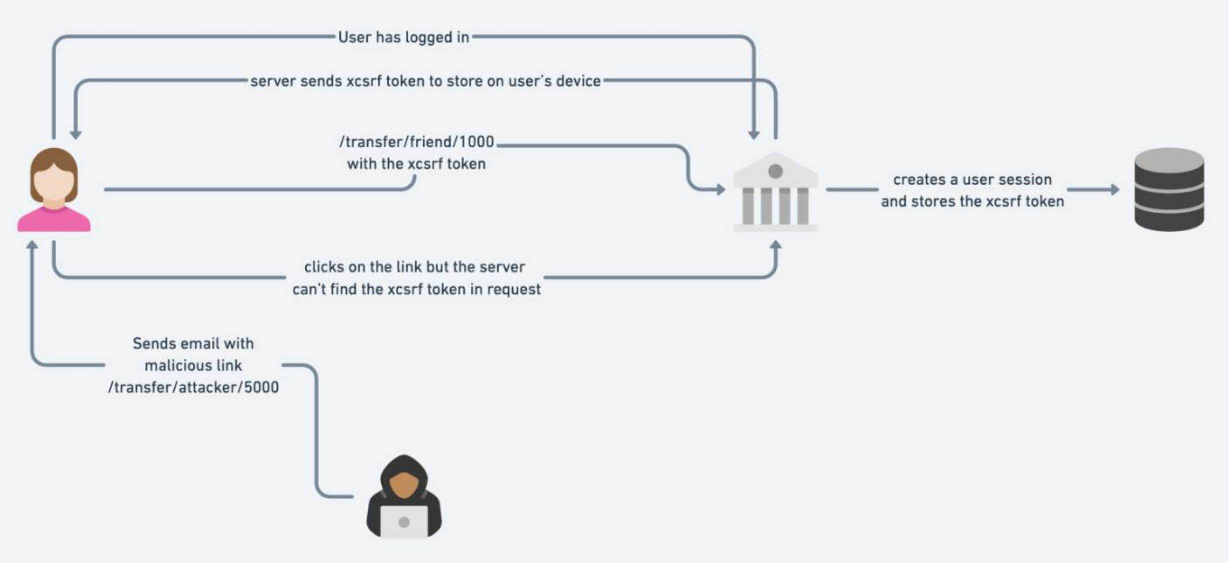
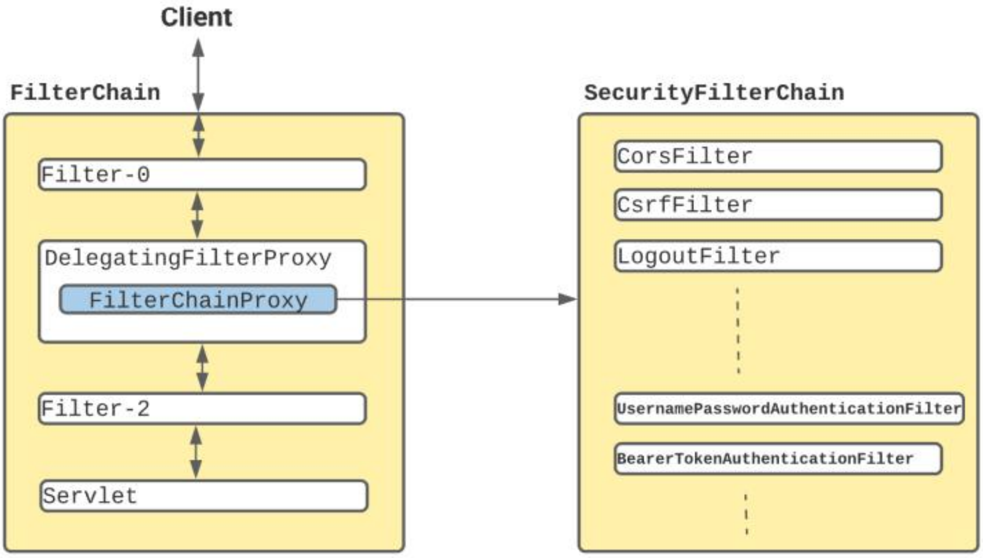
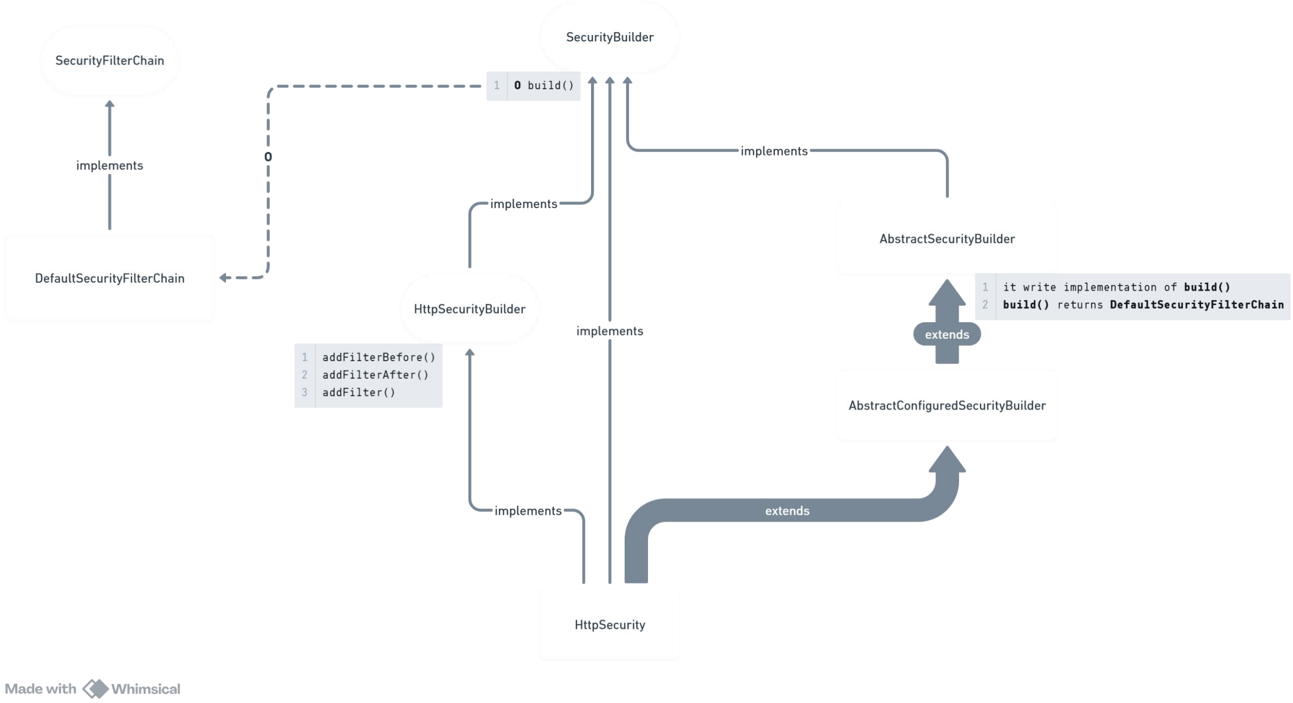
- **CSRF**
  - CSRF (Cross-Site Request Forgery) is an attack where a malicious website tricks a logged-in user's browser into sending an unauthorized request to a trusted website using the user's existing session/cookies.
  - So basically, **CSRF** attacks doesn't steal password; It uses user's cookies 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.
  - 
User has logged in
server sends xcsrf token to store on user's device
/transfer/friend/1000
with the xcsrf token
creates a user session
and stores the xcsrf token
clicks on the link but the server
can't find the xcsrf token in request
Sends email with
malicious link
/transfer/attacker/5000
-

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 ***filter*** with a bean named **springSecurityFilterChain** with the Servlet container for every request.
    -



```
0 build()
```

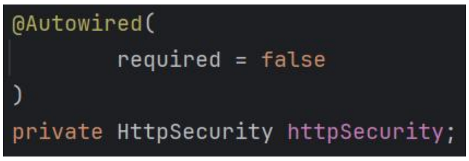
```
addFilterBefore()
addFilterAfter()
addFilter()
```

```
it write implementation of build()
build() returns DefaultSecurityFilterChain
```

➢ This is the flow of **Security Filters**.

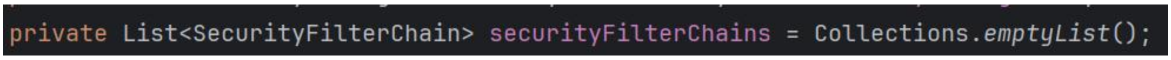
➢ In the class **WebSecurityConfiguration**, the beans are created.

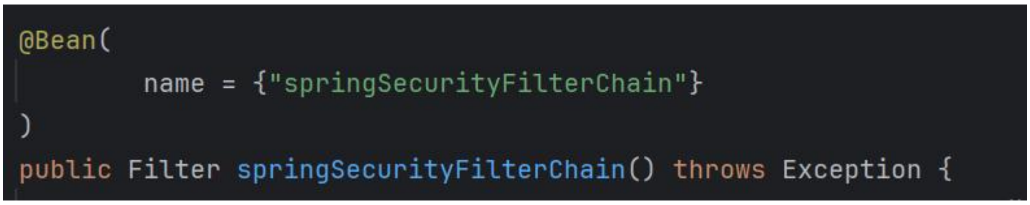
- The bean of **HttpSecurity** is created.

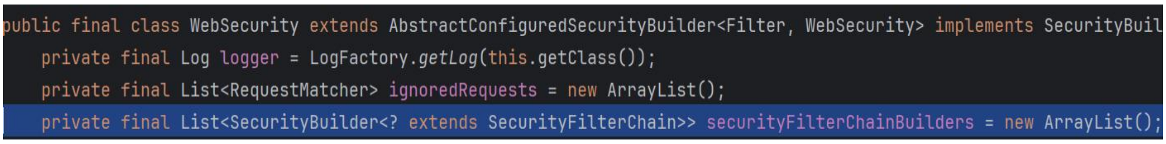


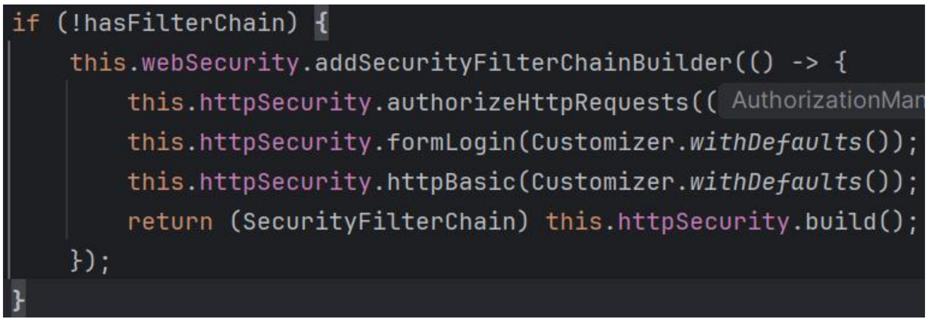
```
@Autowired(
        required = false
)
private HttpSecurity httpSecurity;
```

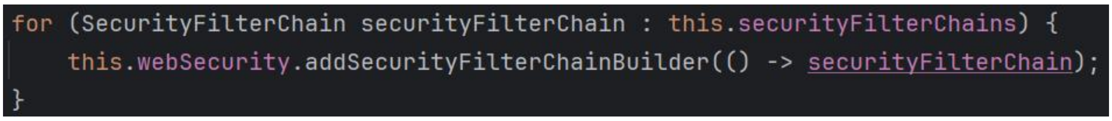
- One **HttpSecurity** type of object can build only **one SecurityFilterChain**.

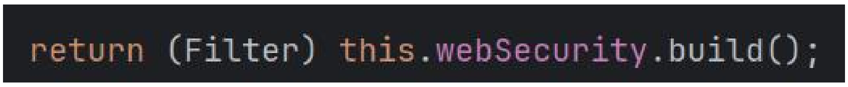
- **HttpSecurity** contains **build()** method that returns object of type **DefaultSecurityFilterChain**. Then why Spring needs to create a bean of **SecurityFilterChain** ?
  - **WebSecurityConfiguration** has a list of **SecurityFilterChain**
    - 

      ```
      private List<SecurityFilterChain> securityFilterChains = Collections.emptyList();
      ```
  - Here one method is there to create the bean named as **springSecurityFilterChain.**
    - 

      ```
      @Bean(
              name = {"springSecurityFilterChain"}
      )
      public Filter springSecurityFilterChain() throws Exception {
      ```
    - But here, **Filter** is the return type; confusing; will discuss at the end :)
  - If there is no filter chains, it'll add the default chain from *HttpSecurity* build() method.
    - it'll not add the default chain to the list i.e. **securityFilterChains**, rather it is being added to **webSecurity** object.
    - 

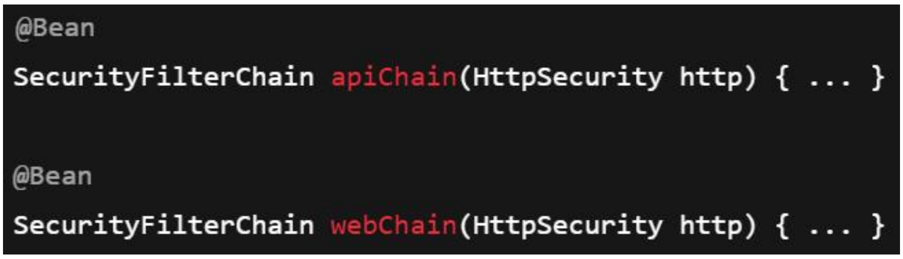
      ```
      public final class WebSecurity extends AbstractConfiguredSecurityBuilder<Filter, WebSecurity> implements SecurityBuil
          private final Log logger = LogFactory.getLog(this.getClass());
          private final List<RequestMatcher> ignoredRequests = new ArrayList();
          private final List<SecurityBuilder<? extends SecurityFilterChain>> securityFilterChainBuilders = new ArrayList();
      ```
    - It is being added here (WebSecurity class).
    - 

      ```
      if (!hasFilterChain) {
          this.webSecurity.addSecurityFilterChainBuilder(() -> {
              this.httpSecurity.authorizeHttpRequests(( AuthorizationMan
              this.httpSecurity.formLogin(Customizer.withDefaults());
              this.httpSecurity.httpBasic(Customizer.withDefaults());
              return (SecurityFilterChain) this.httpSecurity.build();
          });
      }
      ```
  - If filter chains are already there, then it add those filter chains to **webSecurity** object.
    - 

      ```
      for (SecurityFilterChain securityFilterChain : this.securityFilterChains) {
          this.webSecurity.addSecurityFilterChainBuilder(() -> securityFilterChain);
      }
      ```
  - At the end it'll return an object of type **Filter**
    - 

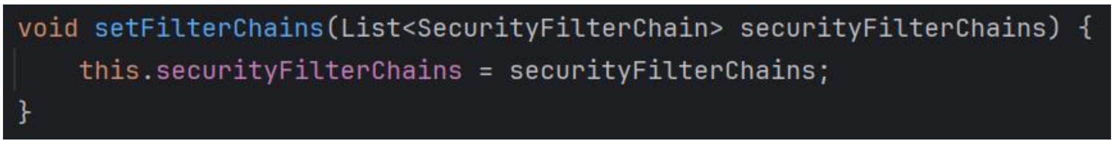
      ```
      return (Filter) this.webSecurity.build();
      ```

➢ Spring Filter Behind the Scene

- What we see, **chain of filters are being executed in between Servlet Container and Servlet.**
  - One **servlet containers** can have many filters, many servlets; but in case of spring we have only one Servlet which is **DispatcherServlet.**
  - And, its not by limitation, but by design spring make sure only one **Filter** should be there in between **Servlet Container** (tomcat in our case) and **Servlet** (DispatcherServlet).
  - So, only **one Filter** should be able to handle **multiple Filter Chains** where **each chain contains multiple Filters.**
    - its like **one object** is equivalent to **list of *list of the same object*.**
  - So, **FilterChainProxy** was introduced which implements **Filter**. And it contains the list of **SecurityFilterChain** objects.
    - And the thing is, this ***SecurityFilterChain*** class contains a method **getFilters()** which returns a list of **Filter** objects.
  - So now, we can return **FilterChainProxy** object instead of **Filter** because *FilterChainProxy* is nothing but the child of *Filter.*
    - And also it contains *list of SecurityFilterChain* which means ***list of (list of (Filter)).***
- If you want to create your own **filter chains.**
  - Don't override the bean creation of the bean **springSecurityFilterChain**, otherwise spring will only create your bean and all the necessary steps like the below will be skipped.
    - Adding the filter chain to **webSecurity.**
    - **customize** using the **Customizer** object..
  - This all will have to be implemented in your own bean creation method.
  - So, create beans of type **SecurityFilterChain**
    - 

```
@Bean
SecurityFilterChain apiChain(HttpSecurity http) { ... }

@Bean
SecurityFilterChain webChain(HttpSecurity http) { ... }
```

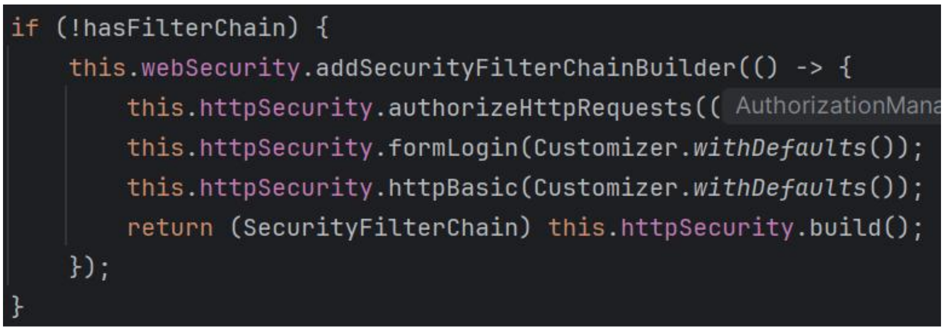
    - Now you might be thinking, if we are creating **multiple beans of same type**, spring will be confused which bean has to be created; but in this case we spring has **list of SecurityFilterChain** not a single object of *SecurityFilterChain*, so all the beans will be added to that list.



```
void setFilterChains(List<SecurityFilterChain> securityFilterChains) {
    this.securityFilterChains = securityFilterChains;
}
```

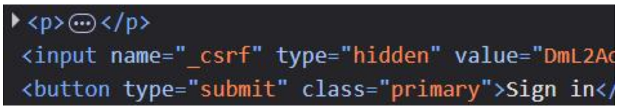
- This method is inside **WebSecurityConfiguration** class.
- All the **SecurityFilterChain** objects that you created beans of, will be passed to this ***setFilterChains*** method.

- Now, when the list i.e. **securityFilterChains** (present inside WebSecurityConfiguration.class) has already some elements present, so the default **filter chain** will not be added to this.

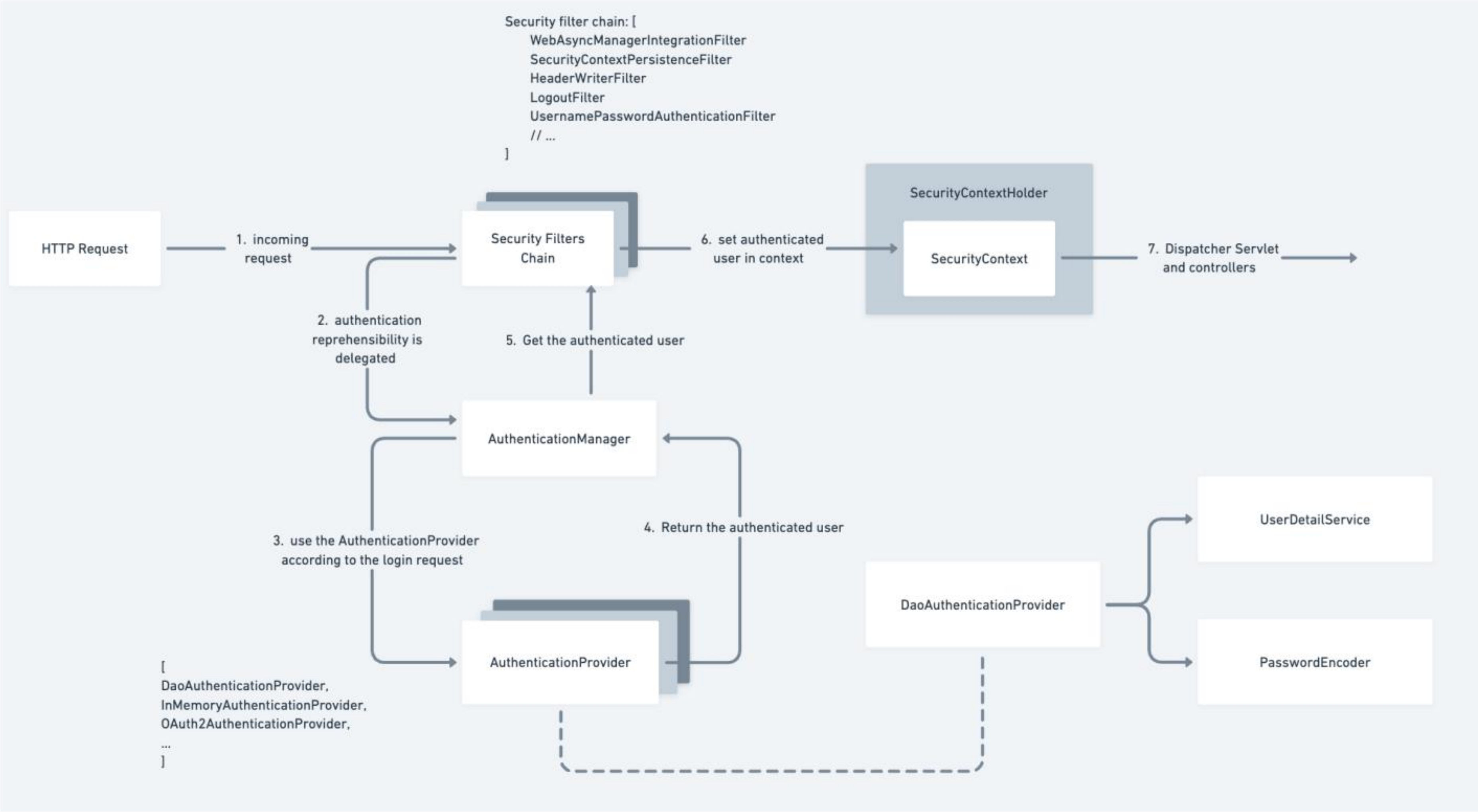


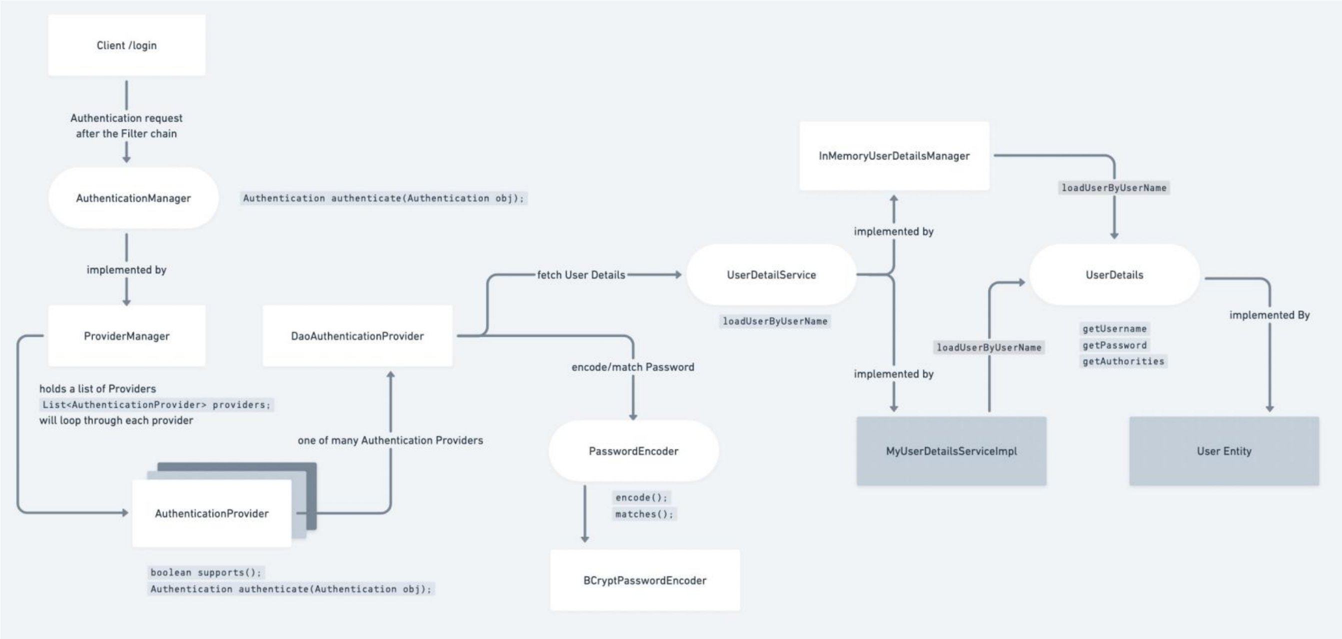
```
if (!hasFilterChain) {
    this.webSecurity.addSecurityFilterChainBuilder(() -> {
        this.httpSecurity.authorizeHttpRequests(( AuthorizationMana
        this.httpSecurity.formLogin(Customizer.withDefaults());
        this.httpSecurity.httpBasic(Customizer.withDefaults());
        return (SecurityFilterChain) this.httpSecurity.build();
    });
}
```

- F
- F
- F
- F

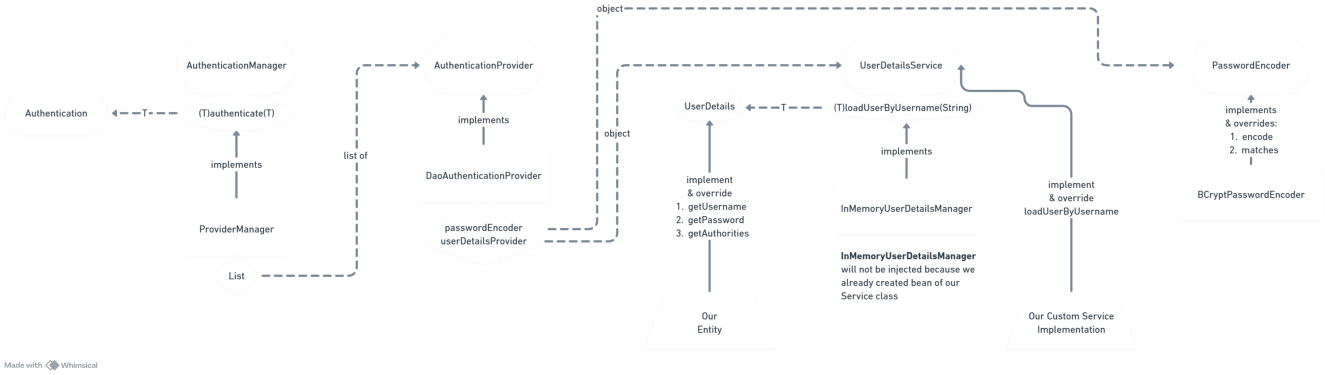
- When you run the application after including the dependency **spring-boot-starter-security**, by default one login page will appear to authenticate you.
  - If you inspect that, you'll find one *hidden input* field containing the csrf token as its value which is being attached to request.
  - 

    ```
    ▸<p>⋯</p>
     <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.
- Default Security Filters configurations
  - **SecurityFilterChain** is an interface containing list of filters.
    - **DefaultSecurityFilterChain** *implements SecurityFilterChain* and initializes the filters inside its constructor.
    - But someone needs to call this constructor passing the **filters** as argument to initialize the filters.
  - **HttpSecurity** class extends **AbstractSecurityBuilder** class which contains **build()** method which returns an object of type **DefaultSecurityFilterChain**.


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
AuthenticationProvider
DaoAuthenticationProvider
UserDetailService
PasswordEncoder
[
DaoAuthenticationProvider,
InMemoryAuthenticationProvider,
OAuth2AuthenticationProvider,
...
]



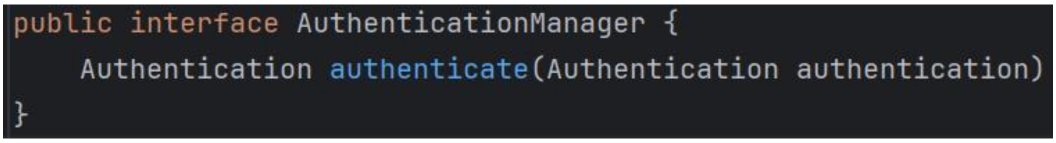
(rectangles: classes, ellipse: interfaces)



InMemoryUserDetailsManager will not be injected because we already created bean of our Service class

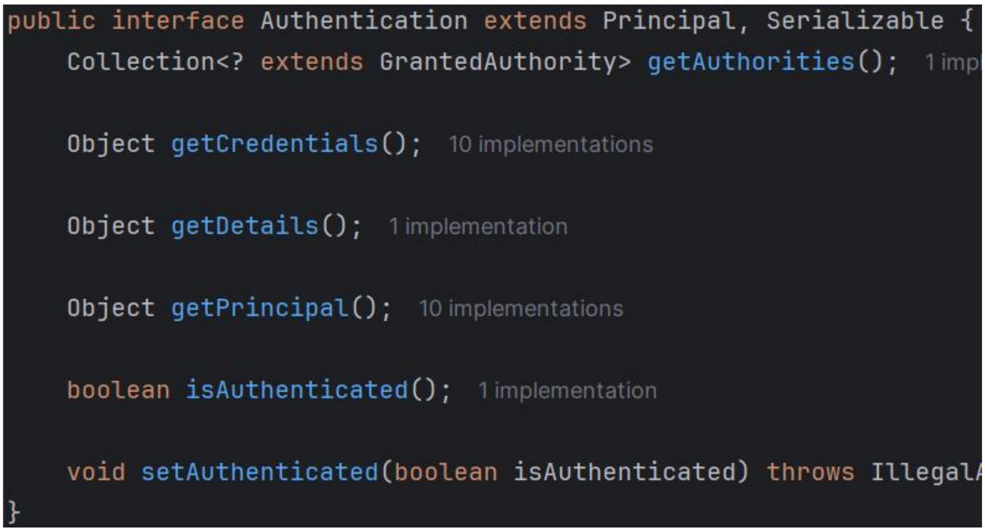
Flowchart of Authentication

- **AuthenticationManager** is an interface.

  ```
  public interface AuthenticationManager {
      Authentication authenticate(Authentication authentication)
  }
  ```

  - **Authentication** is also an interface containing necessary methods

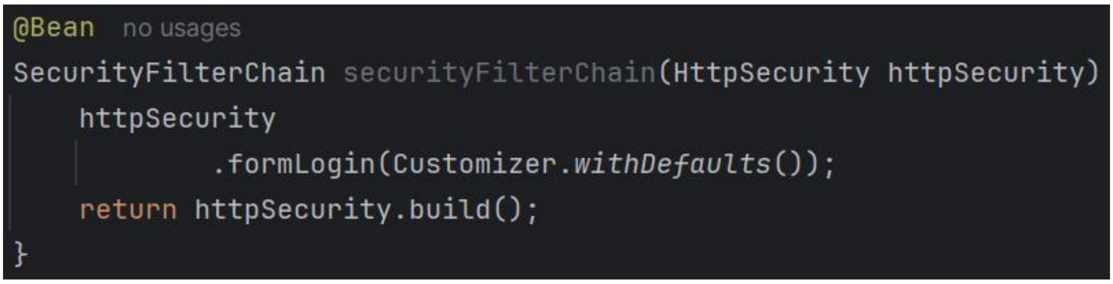
    ```
    public interface Authentication extends Principal, Serializable {
        Collection<? extends GrantedAuthority> getAuthorities();  1 impl

        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)
- **DaoAuthenticationProvider** implements **AuthenticationProvider**
  - It holds the **UserDetailsService** and **PasswordEncoder** type of objects.
- **UserDetailsService**
  - We will implement this interface from our custom **Service** class and implement the **loadUserByUsername** method.
  - This method use **UserDetails** type of object.
- **UserDetails**
  - Our entity should implement this interface.

➢ Writing custom Filter Chains

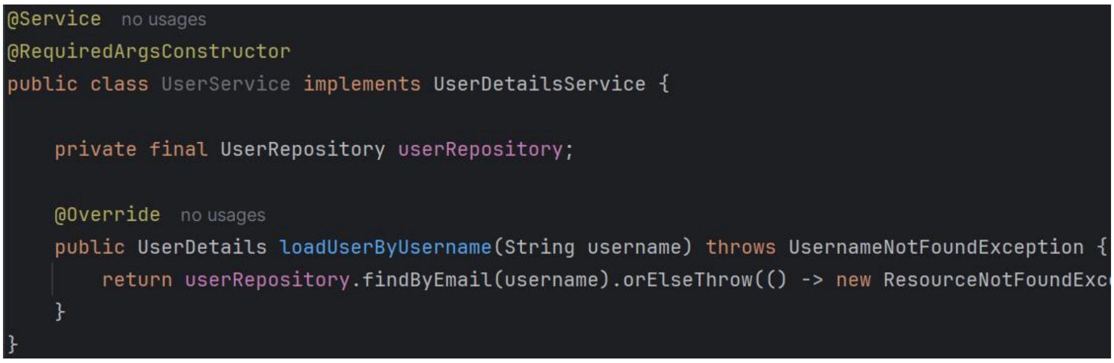
- We'll create the custom filters like the below

```
@Bean  no usages
SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity)
    httpSecurity
            .formLogin(Customizer.withDefaults());
    return httpSecurity.build();
}
```

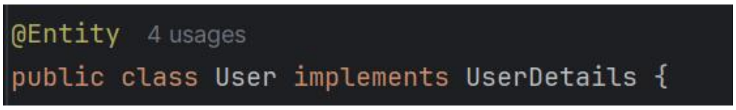
  - You might be thinking, while creating the bean, how this **HttpSecurity** type of objects is being coming; as according to what we were doing earlier, it didn't have any argument in the @Bean method.
  - You can mention any type of argument you want in the **@Bean** method, if that type of **bean** is created, then Spring will automatically inject that bean to your **@Bean** method's argument.

- Lets say we want to write our own **UserDetailsService** and **UserDetails**
  - As we know, by default **InMemoryUserDetailsManager** implements the **UserDetailsService** interface implementing the ***loadUserByUsername*** method.
    - It is responsible for authenticating the user and password (From the default login form of spring security) being mentioned in the *application.properties* file.
  - If we want to write our own service, then **InMemoryUserDetailsManager** bean will not be created; rather our ***service*** bean will be created in place of that.
  - So, to write custom user details and the service, we need to do the following:
    - **Extend the UserDetails from our User entity**
    - **Extend the UserDetailsService from our UserService**
      - In this UserDetailsService, write the implementation of the method *loadUserByUsername.*
- I wrote the following:
  - In UserService class:

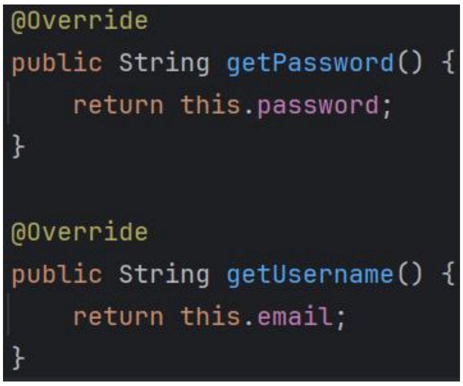


```
@Service  no usages
@RequiredArgsConstructor
public class UserService implements UserDetailsService {

    private final UserRepository userRepository;

    @Override  no usages
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return userRepository.findByEmail(username).orElseThrow(() -> new ResourceNotFoundExc
    }
}
```

  - In User entity



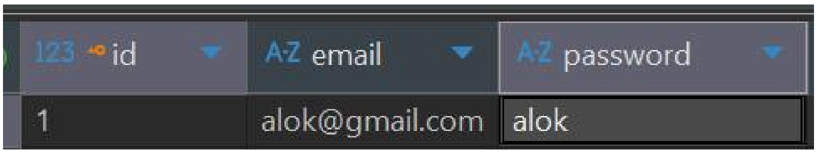
```
@Entity  4 usages
public class User implements UserDetails {
```



```
@Override
public String getPassword() {
    return this.password;
}

@Override
public String getUsername() {
    return this.email;
}
```

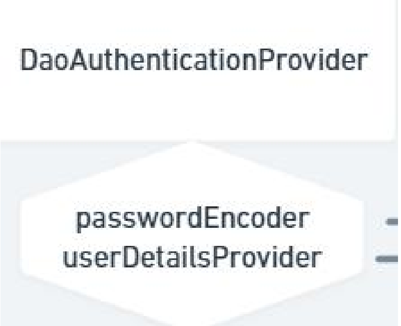
      - These methods are required as there are mentioned in UserDetails interface

  - Now, I added one dummy entry in database

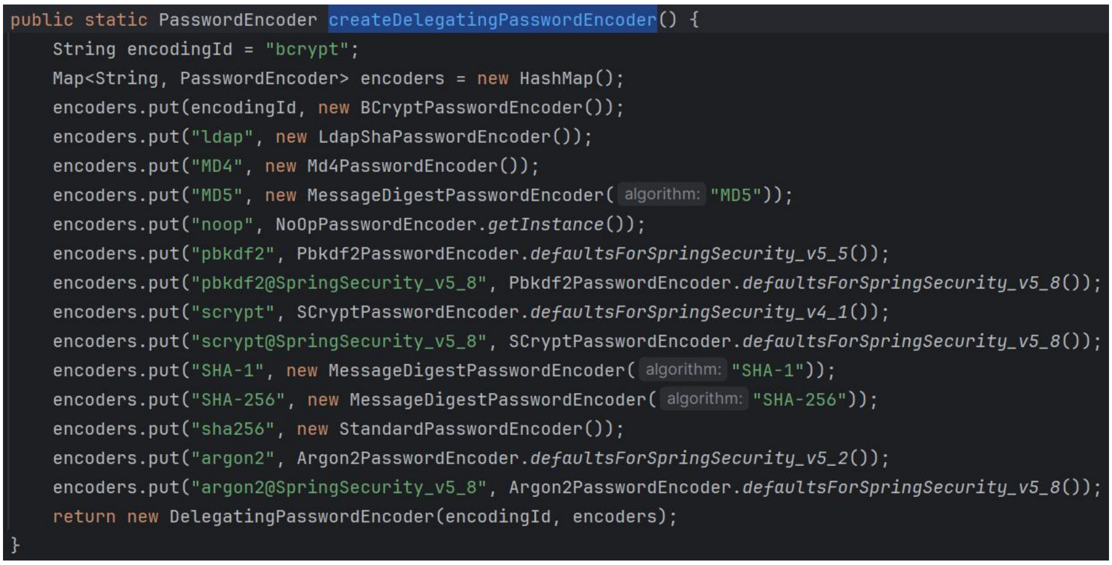
123 id	A-Z email	A-Z password
1	alok@gmail.com	alok

- Now, If I give alok@gmail.com in the username field and give some random password, it should work right? Because in the ***loadUserByUserName*** method I am only fetching the user by username (which is username in our case) and not checking the password.
  - But it'll not work :)
  - If you remember the diagram:

  - We have 2 things here, one is userDetailsProvider (of type UserDetailsService) and passwordEncoder (of type PasswordEncoder)
    - It first get the object returned from the method **loadUserByUserName**.
    - Then it calls the methods: **getUsername** and **getPassword**
    - Now, it validate the password by itself (via the specific hashing algorithm)
    - So, **loadUserByUsername** is just used to get the object of type **UserDetails**; remaining verification are done separately.
  - Spring Security uses **UserDetailsService** only to load a **UserDetails** object via **loadUserByUsername()**, then independently calls **getUsername()** and **getPassword()** on it and validates the login password using **PasswordEncoder.matches()** with the configured hashing algorithm.

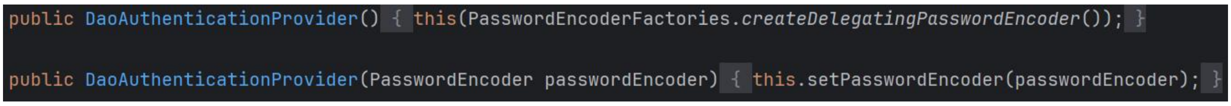
- **PasswordEncoderFactories.class** contains **createDelegatingPasswordEncoder** method which creates the object of password encoders like bcrypt, MD4, MD5 ..etc; there is no configuration file to create bean of any specific Password encoder.



```
public static PasswordEncoder createDelegatingPasswordEncoder() {
    String encodingId = "bcrypt";
    Map<String, PasswordEncoder> encoders = new HashMap();
    encoders.put(encodingId, new BCryptPasswordEncoder());
    encoders.put("ldap", new LdapShaPasswordEncoder());
    encoders.put("MD4", new Md4PasswordEncoder());
    encoders.put("MD5", new MessageDigestPasswordEncoder( algorithm: "MD5"));
    encoders.put("noop", NoOpPasswordEncoder.getInstance());
    encoders.put("pbkdf2", Pbkdf2PasswordEncoder.defaultsForSpringSecurity_v5_5());
    encoders.put("pbkdf2@SpringSecurity_v5_8", Pbkdf2PasswordEncoder.defaultsForSpringSecurity_v5_8());
    encoders.put("scrypt", SCryptPasswordEncoder.defaultsForSpringSecurity_v4_1());
    encoders.put("scrypt@SpringSecurity_v5_8", SCryptPasswordEncoder.defaultsForSpringSecurity_v5_8());
    encoders.put("SHA-1", new MessageDigestPasswordEncoder( algorithm: "SHA-1"));
    encoders.put("SHA-256", new MessageDigestPasswordEncoder( algorithm: "SHA-256"));
    encoders.put("sha256", new StandardPasswordEncoder());
    encoders.put("argon2", Argon2PasswordEncoder.defaultsForSpringSecurity_v5_2());
    encoders.put("argon2@SpringSecurity_v5_8", Argon2PasswordEncoder.defaultsForSpringSecurity_v5_8());
    return new DelegatingPasswordEncoder(encodingId, encoders);
}
```

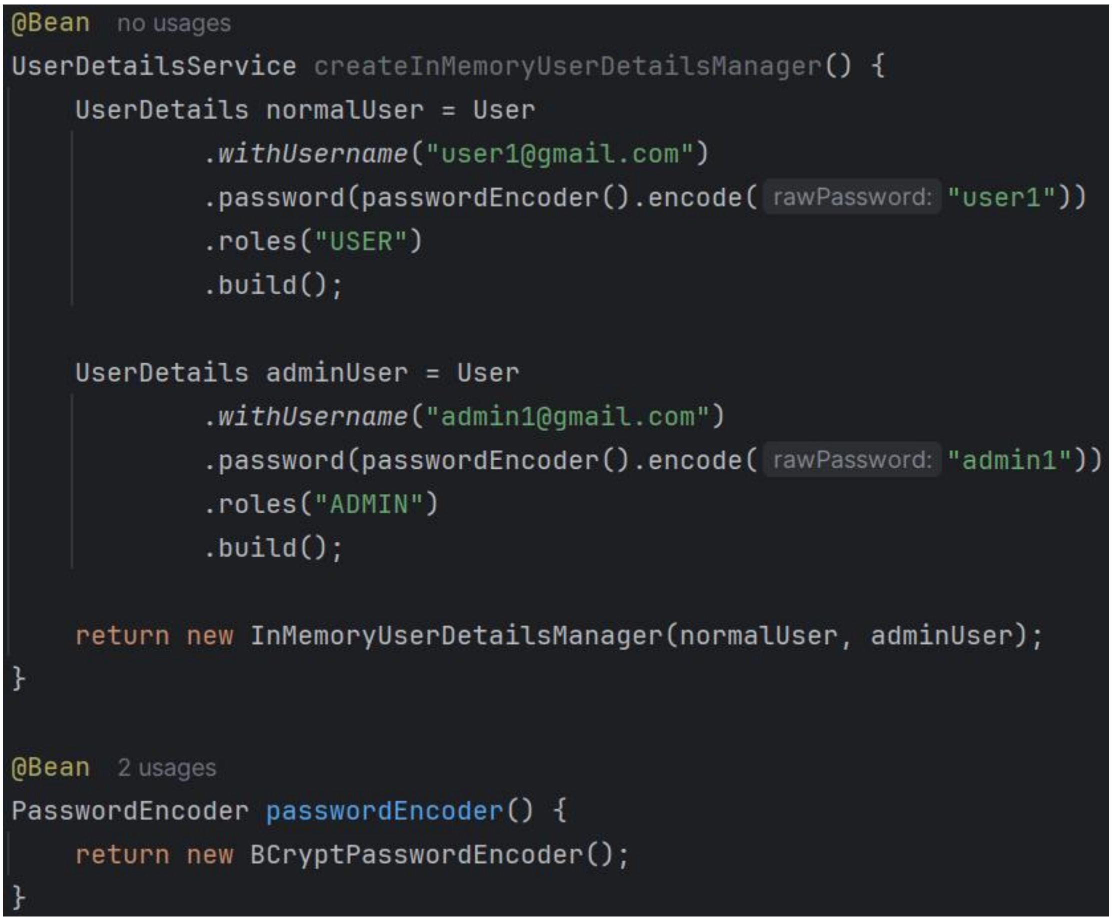
(PasswordEncoderFactories.class)

- **DaoAuthenticationProvider** uses this PasswordEncoderFactories to validate the password.



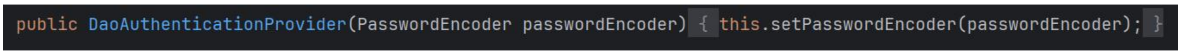
```
public DaoAuthenticationProvider() { this(PasswordEncoderFactories.createDelegatingPasswordEncoder()); }

public DaoAuthenticationProvider(PasswordEncoder passwordEncoder) { this.setPasswordEncoder(passwordEncoder); }
```

- For testing purpose of users you can create **InMemoryUserDetailsManager** bean by yourself (make sure to not create bean of any service that extends UserDetailsService)



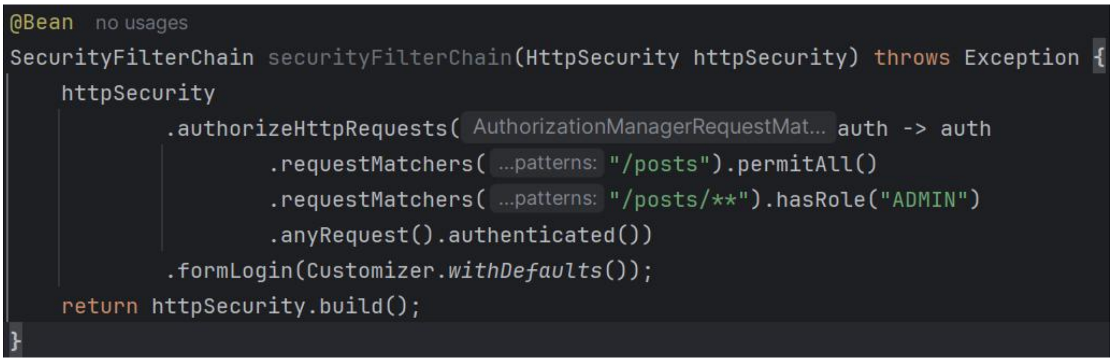
```
@Bean  no usages
UserDetailsService createInMemoryUserDetailsManager() {
    UserDetails normalUser = User
            .withUsername("user1@gmail.com")
            .password(passwordEncoder().encode( rawPassword: "user1"))
            .roles("USER")
            .build();

    UserDetails adminUser = User
            .withUsername("admin1@gmail.com")
            .password(passwordEncoder().encode( rawPassword: "admin1"))
            .roles("ADMIN")
            .build();

    return new InMemoryUserDetailsManager(normalUser, adminUser);
}

@Bean  2 usages
PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

- If you create a bean of **PasswordEncoder** then it'll automatically be injected to the constructor of **DaoAuthenticationProvider** .



```
public DaoAuthenticationProvider(PasswordEncoder passwordEncoder) { this.setPasswordEncoder(passwordEncoder); }
```

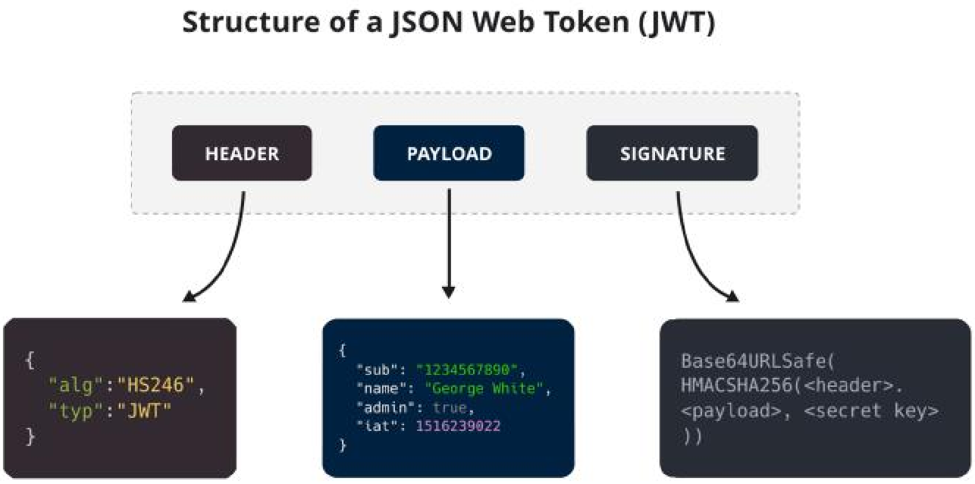
- Also I restricted some paths for users; only admins can see those;



```
@Bean  no usages
SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
    httpSecurity
            .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
                    .requestMatchers( ...patterns: "/posts").permitAll()
                    .requestMatchers( ...patterns: "/posts/**").hasRole("ADMIN")
                    .anyRequest().authenticated())
            .formLogin(Customizer.withDefaults());
    return httpSecurity.build();
}
```

- This is how you can manage the routes;

- **JWT**

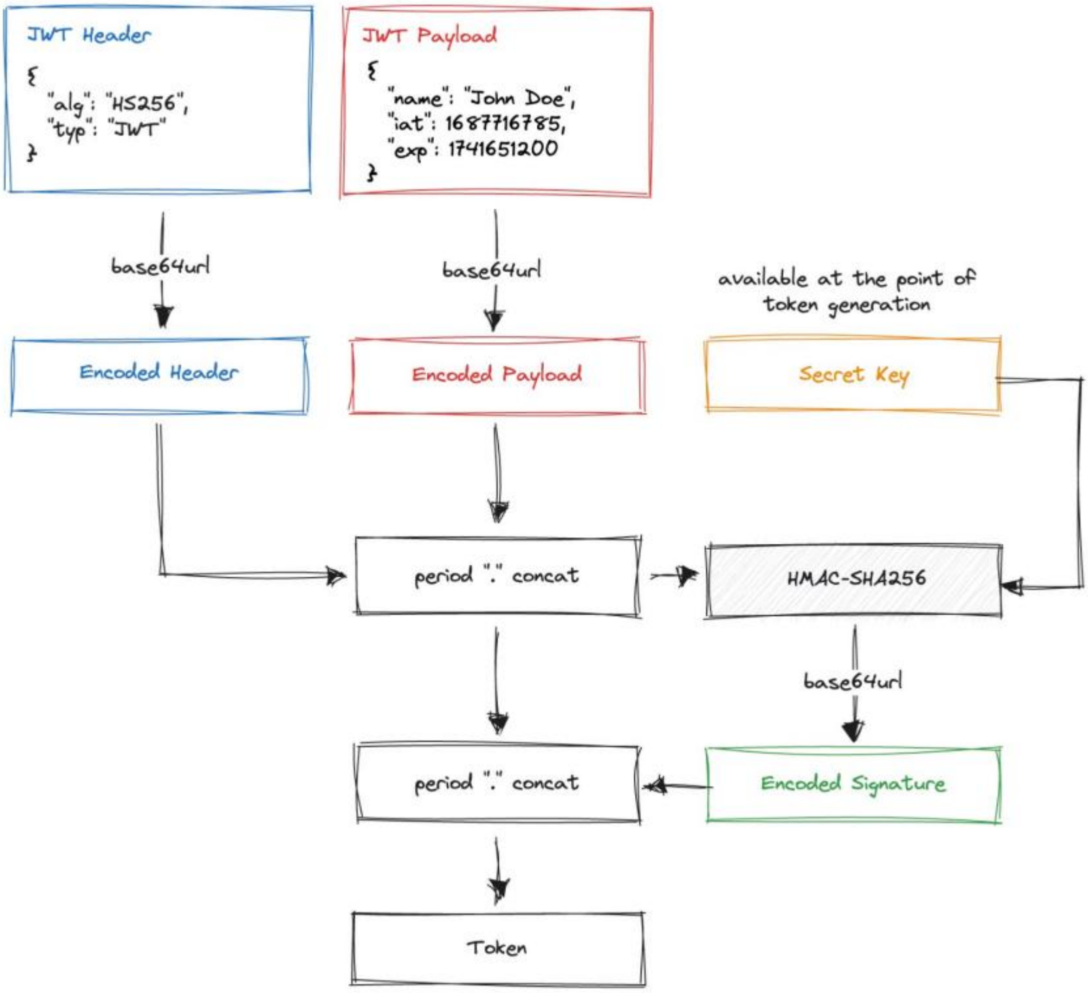


```
{
  "alg":"HS246",
  "typ":"JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "George White",
  "admin": true,
  "iat": 1516239022
}
```

```
Base64URLSafe(
HMACSHA256(<header>.
<payload>, <secret key>
))
```

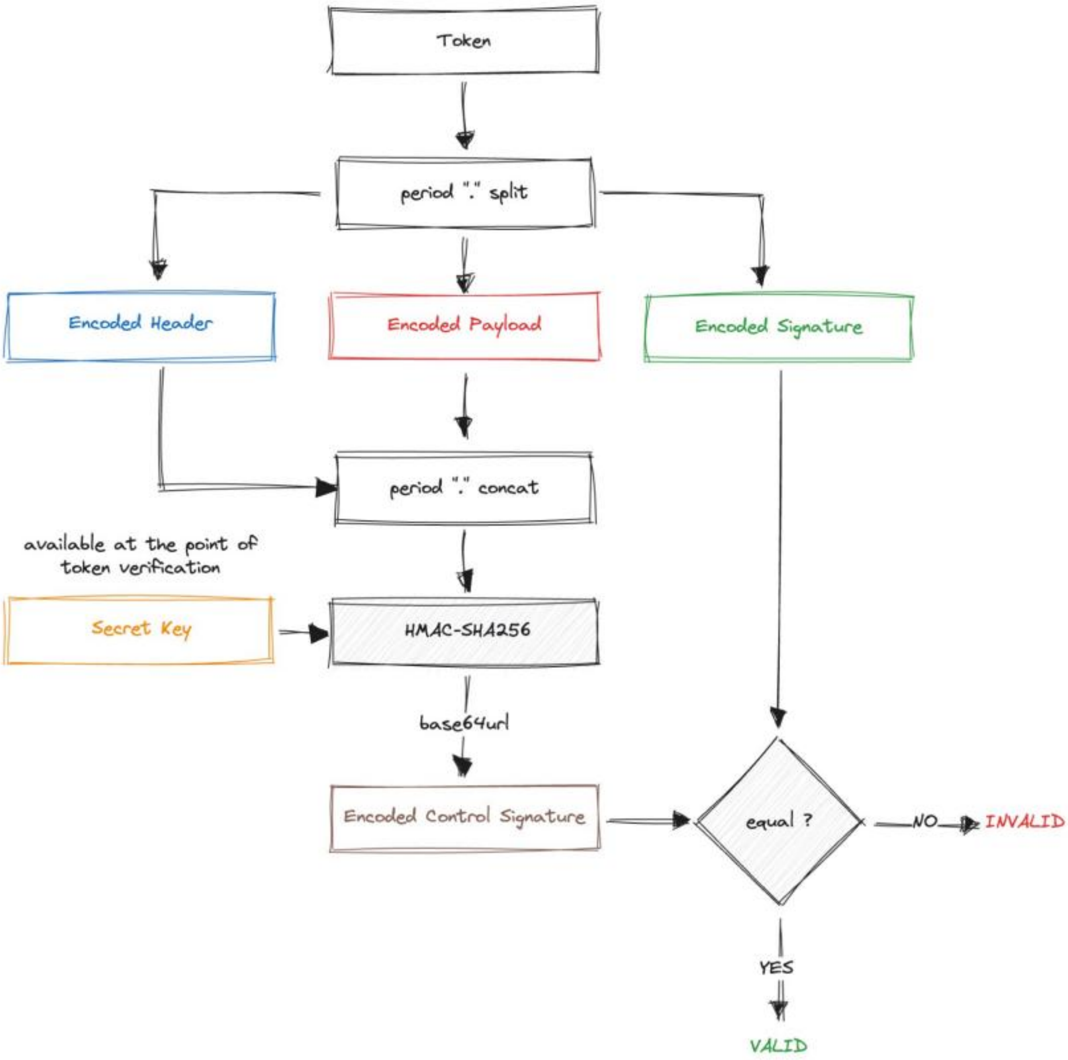
-



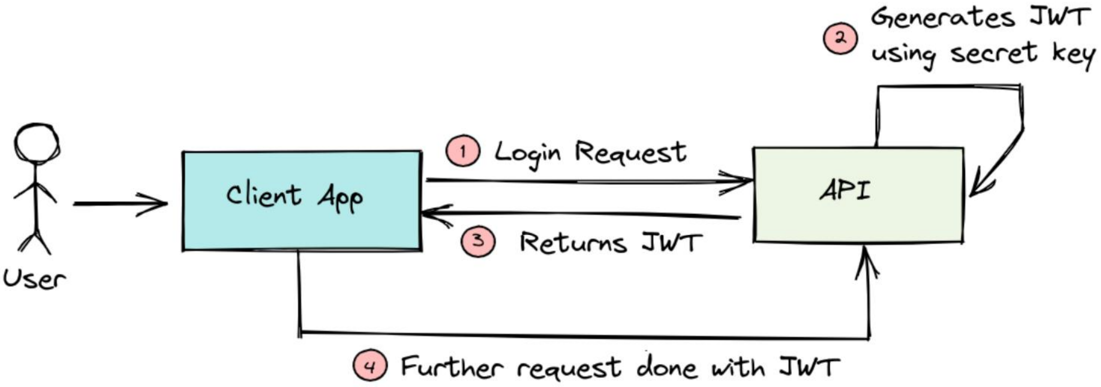
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "name": "John Doe",
  "iat": 1687716785,
  "exp": 1741651200
}
```

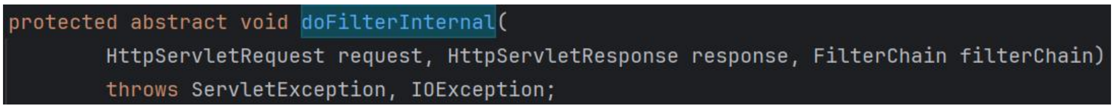
-


Token
period "." split
Encoded Header
Encoded Payload
Encoded Signature
period "." concat
available at the point of token verification
Secret Key
HMAC-SHA256
base64url
Encoded Control Signature
equal ?
NO
INVALID
YES
VALID

-

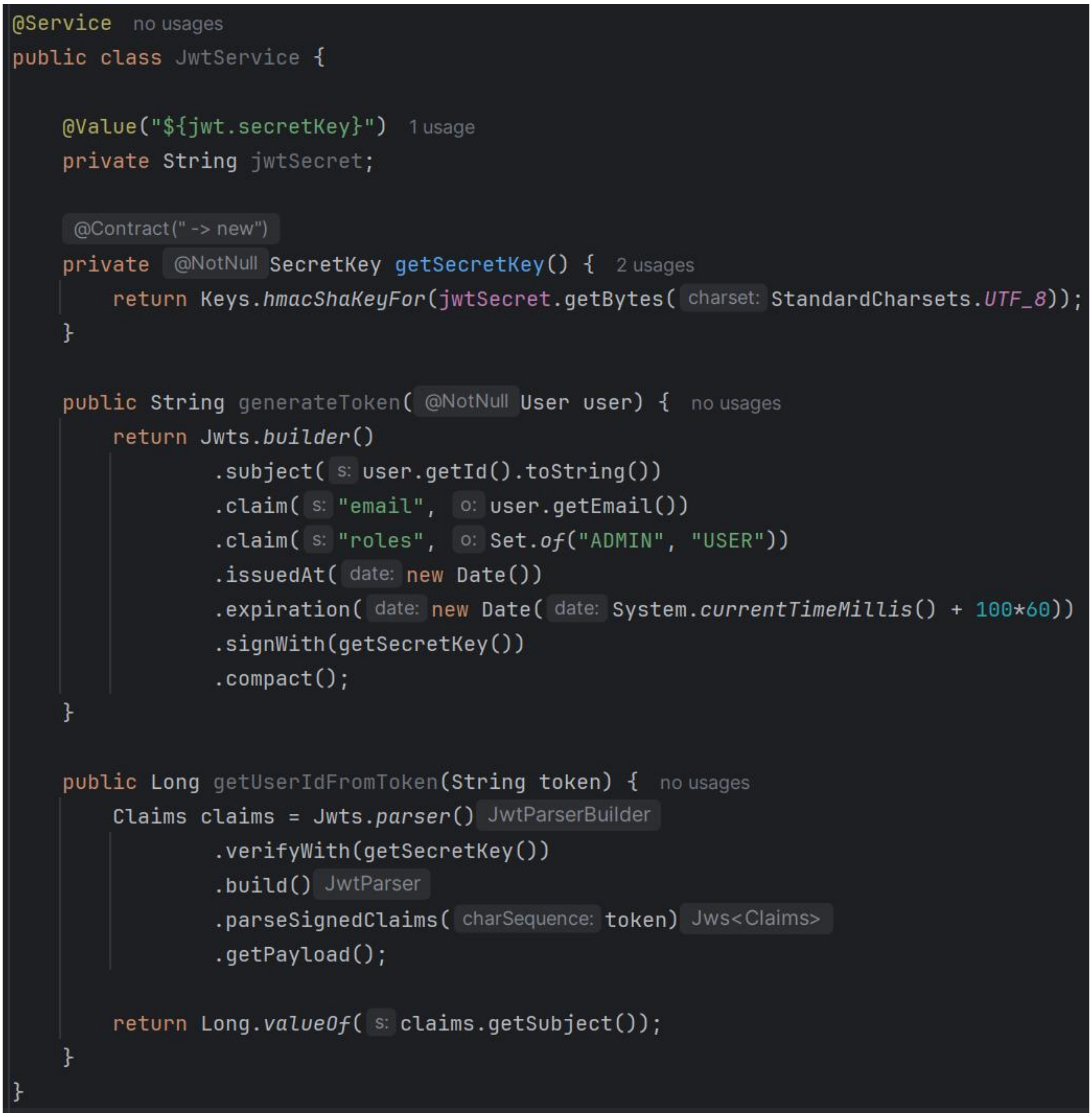

2 Generates JWT using secret key
1 Login Request
Client App
API
User
3 Returns JWT
4 Further request done with JWT

-
- F
- F
- F
- F

- There are 2 things:
    - **FilterChain**
        - Provided by Servlet container (Tomcat)
    - **SecurityFilterChain**
        - Provided by Spring Security.
        - It is just a callback handle to **"who comes next"**
- **FilterChain** and **SecurityFilterChain** looks similar but both are different.
    - FilterChain is like a middleware that is used to connect the filters of the SecurityFilterChain.
    - Its just a callback, means if one filter passes then FilterChain will be the one who'll call the next filter; if fails then it stops.
    - There is a method **doFilter** inside the **FilterChain** interface, which is being called in each filter.
    - 

```java
protected abstract void doFilterInternal(
        HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException;
```

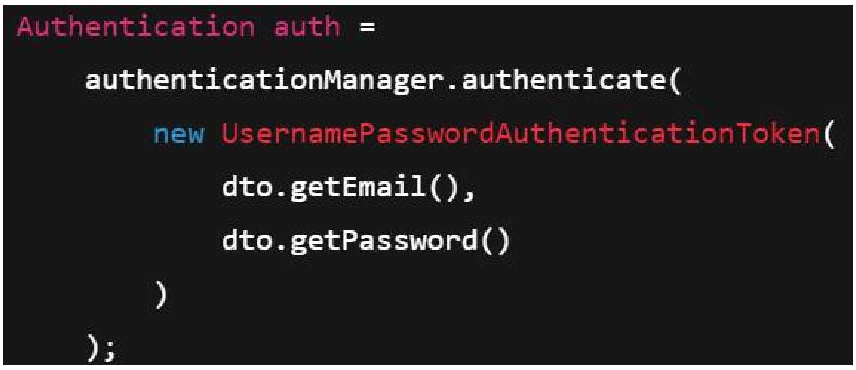
Creating *JwtService*



```
@Service  no usages
public class JwtService {

    @Value("${jwt.secretKey}")  1 usage
    private String jwtSecret;

    @Contract(" -> new")
    private  @NotNull SecretKey getSecretKey() {  2 usages
        return Keys.hmacShaKeyFor(jwtSecret.getBytes( charset: StandardCharsets.UTF_8));
    }

    public String generateToken( @NotNull User user) {  no usages
        return Jwts.builder()
                .subject( s: user.getId().toString())
                .claim( s: "email",  o: user.getEmail())
                .claim( s: "roles",  o: Set.of("ADMIN", "USER"))
                .issuedAt( date: new Date())
                .expiration( date: new Date( date: System.currentTimeMillis() + 100*60))
                .signWith(getSecretKey())
                .compact();
    }

    public Long getUserIdFromToken(String token) {  no usages
        Claims claims = Jwts.parser() JwtParserBuilder
                .verifyWith(getSecretKey())
                .build() JwtParser
                .parseSignedClaims( charSequence: token) Jws<Claims>
                .getPayload();

        return Long.valueOf( s: claims.getSubject());
    }
}
```

- **subject :** it defines the owner; usually give the unique key as subject. **getPayload()**
- **claims :** it is just some additional details that you can give. **get("claim_key_name")**

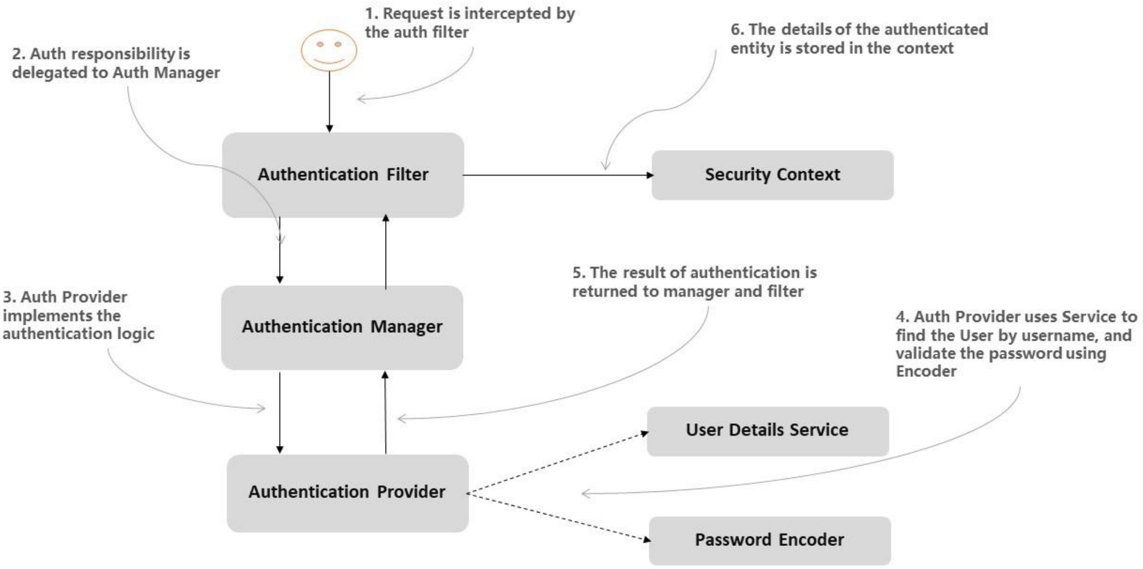
Important Points

- JWT filter is used for authentication; not authorization;
- For **login**, you don't want the user to be authenticated:
  - Either you can use **form login** (which is not recommended), it uses **UsernamePasswordAuthenticationFilter** by default.
  - Or you can directly go to **controller** and there you can check the username and password;
    - If the credentials are proper then generate one token and return.
    - NOTE: You don't need to verify the username and password manually; use **AuthenticationManager** for this.
    - 

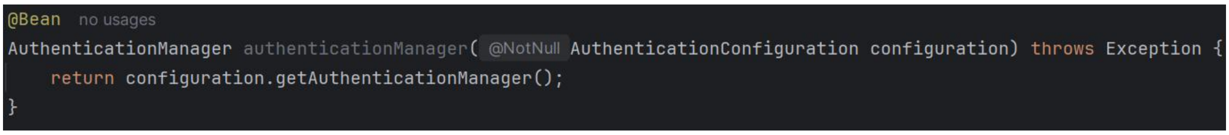
      ```
      Authentication auth =
          authenticationManager.authenticate(
              new UsernamePasswordAuthenticationToken(
                  dto.getEmail(),
                  dto.getPassword()
              )
          );
      ```
    - One thing might be confusing that how does it go to database and get the hashed password and compare the **dto** password with that;
    - Here, **authenticationManager.authenticate()** delegates the call to **authenticationProvider.authenticate()** , and this **retrieve** the user details calling the method **loadUserByUsername** that is being written by us in the **service** class that extends **UserDetailsService.**
  - **tomcat --- controller --- authentication manager**
- For accessing any **protected** path (paths which are being mentioned with **authenticated()**)
  - Request will come with the **authorization header** containing the **JWT** token.
  - It'll then go to **JWT filter** (written manually)
  - Here it'll validate the token, and store the **authentication** in the **SecurityContext.**
  - **SecurityContext** is accessible in the current request thread to get the details of current authenticated user.
  -
- F
- F

➢ NOTE

- You don’t need to create bean of **AuthenticationManager** manually, if you are creating bean of **PasswordEncoder** and **UserDetailsService**, spring will automatically create AuthenticationManager using **DaoAuthenticationProvider** as the AuthenticationProvider.
- Because **UserDetailsService** is only related to the **DaoAuthenticationProvider** and not related to any of the other providers.


1. Request is intercepted by the auth filter
2. Auth responsibility is delegated to Auth Manager
6. The details of the authenticated entity is stored in the context
Authentication Filter
Security Context
5. The result of authentication is returned to manager and filter
3. Auth Provider implements the authentication logic
Authentication Manager
4. Auth Provider uses Service to find the User by username, and validate the password using Encoder
User Details Service
Authentication Provider
Password Encoder

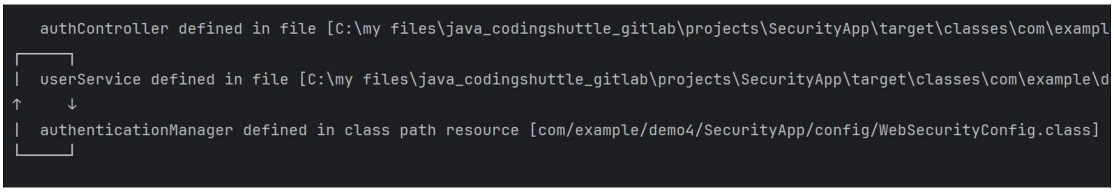
- It’ll be **true only if** you have **not done** any of the following: otherwise you need to **create** the **bean of AuthenticationManager** by yourself.
  - ❌ Create your own SecurityFilterChain
  - ❌ Customize HttpSecurity
  - ❌ Add JWT / custom filter
  - ❌ Disable formLogin / httpBasic
  - ❌ Use stateless session
  - ❌ Create custom authentication logic



```
@Bean  no usages
AuthenticationManager authenticationManager( @NotNull AuthenticationConfiguration configuration) throws Exception {
    return configuration.getAuthenticationManager();
}
```

Cyclic Dependency

- Sometimes you might get the following type of Exception



```
   authController defined in file [C:\my files\java_codingshuttle_gitlab\projects\SecurityApp\target\classes\com\exampl
┌─────┐
|  userService defined in file [C:\my files\java_codingshuttle_gitlab\projects\SecurityApp\target\classes\com\example\d
↑     ↓
|  authenticationManager defined in class path resource [com/example/demo4/SecurityApp/config/WebSecurityConfig.class]
└─────┘
```

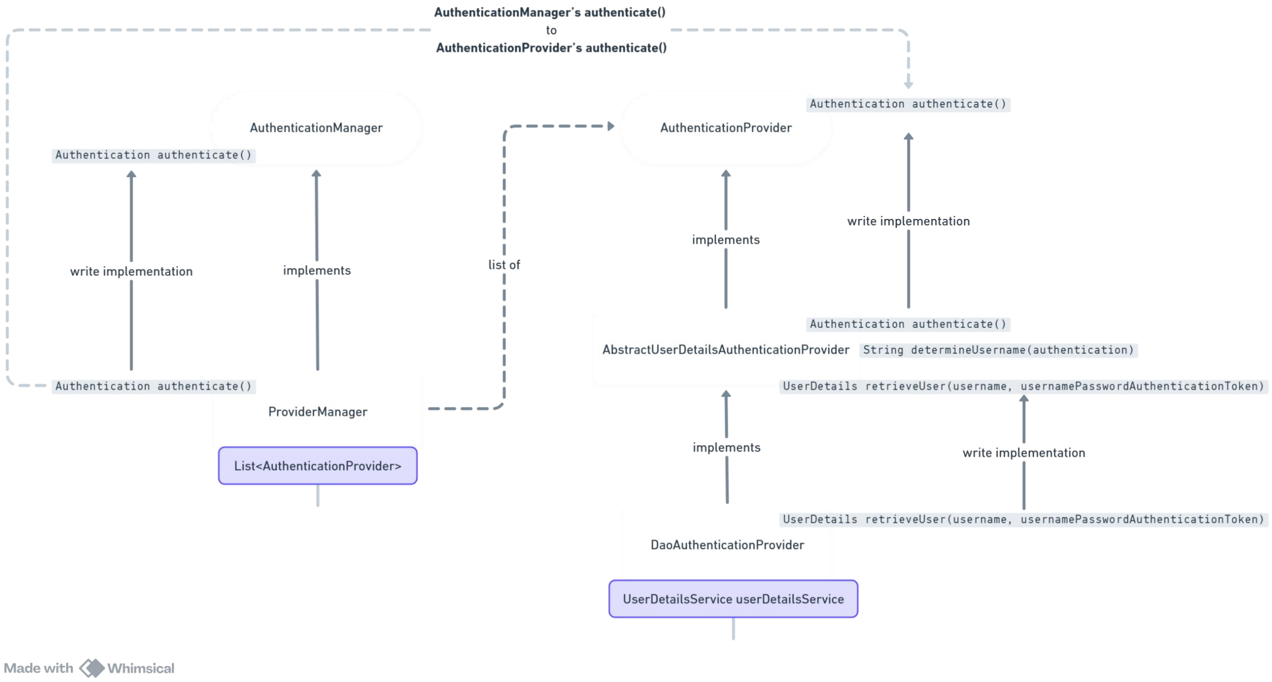
- You need to avoid this type of cyclic dependencies.
- Here I got this exception because,
  - **AuthenticationManager** uses **UserServiceDetails** which uses **UserService**
  - And I am using **AuthenticationManager** inside **UserService**.
  - AuthenticationManager -- UserService -- AuthenticationManager -- UserService -- -- -- -- -- --

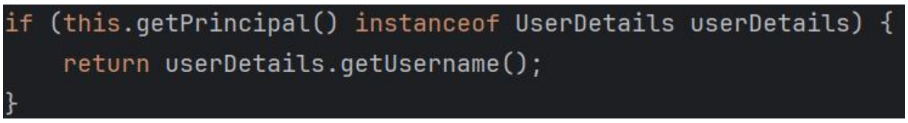
httpSecurity's **authentication()** vs **JWT Filter** --

- It is confusing that, if we are already checking the token with JWT Filter then why do we even write **authenticated()** on the protected routes.
- **authenticated()** checks **SecurityContext**
  - if it contains **Authentication** object: user is ***authenticated***;
  - Else: user is ***not authenticated***.
- ---
- In JWT Filter, we check the token and add the user details in the **SecurityContext**, so Spring Security now knows that user is authenticated.

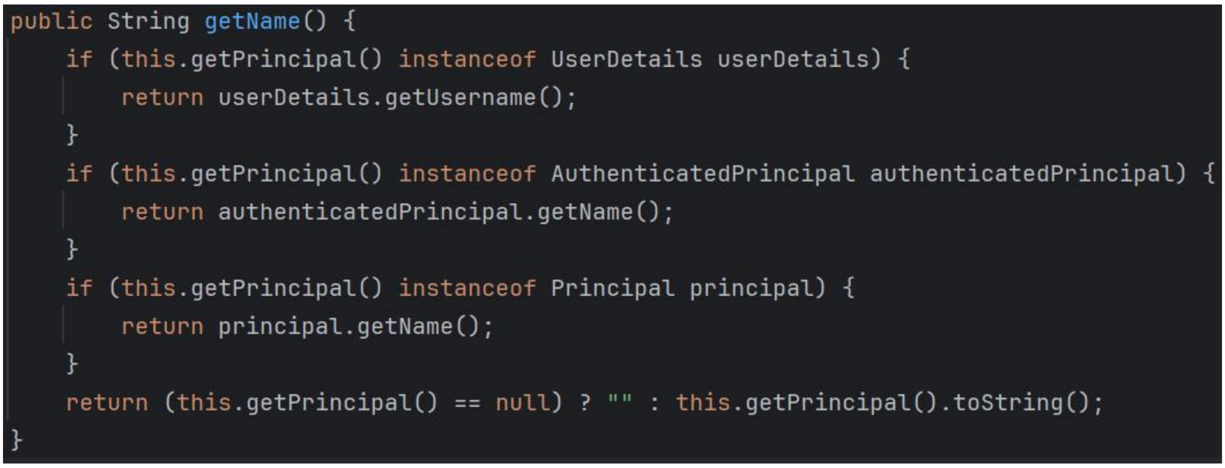
Principal vs Credentials vs Authorities in Authentication Object

Principal -------


AuthenticationManager's authenticate()
to
AuthenticationProvider's authenticate()
AuthenticationManager
Authentication authenticate()
write implementation
implements
Authentication authenticate()
ProviderManager
List<AuthenticationProvider>
list of
AuthenticationProvider
Authentication authenticate()
implements
write implementation
Authentication authenticate()
AbstractUserDetailsAuthenticationProvider
String determineUsername(authentication)
UserDetails retrieveUser(username, usernamePasswordAuthenticationToken)
implements
write implementation
UserDetails retrieveUser(username, usernamePasswordAuthenticationToken)
DaoAuthenticationProvider
UserDetailsService userDetailsService
Made with Whimsical

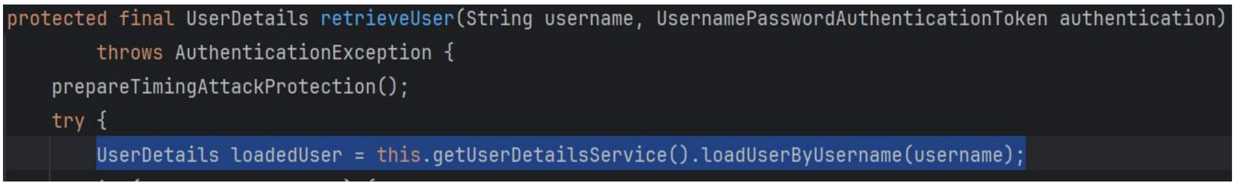
- If you pass **UserDetails** type of object, then it must be having **getUsername()** method.
  - 

```java
if (this.getPrincipal() instanceof UserDetails userDetails) {
    return userDetails.getUsername();
}
```

  - AbstractAuthenticationToken.class
- *AuthenticationProvider*'s **authenticate()** calls **determineUsername()**
- **determineUsername()** calls *authentication*'s **getName()**
- Inside **getName()**, it'll check the type of *principal*
  - if it is UserDetails type then it'll call **loadByUserName()** method;
  - if it is **String**, then it'll return it simply.
  - 

```java
public String getName() {
    if (this.getPrincipal() instanceof UserDetails userDetails) {
        return userDetails.getUsername();
    }
    if (this.getPrincipal() instanceof AuthenticatedPrincipal authenticatedPrincipal) {
        return authenticatedPrincipal.getName();
    }
    if (this.getPrincipal() instanceof Principal principal) {
        return principal.getName();
    }
    return (this.getPrincipal() == null) ? "" : this.getPrincipal().toString();
}
```

- after getting username, it calls the **retrieveUser()** method to get the object of type ***UserDetails***.
- *DaoAuthenticationProvider* keeps the bean of type ***UserDetails*** to call the **loadUserByUserName()** method passing the username.

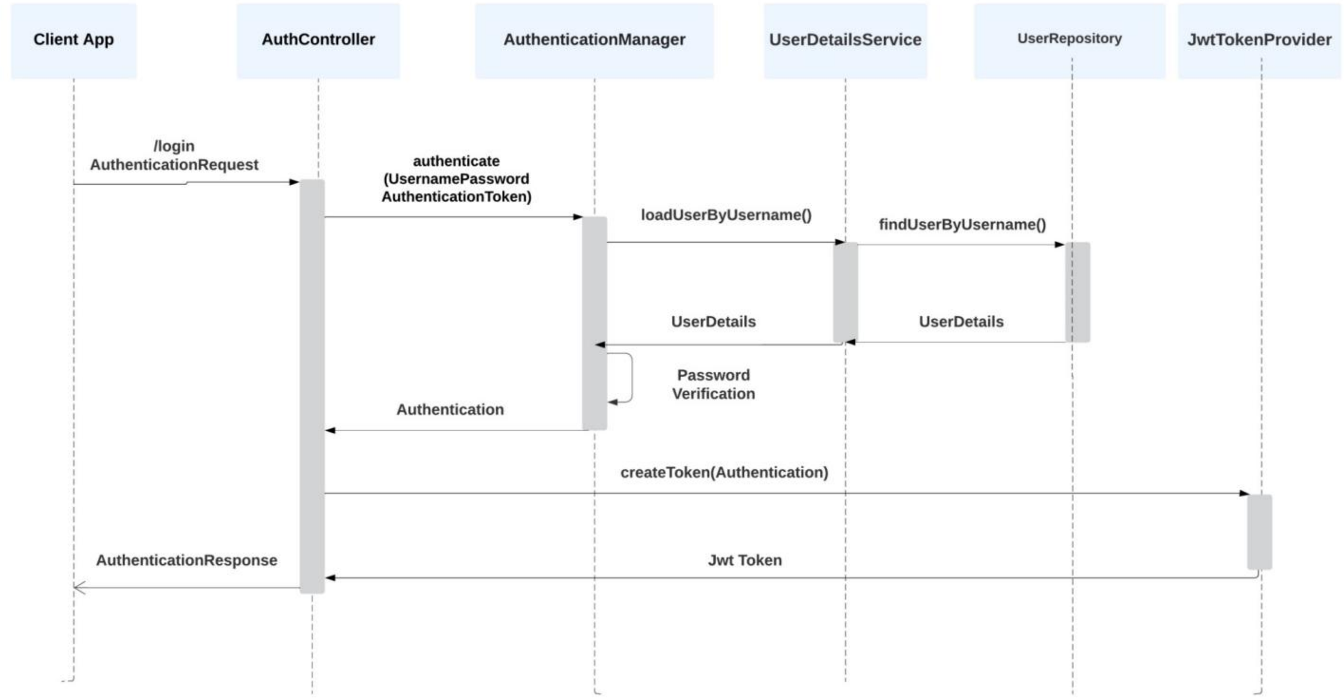


```
protected final UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken authentication)
        throws AuthenticationException {
    prepareTimingAttackProtection();
    try {
        UserDetails loadedUser = this.getUserDetailsService().loadUserByUsername(username);
```

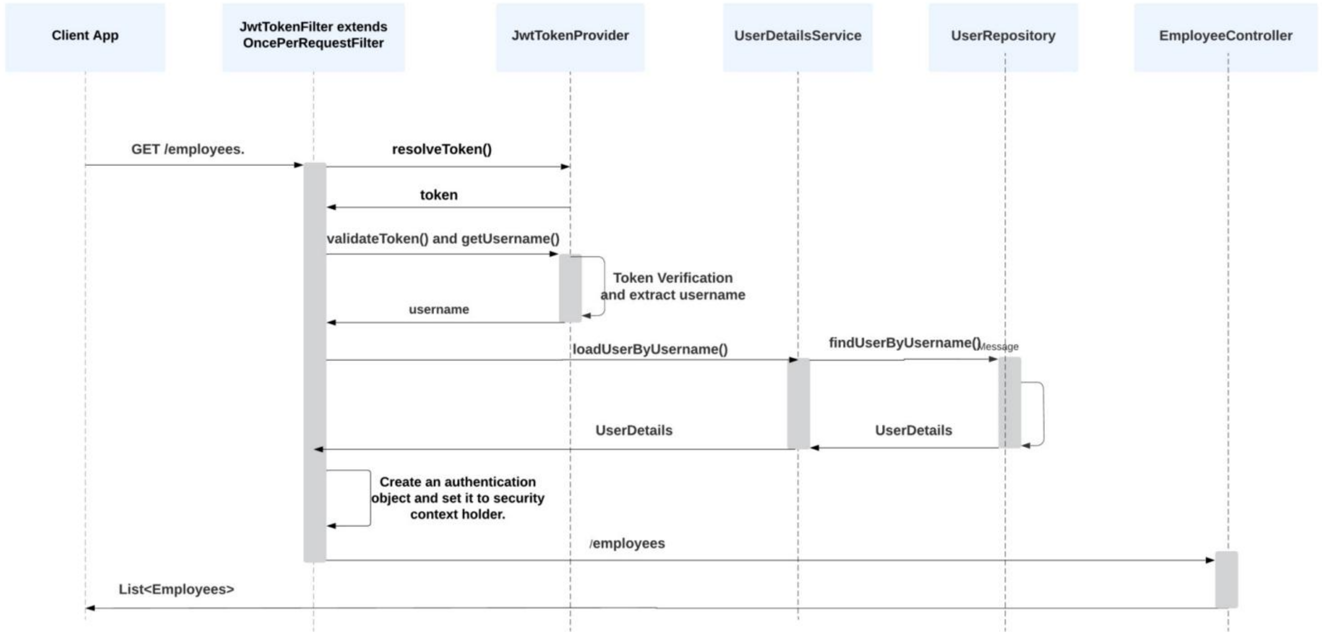
-

- **Credentials -------**
- Here you'll pass the **raw password** (not the **hashed password**)
- **Authorities -------**
-

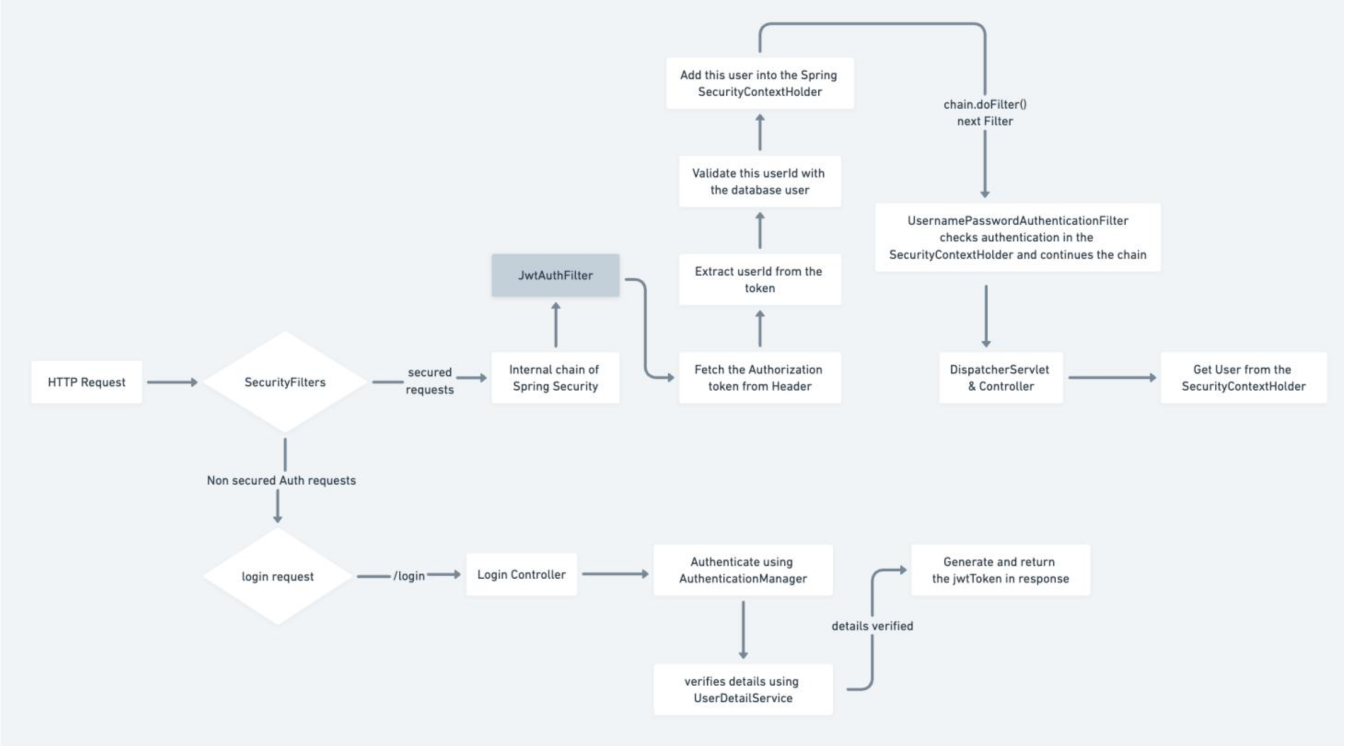
➢ Login Flow with JWT



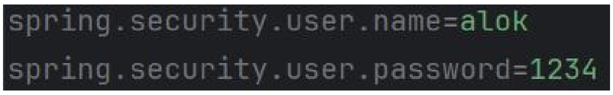
➢ Authentication Workflow with JWT



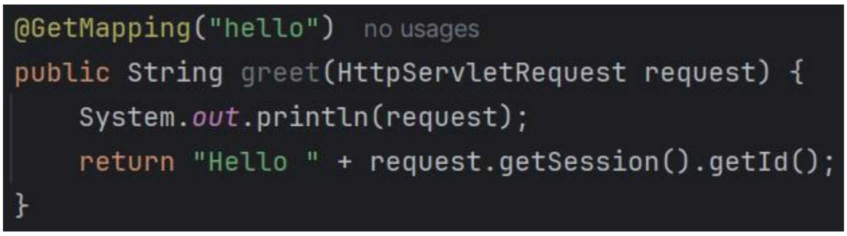
- **JWTAuthFilter Control Flow**


HTTP Request
SecurityFilters
secured requests
Internal chain of Spring Security
JwtAuthFilter
Fetch the Authorization token from Header
Extract userId from the token
Validate this userId with the database user
Add this user into the Spring SecurityContextHolder
chain.doFilter() next Filter
UsernamePasswordAuthenticationFilter checks authentication in the SecurityContextHolder and continues the chain
DispatcherServlet & Controller
Get User from the SecurityContextHolder
Non secured Auth requests
login request
/login
Login Controller
Authenticate using AuthenticationManager
verifies details using UserDetailService
details verified
Generate and return the jwtToken in response

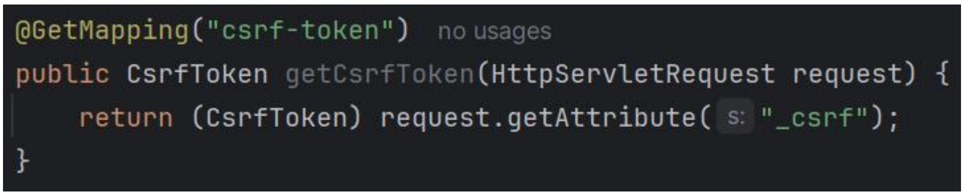
Udemy Part

- **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.
    - 

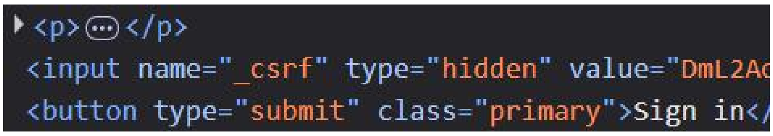
```
spring.security.user.name=alok
spring.security.user.password=1234
```

- 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();
}
```

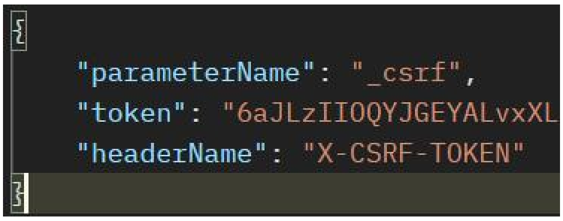
- Get request working; post not working because we are not sending csrf token;
- When you trigger get request, **csrf** token is not required because its read-only;
  - But, for remaining operations, you need to pass **csrf** token in the *headers.*
  - 

```
@GetMapping("csrf-token")  no usages
public CsrfToken getCsrfToken(HttpServletRequest request) {
    return (CsrfToken) request.getAttribute( s: "_csrf");
}
```

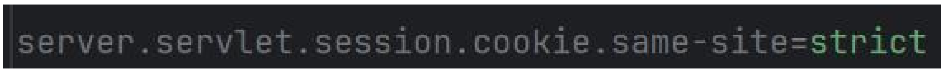
  - I created this end-point to get the **csrf** token.
  - 

```
▸<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**
        - 

```
{
    "parameterName": "_csrf",
    "token": "6aJLzIIOQYJGEYALvxXL
    "headerName": "X-CSRF-TOKEN"
}
```

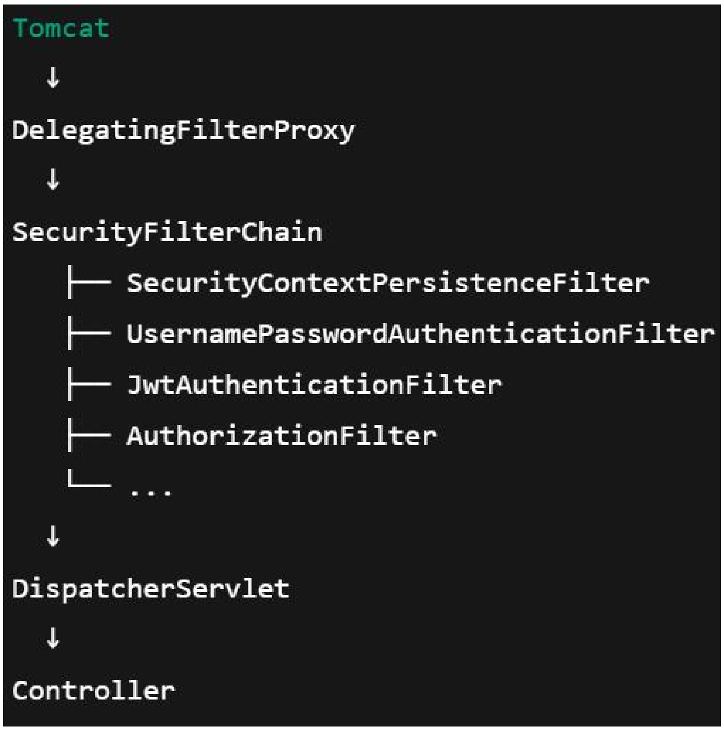
    - While triggering POST request via *postman*, you need to pass **X-CSRF-TOKEN** in the headers.
    - It is one way of solving CSRF issue; other way is "Don't allow any other website to use your session ID"
        - 

```
server.servlet.session.cookie.same-site=strict
```

        - 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**’



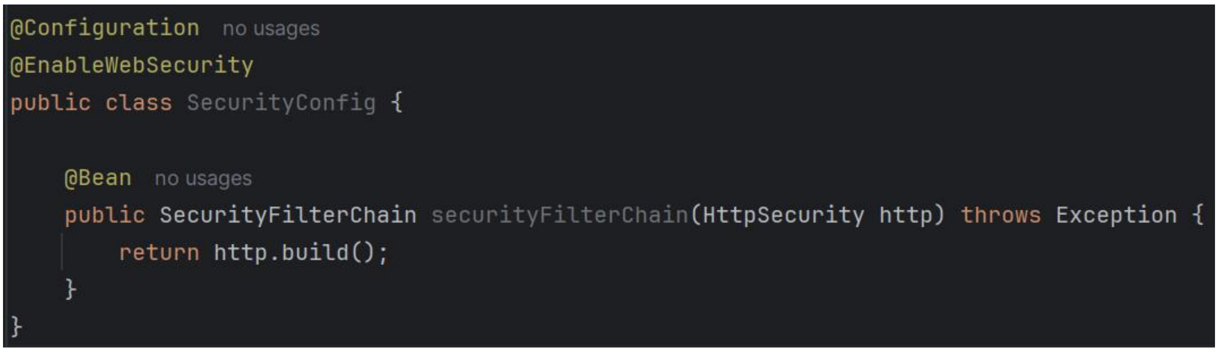
```
Tomcat
  ↓
DelegatingFilterProxy
  ↓
SecurityFilterChain
    ├── SecurityContextPersistenceFilter
    ├── UsernamePasswordAuthenticationFilter
    ├── JwtAuthenticationFilter
    ├── AuthorizationFilter
    └── ...
  ↓
DispatcherServlet
  ↓
Controller
```

➢ **@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.