By Default, Spring boot security secure all routes. We need to use annotation **@EnableGlobalMethodSecurity(prePostEnabled = true)** to enable the method level security and **@EnableWebSecurity** to enable spring security configuration class.

**Secured Vs PreAuthorized:**

Simply, *@PreAuthorize* (preferred) is newer than *@Secured.* The real difference is that @PreAuthorize can work with Spring Expression Language (**SpEL**). So, we can

* Access methods and properties of **SecurityExpressionRoot (**this class has methods such as *getPrinciple(), getAuthenticatio(), hasRole(), hasAuthority()* etc.**)**.
* Access method arguments (requires compilation with debug info or custom **ParameterNameDiscoverer**)



* Preauthorized can use AND, OR, NOT operation. Secured only has OR by default. For example-





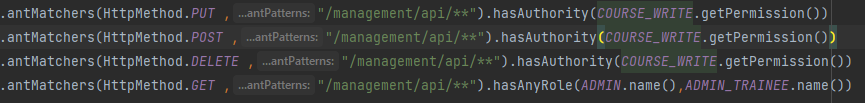


**AntMatcher:** It is used to configure the URL paths from which Spring Security should permit requests based on the user’s roles. It has 3 overloads

* Takes the path only



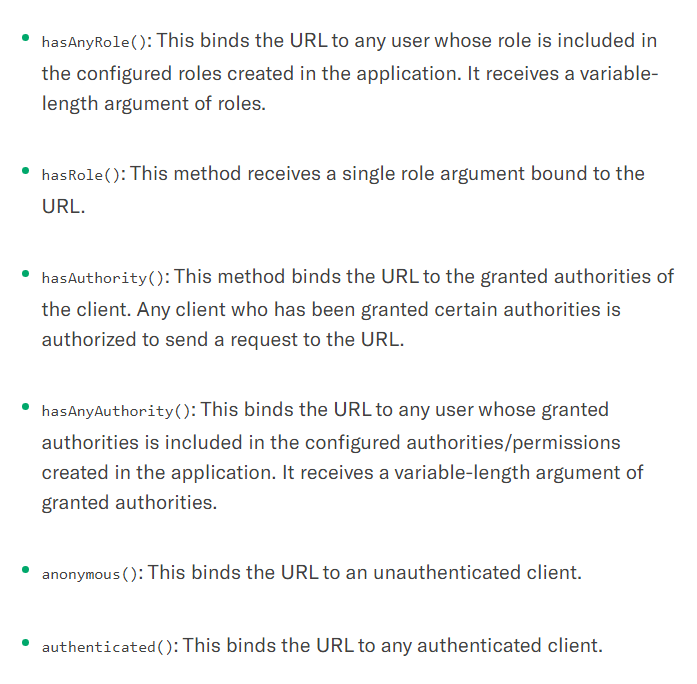
* Takes HTTP method and path



* Takes HTTP method only



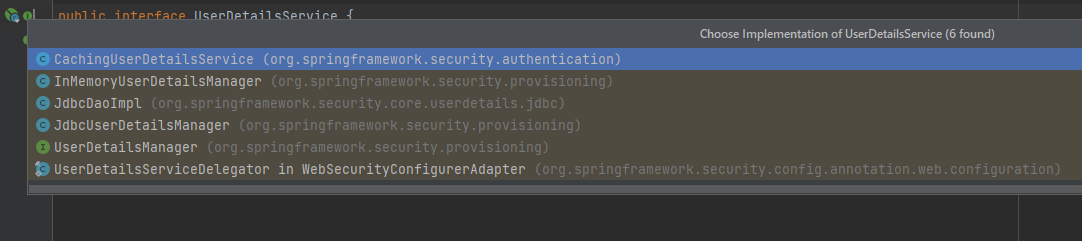
**Important:** Similar to React router, order does matter in ant matcher as well, we need to put more specific path first, and then later the general parts. For example, if we put the last line first on 2nd image, it would allow both ADMIN and ADMIN\_TRAINEE access to all route.



**Password Encoder:** Spring needs a password encoder bean configured, by default is **noop**, but we should use stronger ones, Spring docs suggest things like **brypt** and others and the strength should be tuned so it takes about a second for the system to make the password. It is an interface with 3 methods (*encode*, *matches*, *upgradeEncoding*).

**In-Memory Authentication:**

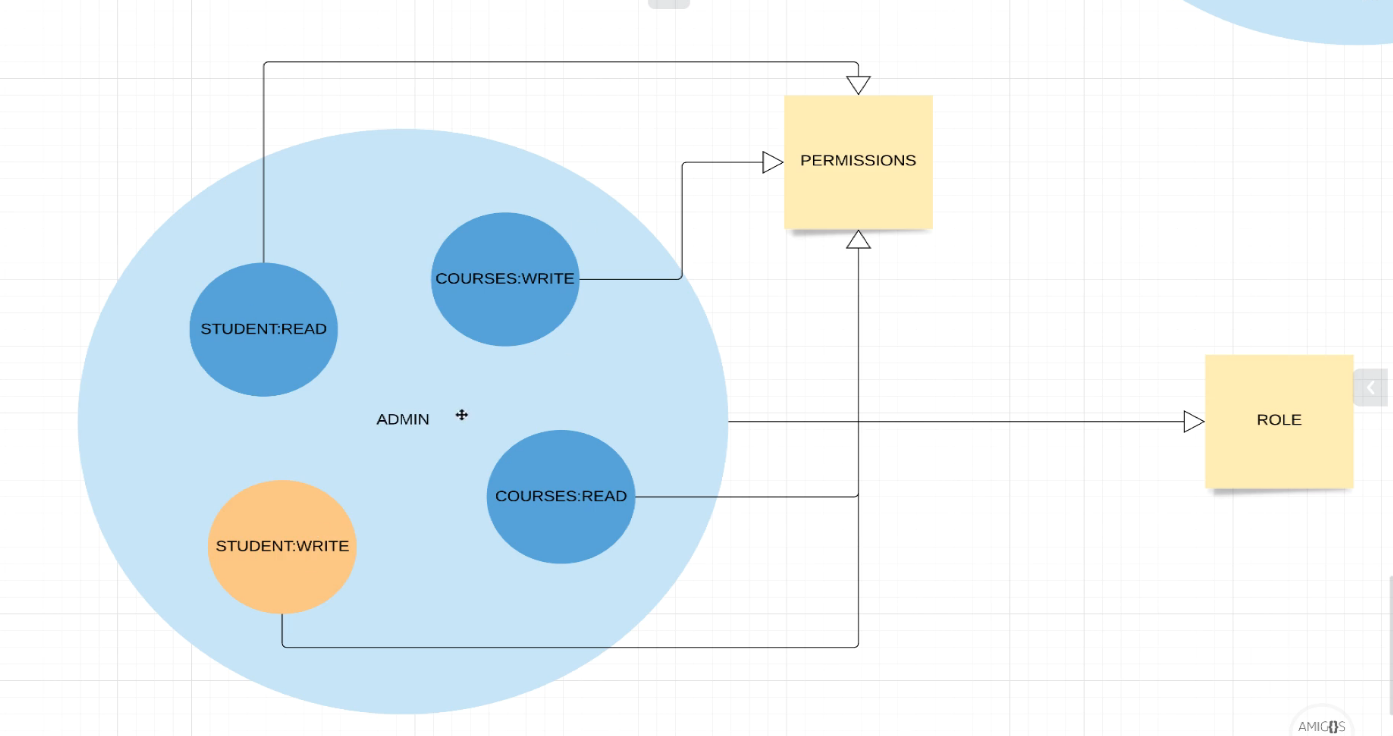
This user details service is how we retrieve our users from database. It is an interface which is implemented by several classes. For the in-memory authentication, it stores the user’s name and passwords hardcoded in memory, and only available till application is running.



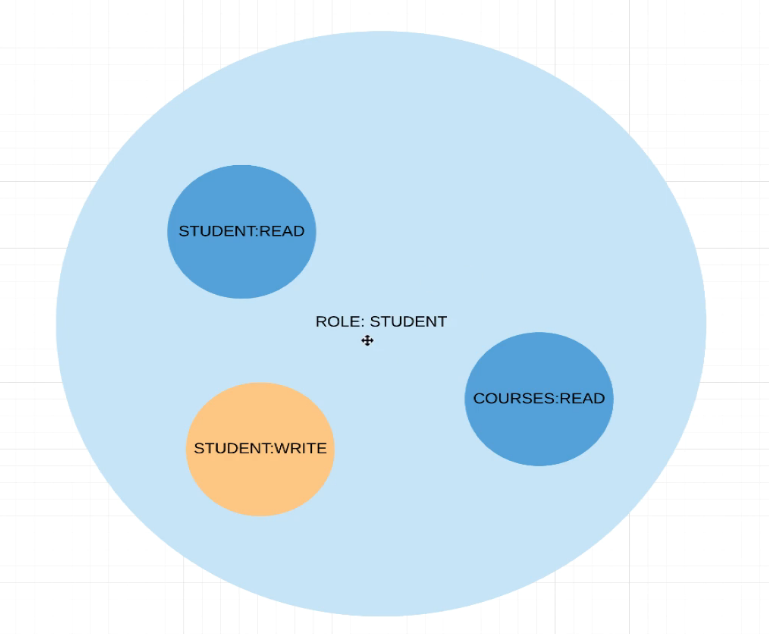
**Roles:** Each Role is a a **coarse-grained** (A few objects hold a lot of related data, thus increase complexity) **GrantedAuthority** that is represented as a String and prefixed with “**ROLE\_** “(default “ROLE” prefix is configurable,).

**Granted Authority:** Each **GrantedAuthority** is a **fine-grained** (more objects each holding less data each, thus less complex) individual privilege (permission). It is optional.

For examples: *READ\_AUTHORITY, WRITE\_PRIVILEGE*, or even *CAN\_EXECUTE\_AS\_ROOT*

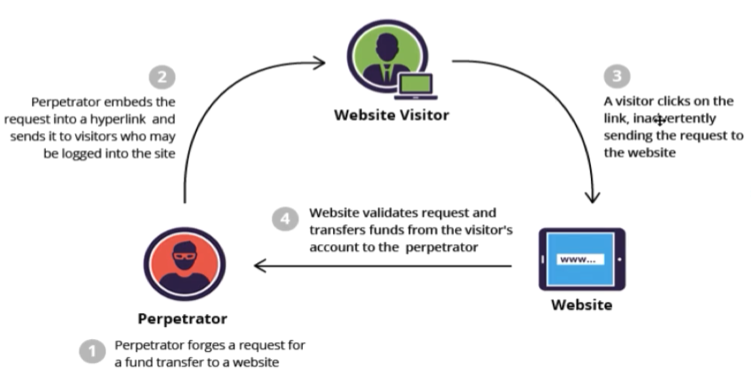


Role is high level view, and within a role we have a set of permission (authority). Here, Admin is a role, and it has 4 authorities. While, Student role has 3. Implementation of this model is in the demo.



* We can use ant matchers (has role and others) to implement Roles, but method annotation (pre/post authorize) is preferred way.
* Both Role and Authority is a string, When building a user, we can assign roles by using roles(String… s) or authorities() which has 3 overload. Authorities’ method accepts SimpleGrantedAuthority object as parameter, so in case we want to create a group of role and permission together, we can do here. Check the demo for example (implementation of getGrantedAuthority() method in the Role class and in config method).

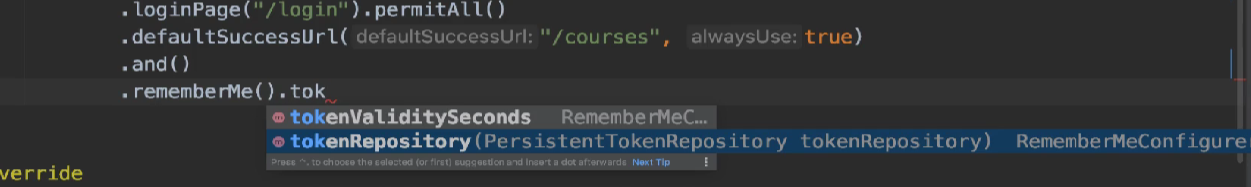
**CSRF:** Cross site request forgery. Consider the scenario here. The visitor is logged onto his bank, but he visits another website. So, this evil website has a link that sends a request to transfer the visitor’s money to evil site owner’s account. Bank will validate the request as the visitor is logged in already and send the money to evil person.



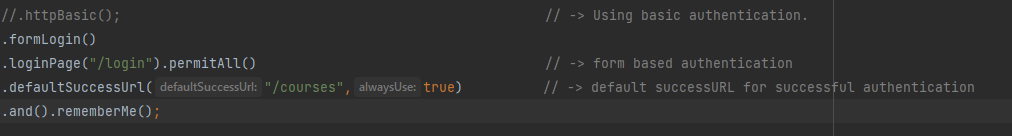
Spring security, by default disable these sorts of request. When the client logs in to the website, spring security sends a CSRF token into a cookie. Then the frontend uses this cookie in any form submission. So, if the evil person tries same technique now it will not work as the server will verify the CSRF token first.

We can configure the process of CSRF in the **HttpSecurity** hook in the config method of our securityconfig class. More details on the AmigosCode course section.

**SessionID:** It is used for session-based authentication, on successful authentication server sends a cookie with sessionID and optional remember me. Spring security by default stores the sessionID in a in memory database. The issue with this is, when we restart server all session ID are gone. So, in real application it is recommended to use Redis or PostgreSQL etc. for storing session ID. We can configure the external database like this,

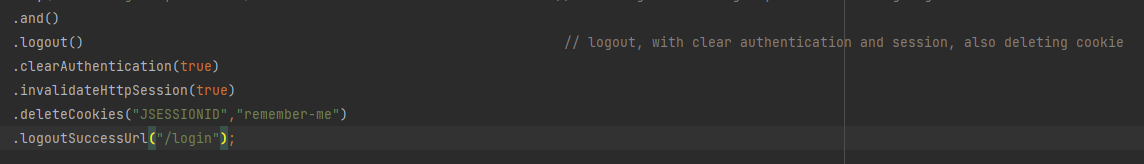


**Rememer Me:** By default, sessionID expire in 30 min. Spring security let us extend this by using remember me (Here session ID default is 2 week).



As we know by default session ID will be stored in memory database, but we should use redis or PostgreSQL. Remember me Cookie has user name, expire date and md5 hash value of these two values. These are managed in the actual database.

**Logout:** In form-based login, once we logout using the default /logout route. Here is the configuration, self-explained.

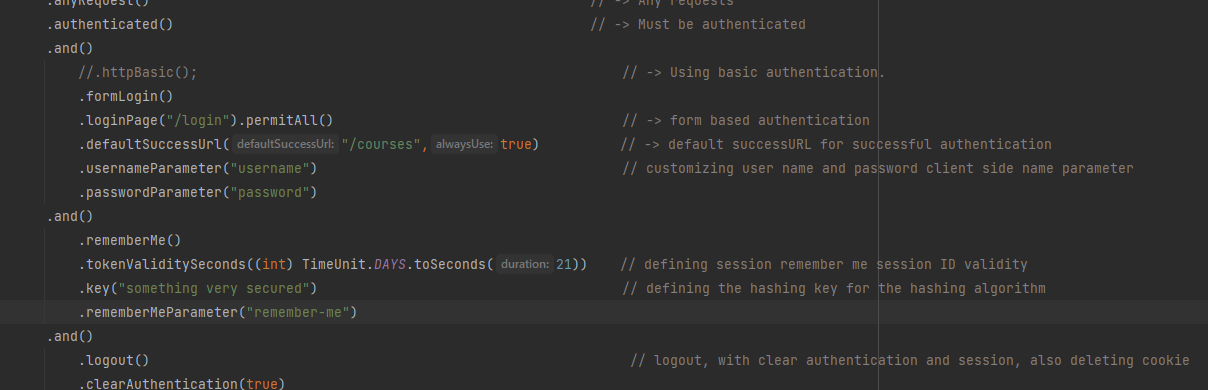


**Configuring input parameters in client side:**

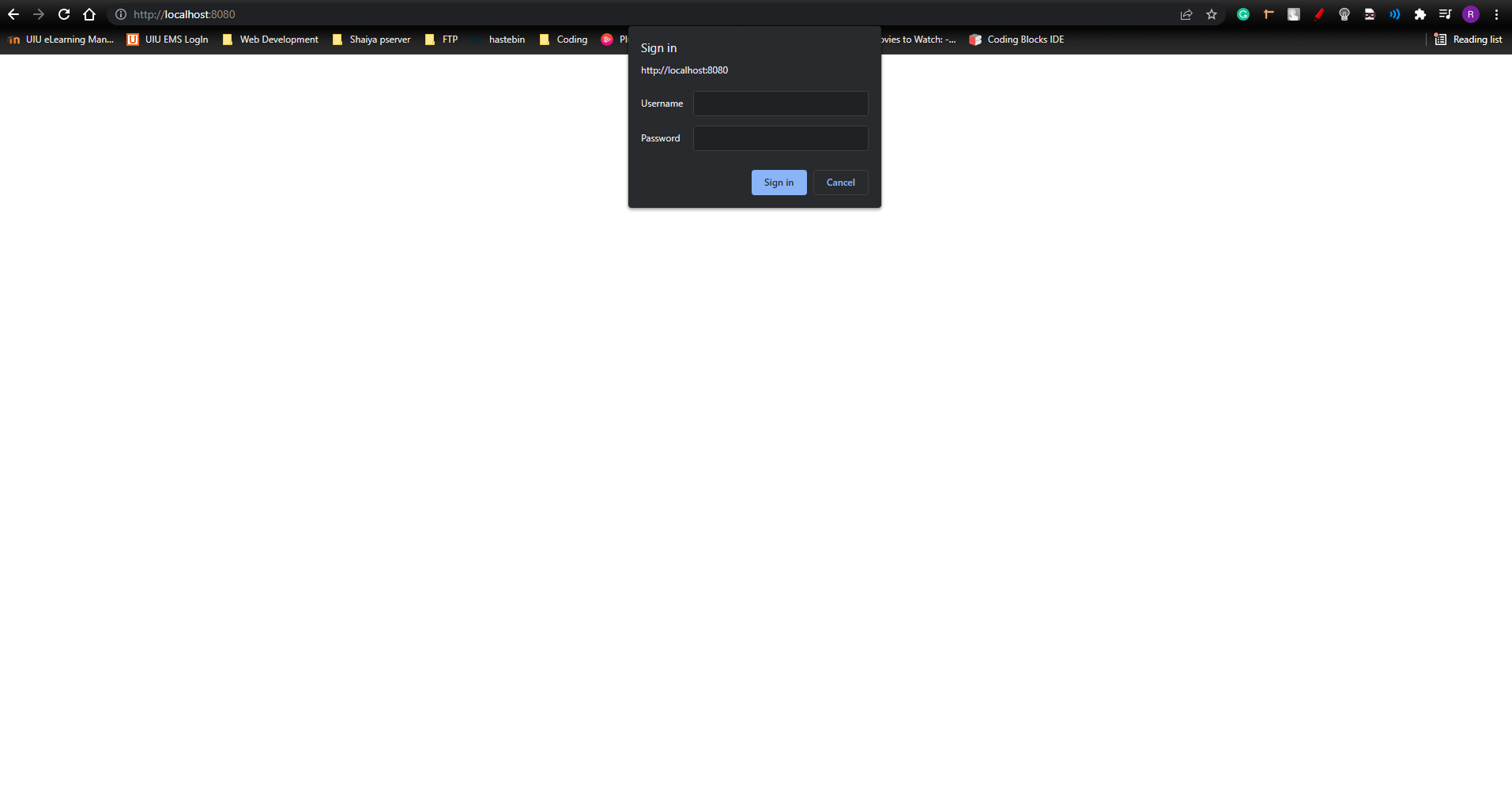
In client side by default

* **Remember-me** checkbox input field **name** attribute should be **remember-me**.
* **username** name attribute should be **username**
* **password** name attribute should be **password**

But we can customize these too



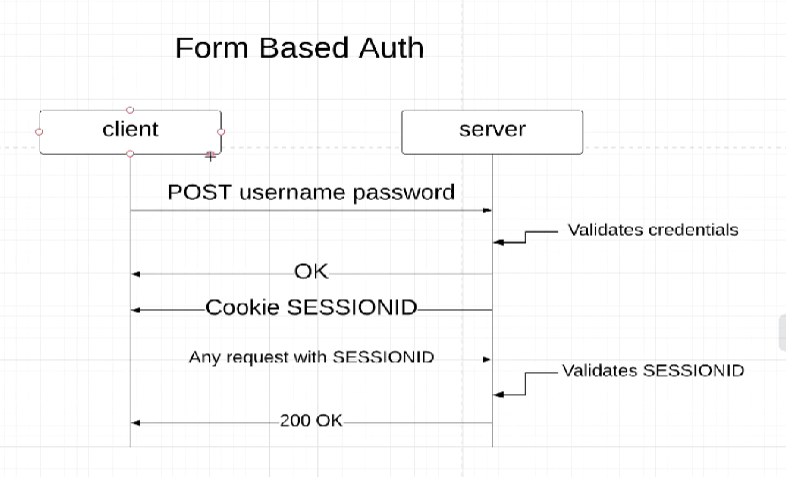
**Basic Auth:** It’s very simple. User sends user name and password for every single request and server authenticate it. It is not very convenient, that’s why it is rarely used. This is usually good only when you are accessing external API. Here is an example, any request we need to enter username and password.

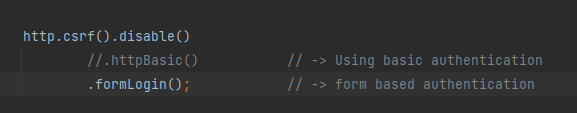


Its downside is we can’t logout once we login, because username/password send in every request.

Setting the http to **Stateless** means every request need authentication

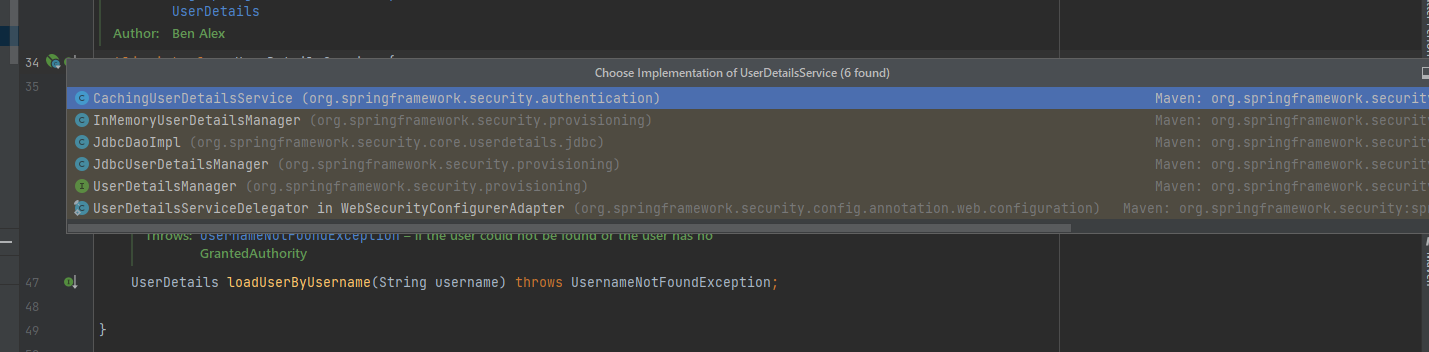
**Form-Based Authentication**: Widely used in most sites. We have full control over how to style our login form, and unlike Basic Authentication we can logout here. HTTPS is recommended for this. Here, after server validates a login request, it sends a cookie with session id. Next time when client send request, it send with session id and server validates it. Usually, session id has a expire time after it expires.



****

**Database authentication:**

As we know the **userDetailsService** method in the **securityConfig** class returns the **UserDetailsService**. It is an interface with only one abstract method **loadUserByUsername.** There are few implementations of this class by default in spring already.



But in case we want to customize our authentication tables and stuff, we can implement this interface and return the implementation here.

* First, we have to create a model for the User table. Spring has the interface **UserDetails** to represent a user. It has some properties that must be there. So, in order to make our own authentication user model, we can implement this interface. Check demo for the example class.
* Next, we need to implement the **UserDetailsService** interface, then we can fetch any user from any database we want by using this class.
* So, like every service, we need a repository that fetches data for us. Next step is to create the UserDao interface, the implementation of this interface will be database specific and we can switch between them if we want to. Check Demo for a fake implementation.
* Now this FakeDaoimpl will be autowired to fetch data for us in our UserService class.

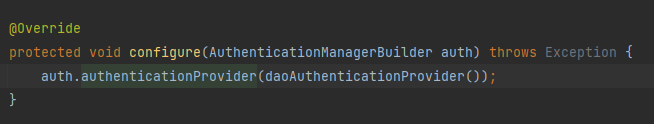
So, in summary we created

1. user model (implementing UserDetails)

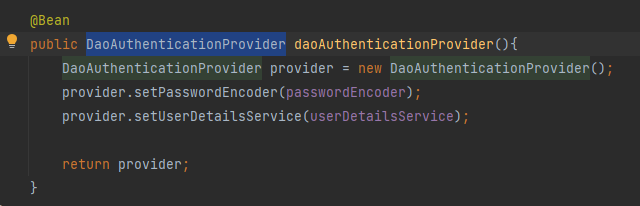
2. UserService (Implementing UserdetailsService)

3. UserDao interface to define data fetch and an implementation for this.

So, now that we have our own user model and user service, we need to configure it to spring security. The place to configure it is here. This method expects an **AuthenticationProvider.**



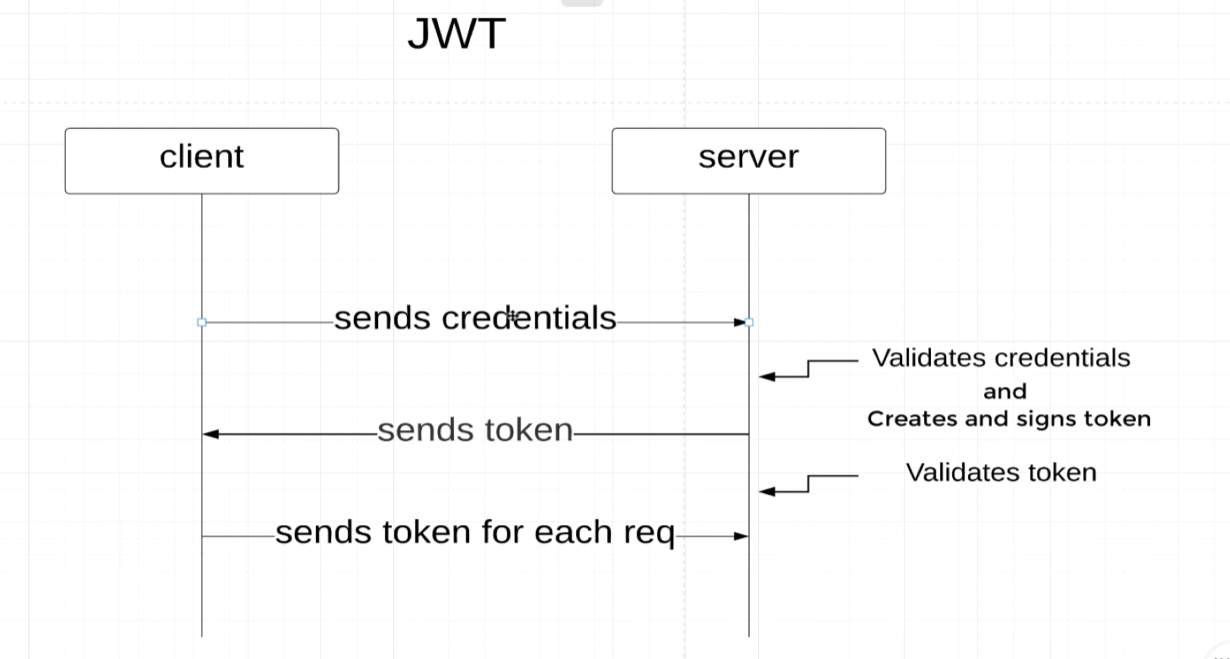
This is an interface, so we need an implementation of it. Spring provides bunch of implementation classes. In this case we created a bean of **DaoAuthenticationProvider (**one of the implementations**)** and passed as the method parameter



This will make the spring security configure the database security. We only have to replace the fakeDaoImplementation to our required database implementation to make it work.

**JWT Authentication:**

JWT is really fast as we don’t need to store session ID, it is stateless and we can use it across multiple platforms. There are some drawbacks in case secret key or JWT token is stolen. Also, we can’t see logged in users.



Client sends credentials and server verify it. If success, server provides a signed token to client. From there, client send this token for every request, server validates this token for validity and expiration.

JWT structure is header + payload + signature. Header and payload everyone can decode, but to decode the signature it needs the secret key. If the token is tempered with then the verification will fail. Term **Claim** is used to represent the key value pair in JWT. If we want to add some info in the token, we can add using the claim method in the library.

Library: study documentation of them to learn more

<https://github.com/jwtk/jjwt> : **Java JWT: JSON Web Token for Java and Android**

Implementation of jwt:

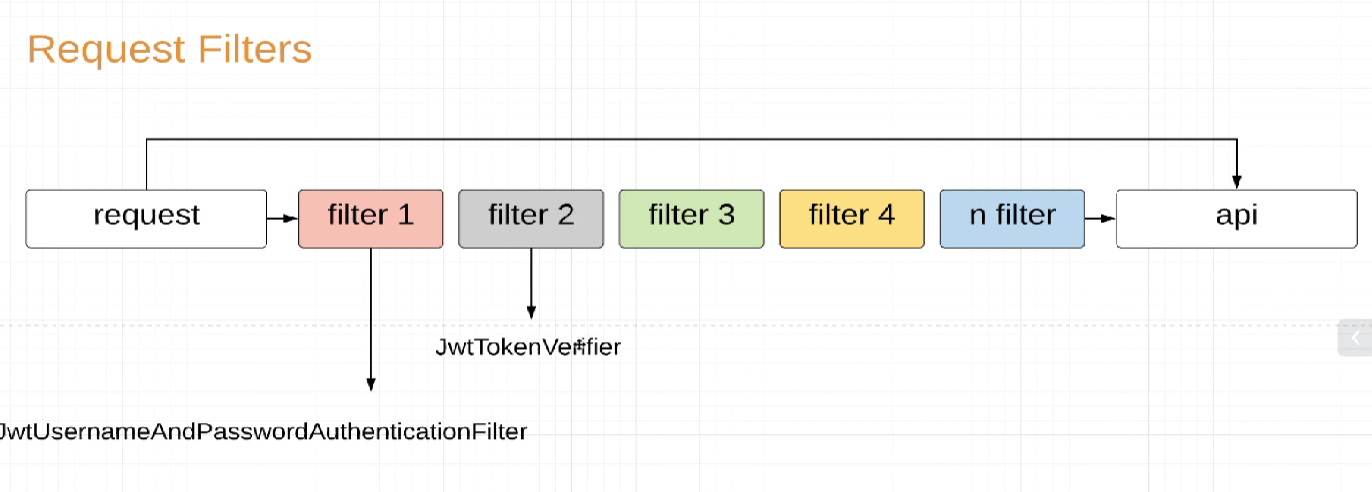
First, we need to extend **UsernamePasswordAuthenticationFilter.** This class process an authentication form submission from client. By extending this class and overwriting its method we can configure how to do the authentication process for JWT.

**attemptAuthentication** **method**: this will attempt to authenticate the request; it will return an authentication object.

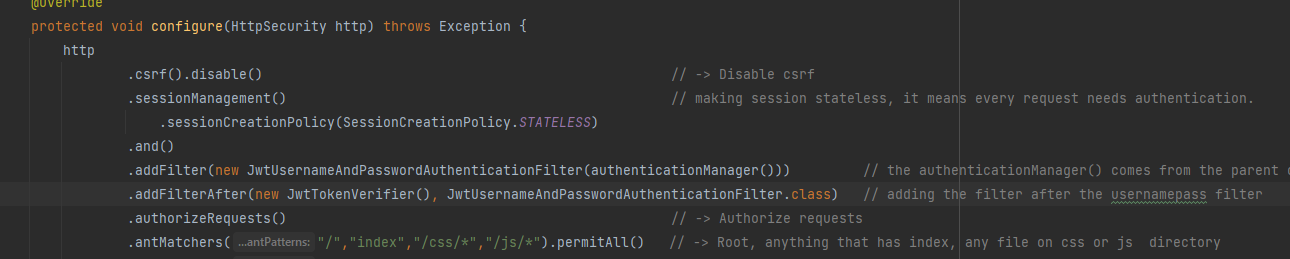
1. First, we deserialize the incoming request and map it to an object.
2. Then, we need to check if this request is valid or not. We use the authenticate() method of Authentication interface. This will check the request and respond if authenticated or not.

**successfulAuthentication method:** If the authentication is successful from the **attempauthentication** method, this method will run. Here we will build the jwt using the library builder and set the header, payload and signature. Then add the jwt token in the response header.

**Request Filters:** Filters are classes that validates the request or reject it. The order is important here.



So, to configure a filter for our jwt, we need to make our session as stateless (the login info won’t be saved, so for each request need the authentication). Then we need to add our custom filters.



First, we add the filter to authenticate the username and password. In the security config, it expects a parameter of type **AuthenticationManager** which we can get via a parent class method. (This part is not clear to me yet, need to look again). Check the demo for that filter class.

Next, we add a filter for the request that comes with a token. It will verify the token for validity. We will extend the OncePerRequestFilter for this, name is enough to understand. Check the demo.