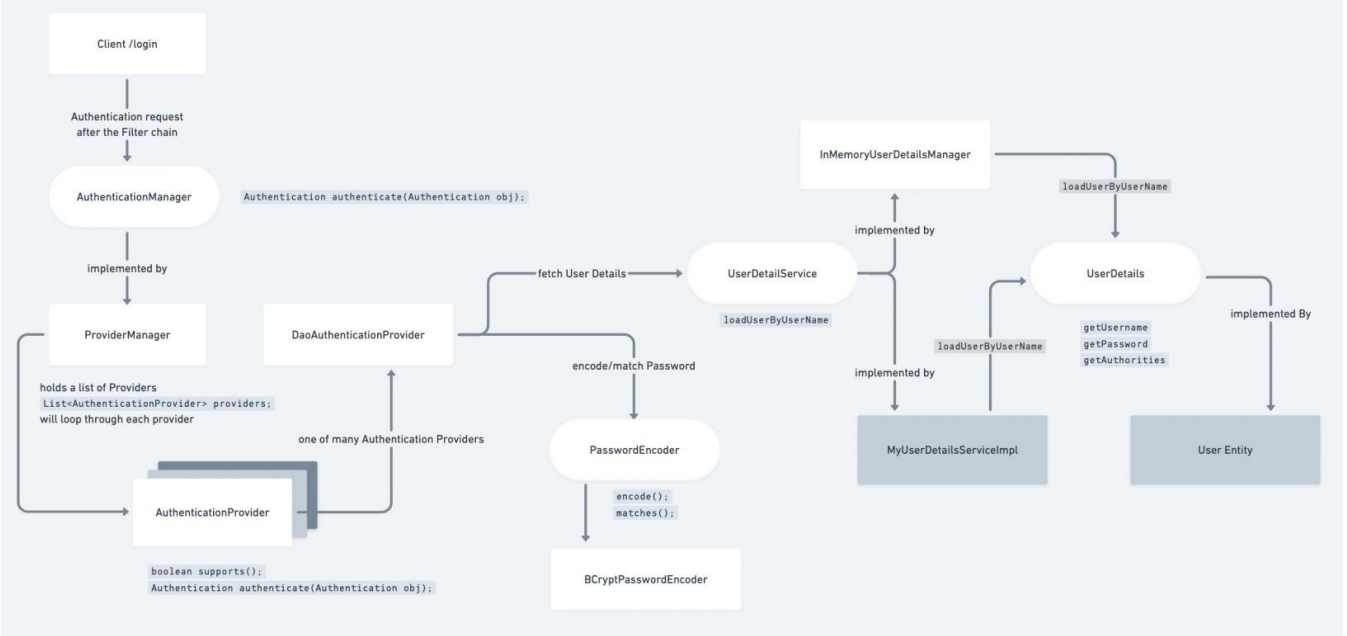
-
  - In Spring Boot application, **SecurityFilterAutoConfiguration** automatically registers the **DelegatingFilterProxy** filter with the name **springSecurityFilterChain**.
  - Once the request reaches to **DelegatingFilterProxy**, Spring delegates the processing to **FilterChainProxy** bean that utilizes the **SecurityFilterChain** to execute the list of all filters to be invoked for the current request.
- Default behaviour of Spring-Security
  - Creates a bean named **springSecurityFilterChain** & registers the ***filters*** with a bean.
- When you run the application after including the dependency **spring-boot-starter-security**, by default one login page will appear to authenticate you.
  - If you inspect that, you'll find one *hidden input* field containing the csrf token as its value which is being attached to request.
  - ```
    ▸ …
      <input name="_csrf" type="hidden" value="DmL2Ac
      <button type="submit" class="primary">Sign in</
    ```
  - Means every time any request being sent, **session id** (cookie) along with **csrf token** (request header) are also sent.
    ```


Security filter chain: [
WebAsyncManagerIntegrationFilter
SecurityContextPersistenceFilter
HeaderWriterFilter
LogoutFilter
UsernamePasswordAuthenticationFilter
// ...
]
HTTP Request
1. incoming request
Security Filters Chain
6. set authenticated user in context
SecurityContextHolder
SecurityContext
7. Dispatcher Servlet and controllers
2. authentication reprehensibility is delegated
5. Get the authenticated user
AuthenticationManager
3. use the AuthenticationProvider according to the login request
4. Return the authenticated user
[
DaoAuthenticationProvider,
InMemoryAuthenticationProvider,
OAuth2AuthenticationProvider,
...
]
AuthenticationProvider
DaoAuthenticationProvider
UserDetailService
PasswordEncoder


Security filter chain: [
WebAsyncManagerIntegrationFilter
SecurityContextPersistenceFilter
HeaderWriterFilter
LogoutFilter
UsernamePasswordAuthenticationFilter
// ...
]
HTTP Request
1. incoming request
Security Filters Chain
6. set authenticated user in context
SecurityContextHolder
SecurityContext
7. Dispatcher Servlet and controllers
2. authentication reprehensibility is delegated
5. Get the authenticated user
AuthenticationManager
3. use the AuthenticationProvider according to the login request
4. Return the authenticated user
DaoAuthenticationProvider
UserDetailService
PasswordEncoder
AuthenticationProvider
[
DaoAuthenticationProvider,
InMemoryAuthenticationProvider,
OAuth2AuthenticationProvider,
...
]

- F
- F
- F
- F


Client /login
Authentication request after the Filter chain
AuthenticationManager
Authentication authenticate(Authentication obj);
implemented by
ProviderManager
holds a list of Providers
List<AuthenticationProvider> providers;
will loop through each provider
AuthenticationProvider
boolean supports();
Authentication authenticate(Authentication obj);
one of many Authentication Providers
DaoAuthenticationProvider
fetch User Details
UserDetailService
loadUserByUserName
encode/match Password
PasswordEncoder
encode();
matches();
BCryptPasswordEncoder
InMemoryUserDetailsManager
implemented by
implemented by
loadUserByUserName
loadUserByUserName
MyUserDetailsServiceImpl
UserDetails
getUsername
getPassword
getAuthorities
implemented By
User Entity



(rectangles: classes, ellipse: interfaces)

- **AuthenticationManager** is an interface.
  - ```
    public interface AuthenticationManager {
        Authentication authenticate(Authentication authentication)
    }
    ```
  - **Authentication** is also an interface containing necessary methods
    - ```
      public interface Authentication extends Principal, Serializable {
          Collection<? extends GrantedAuthority> getAuthorities();  1 imp

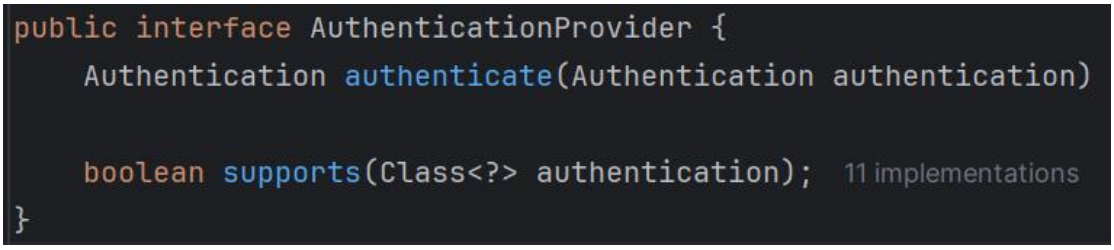
          Object getCredentials();  10 implementations

          Object getDetails();  1 implementation

          Object getPrincipal();  10 implementations

          boolean isAuthenticated();  1 implementation

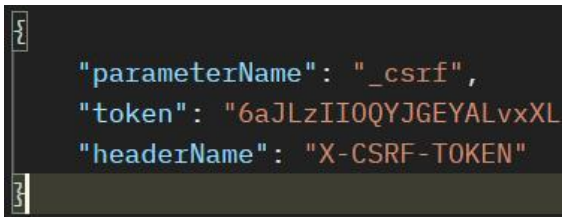
          void setAuthenticated(boolean isAuthenticated) throws IllegalA
      }
      ```
    - After the authentication done, isAuthenticated will be marked as true.
- Now, so many classes implement **AuthenticationManager**, one of those is ***ProviderManager***
  - It holds a list of **AuthenticationProvider** (which is also an interface)

- **AuthenticationProvider**
  - 

  - **supports** : Can I handle this *authentication type*.
  - **supports** : If *yes*, authenticate it
-
- F
- F
- F
- F
-

Udemy Part

- **Client** --- **Servlet Container** --- filter-1 --- filter-2 --- …. --- filter-N --- **Servlet**
  - 2 way (request and response)
- If you include **spring-boot-starter-security**, whenever you'll try to hit any api endpoint, it'll redirect you to the login page;
  - You need to authenticate yourself with username and password then one **session id** will be generated and will be stored in the *browser cookies* .
  - whenever you'll hit any end-point, that **session id** will also be attached with the request.
  - By default username id **user** and password will be generated when you'll run the spring boot application.
  - For customized username and password, you can write those in the **application.properties** file.
    - ```
      spring.security.user.name=alok
      spring.security.user.password=1234
      ```
- In the controller, you can get **HttpServletRequest** type of object which contains details like session id and all.
  - ```
    @GetMapping("hello")  no usages
    public String greet(HttpServletRequest request) {
        System.out.println(request);
        return "Hello " + request.getSession().getId();
    }
    ```
- Get request working; post not working because we are not sending csrf token;
- When you trigger get request, **csrf** token is not required because its read-only;
  - But, for remaining operations, you need to pass **csrf** token in the *headers*.
  - ```
    @GetMapping("csrf-token")  no usages
    public CsrfToken getCsrfToken(HttpServletRequest request) {
        return (CsrfToken) request.getAttribute( s: "_csrf");
    }
    ```
  - I created this end-point to get the **csrf** token.
  - ```
    <p>…</p>
    <input name="_csrf" type="hidden" value="DmL2Ac
    <button type="submit" class="primary">Sign in</
    ```
    - If you see the default login page generated by **spring security**, the name will be **_csrf** here.

- So the attribute name is **_csrf**
- 

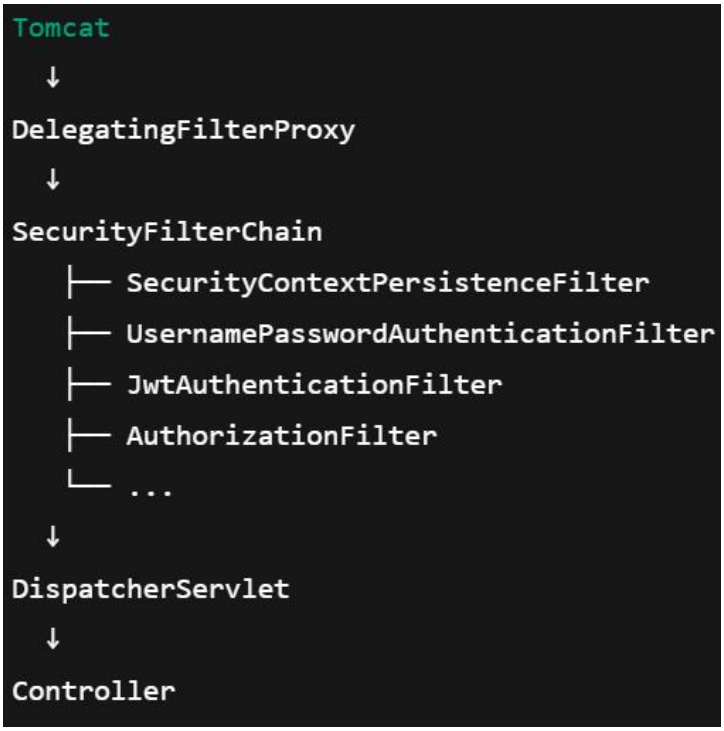
- While triggering POST request via *postman*, you need to pass **X-CSRF-TOKEN** in the headers.
- It is one way of solving CSRF issue; other way is "Don't allow any other website to use your session ID"
  - 

  - Add this in your **application.properties** file and it'll restrict other website to use your session id.

- There are 2 types of application: **stateful** and **stateless**.
  - The one we were using now was **stateful**, because it was using same *session ID* for all the requests.
  - In case of **stateless** we need to pass the **username** & **password** in each request; so there is no need of **csrf token** here.
-

➢ Spring Security Working

- The filters are present in between the **Tomcat** and **DispatcherServlet**'



- **@EnableWebSecurity**
  - It tells Spring "Activate Spring Security's filter chain for web requests".
  - Without it, no security filters are applied.
- In **Spring Boot**, spring security filters are auto-configured if the dependency **spring-boot-starter-security** is present in the pom.xml
- If you create a **bean** of type SecurityFilterChain then Spring will not create bean; menas basically you did override the bean creation.

```
@Configuration  no usages
@EnableWebSecurity
public class SecurityConfig {

    @Bean  no usages
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.build();
    }
}
```

  - Now the filters will not be executed; you need to mention those filters.
-
- F
- F
- F

- F
- F
- F
- F
- F
- F
- vf