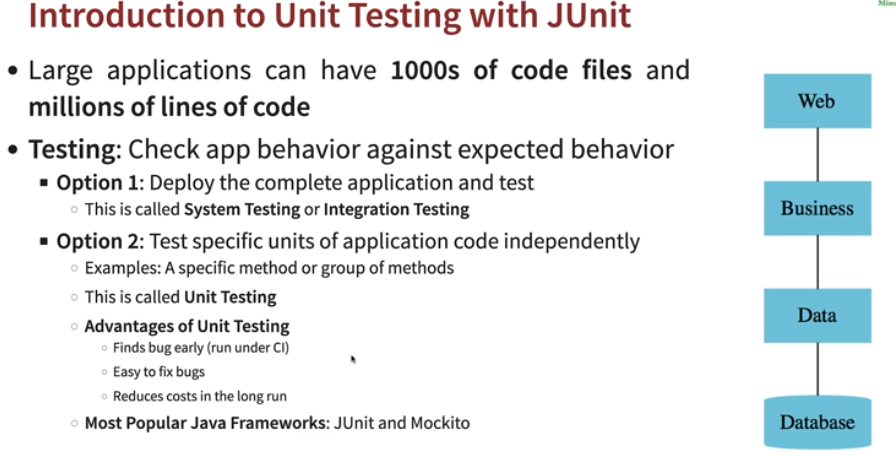
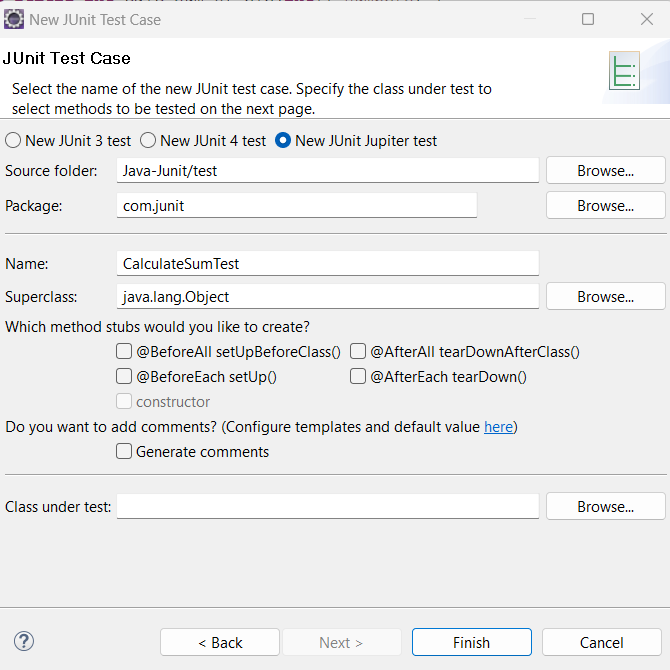
**Unit Test**

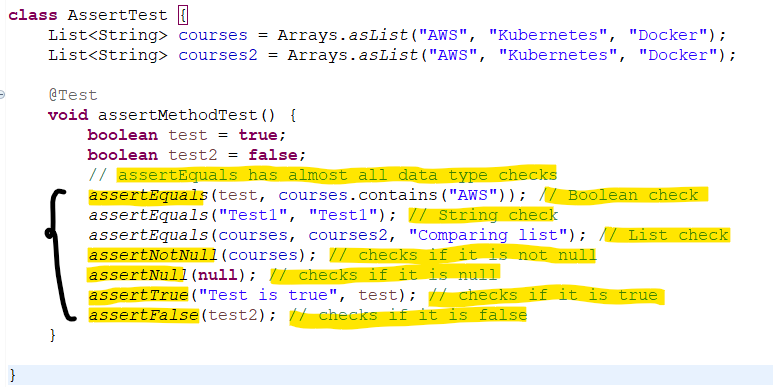
* **JUnit:** Unit testing is testing of small components of applications, it could be method, group of methods or class. Integration/System testing is deploying the application and then testing end to end functionalities. Unit test helps in finding bugs easily and helps in finding bugs at early stage. Two Java frameworks for testing are: JUnit and Mockito.



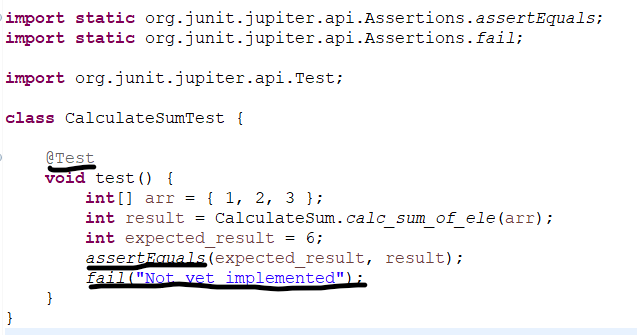
* **Creating Junit test:** 
  + Create a source folder in project with *test(outside src/main)*
  + Create new *Junit cases.*



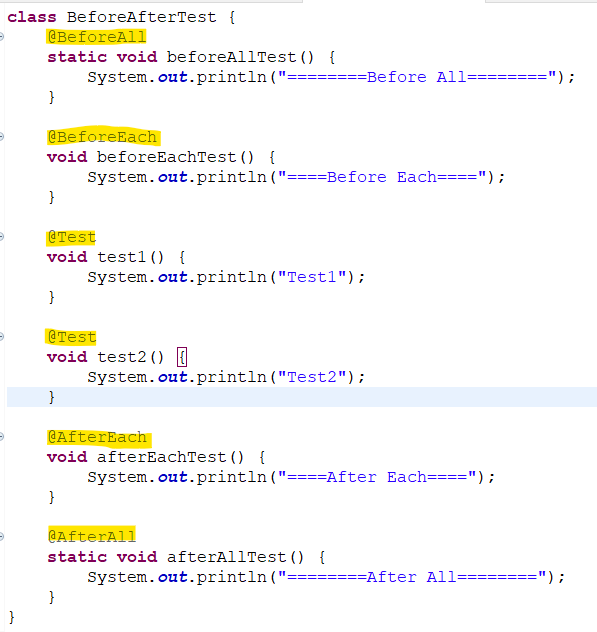
* + Once Junit class written *run as Junit.*
* **@Test:** JUnit *test* is a method contained in a class which is only used for testing. This is called a Test class. To mark a method as a test method, annotate it with the *@Test* annotation.
* **Assert:** Assertions are utility methods to support asserting conditions in tests. Ex: *assertEquals(<1>,<2>);*



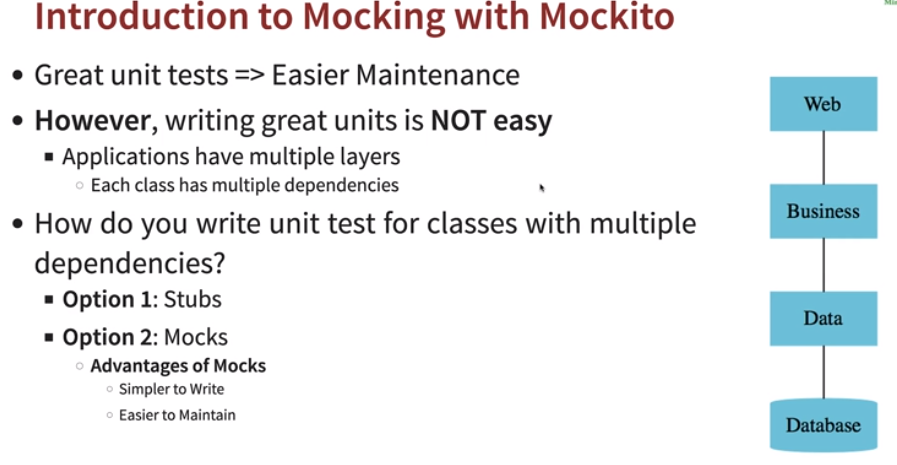
* **Fail:** The fail assertion fails a test throwing an AssertionError. It can be used to verify that an actual exception is thrown or when we want to make a test failing.



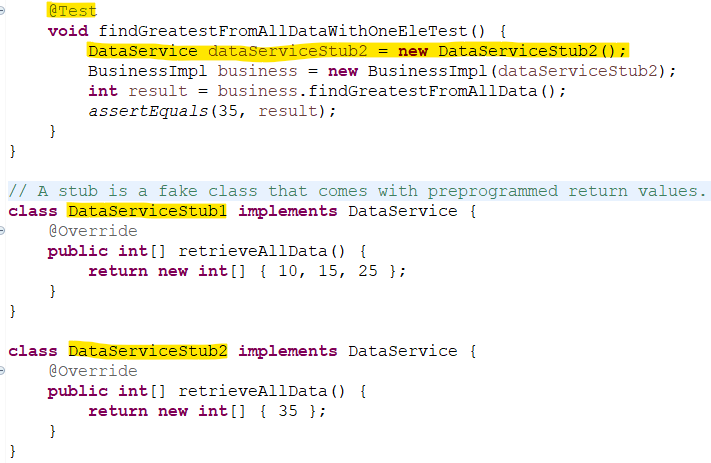
* **@BeforeAll:** *@BeforeAll* is used to signal that the annotated method should be executed before all tests in the current test class. Needs to be *static.* Since it is a class level method and needs to be ran before everything.
* **@AfterAll:** *@AfterAll* is used to signal that the annotated method should be executed after all tests in the current test class. Needs to be *static.*
* **@BeforeEach:** *@BeforeEach* is used to signal that the annotated method should be executed before each *@Test* method in the current test class.
* **@AfterEach:** *@AfterEach* is used to signal that the annotated method should be executed after each *@Test* method in the current test class.



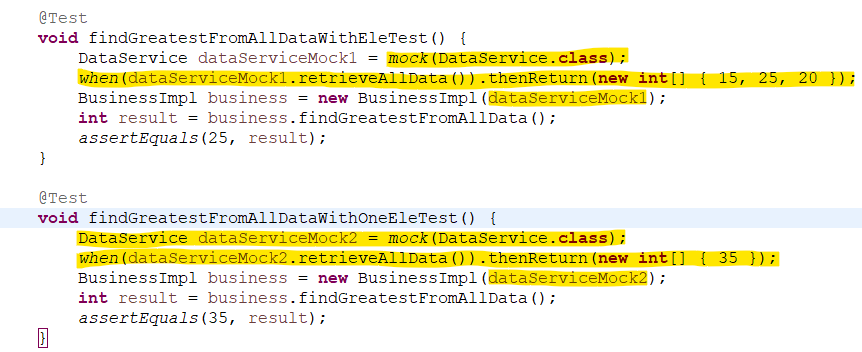
* **Mockito:** Whenever we create a spring boot project, *spring-boot-starter-test* dependency brings all the *mockito* related dependencies used for writing unit tests. *Mockito* is a framework used to write test cases, it makes easier to write test cases for multiple dependencies.



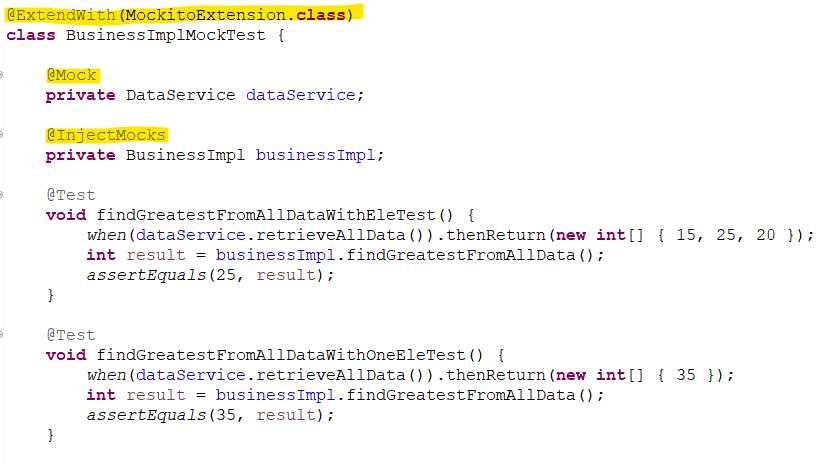
* **Stub:** A stub is a fake class that comes with preprogrammed return values. However, creating stub for each Junit test case in test classes is much more complex and for returning new data we need to create stub again and again, also maintainability is difficult as well.

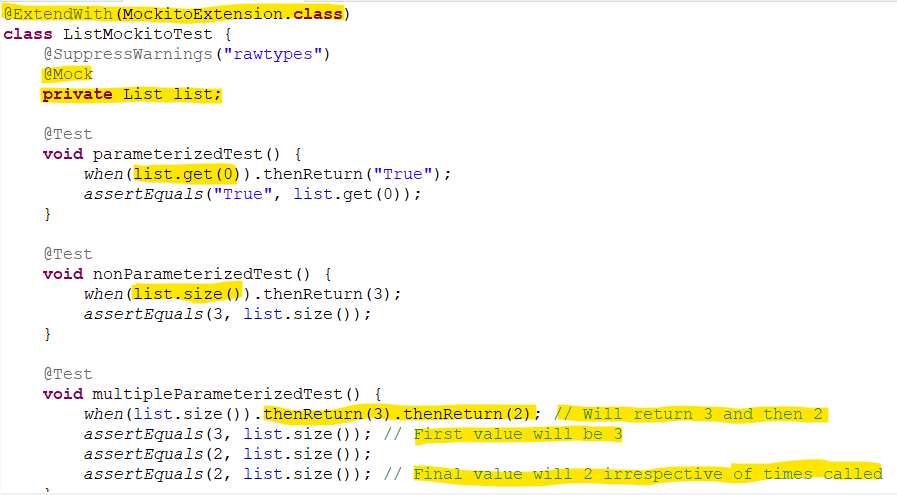


* **Mock:** A *mock* is used to create a fake class and makes it easier to write the test cases. In below code, using mock *when* is used to call the function of that class and *thenReturn* is used to put test data and return result.

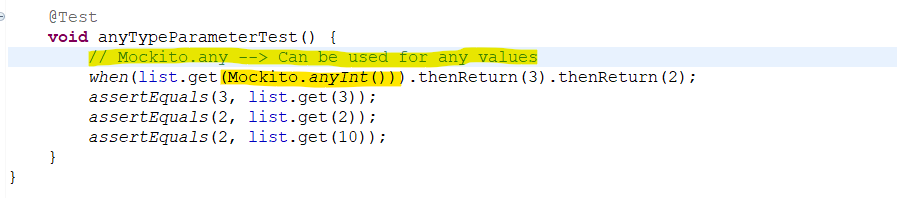


* **ExtendWith(MockitoExtension.class):** *MockitoExtension* is a *JUnit 5* extension provided by the *Mockito* library. It allows you to use the *Mockito* framework to create and inject mocked objects into your JUnit 5 test classes.
* **@Mock:** We can use it to create and inject mocked instances without having to call *Mockito*.
* **@InjectMocks:** It allows you to mark a field on which an injection is to be performed or inject *@Mock class.* It will inject all *@Mock annotated* dependencies.



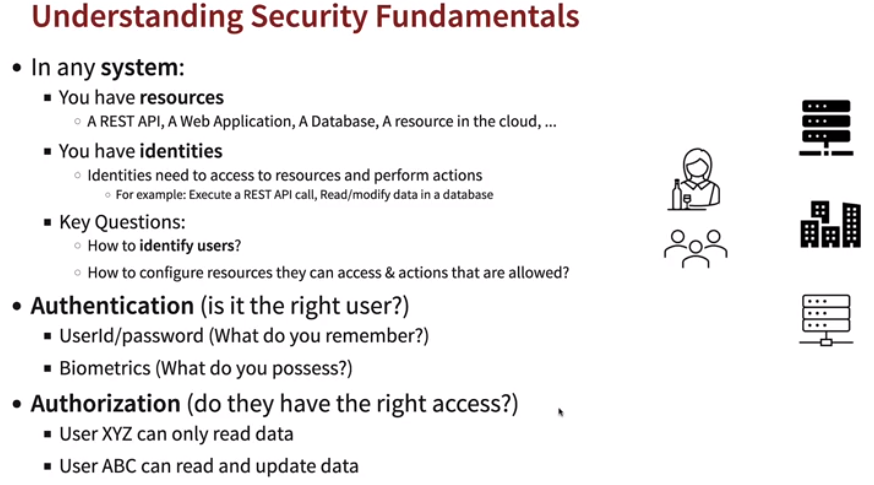


* **Mockito.any<Type>():** It is used to accept any parameter while testing test cases.

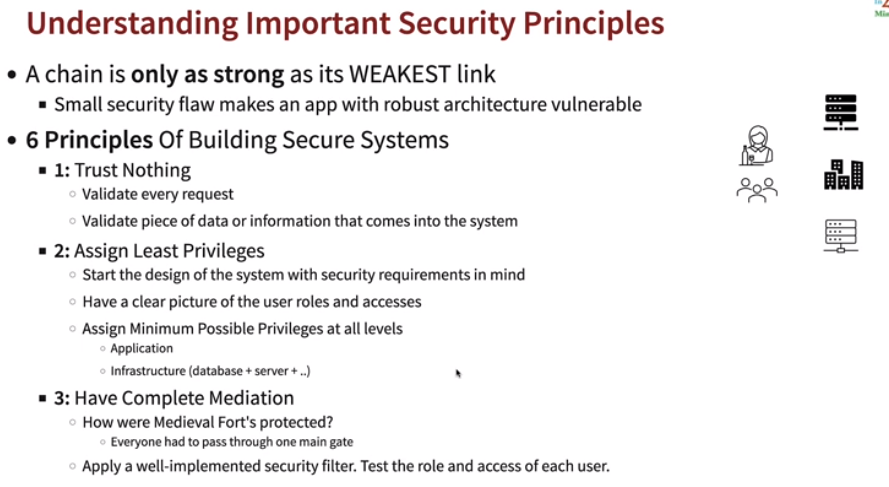


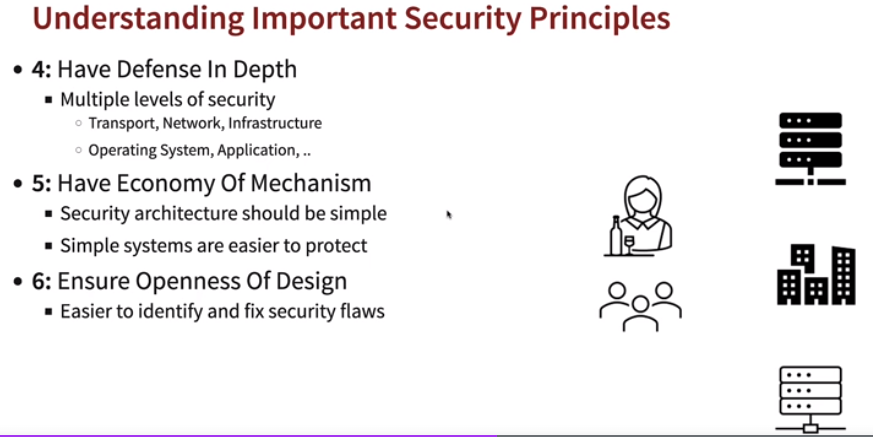
**Spring Security**

In any system we have multiple systems, like RestAPI, webapplications, DB etc. In order to identify the correct request coming in and securing application we can configure *Authentication (Correct request)* and *Authorization (Type of access).*



* **6 principles of implementing security:** 
  + **Trust Nothing**
    - Validate everything and each piece of data or information in system.
  + **Assign Least Privileges**
    - Consider what kind of access each person can have and assign minimum access as per requirement.
  + **Complete Mediation**
    - Apply such security layer, so everything comes and passes through the security filter layers only if they are valid.
  + **Have Defense In Depth**
    - Implement security at each and every layer like data, infrastructure, transport etc.
  + **Economy Of Mechanism**
    - Should be simple security architecture.
  + **Openness Of Design**
    - Should be easier to find flaws in security if in any case of issues.

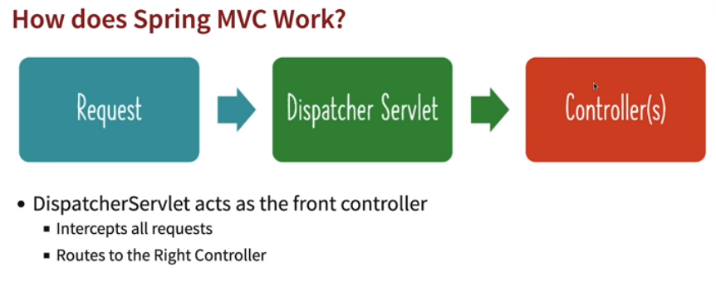




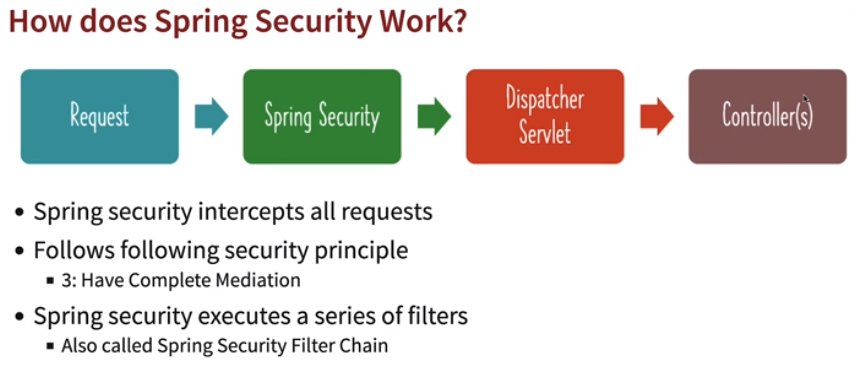
* **Spring Security:** *Spring security* provides protection for RestAPI, webapplications and microservices and is widely used.

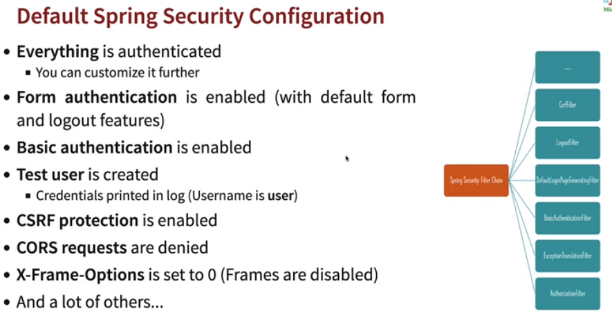
****

In *Spring Framework* every request comes to *DispatcherServlet* which intercepts every requests, then passes the request to right *controller.*



*Spring Security* adds a layer before the *DispatcherServlet* once implemented and it intercepts all requests before *DispatcherServlet. Spring Security* executes multiple series of filters also known as *Spring Security Filter Chain.*

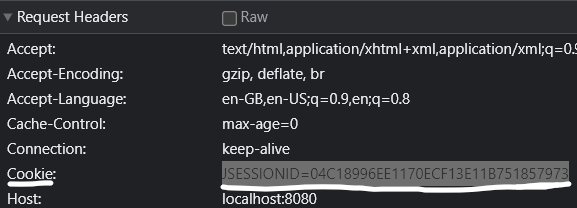


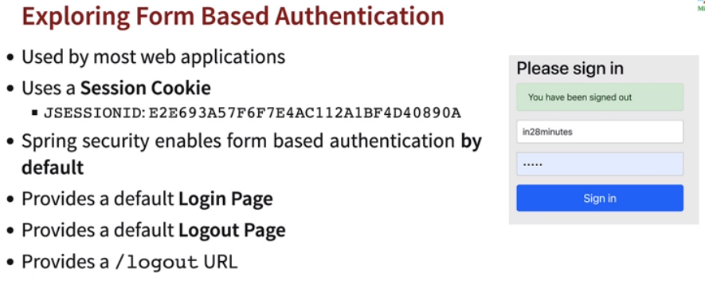


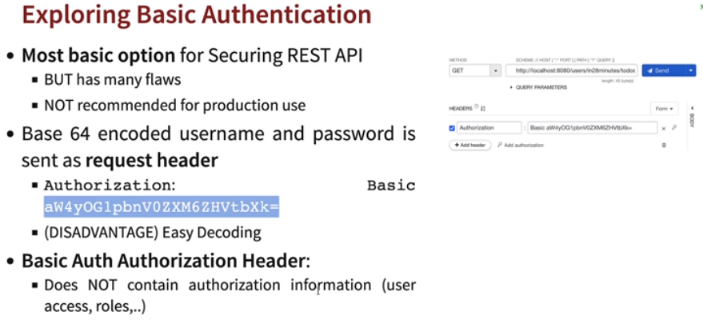
* **Filters provides these features:**
  + *Authentication:* Validates user. *(BasicAuthenticationFilter)*
  + *Authorization:* Verifies the user has access. *(AuthorizationFilter)*
  + *CORS (Cross Origin Request Forgery) (CorsFilter)*
    - Checks for requests coming from frontend, backend or database.
  + *CSRF (Cross Site Request Forgery) (CsrfFilter)*
    - Cookies are stored for each request on browser, if user switches to other malicious website the cookies stored information can be used by malicious site.
  + Login Page, Logout Page
  + Proper response handling for exceptions and return HTTP responses to user.



* **Basic Authentication:** Spring provides basic *form* based basic authentication by default.
  + It will provide with a login page and (*/logout)* logout page.
  + It uses *Base64* encoding by default. Ex: *Basic VHVsc2k6dHVsc2k=*
  + Once *spring security* is added to a RestAPI all valid or invalid urls for that server/host needs to be first logged in.
  + However, it is easily decodable.
  + It doesn’t contain user access role details and once created will not expire.
  + Uses a *Session Cookie.* It is generated when logged in for all other requests. Ex: *JSESSIONID=04C18996EE1170ECF13E11B751857973*



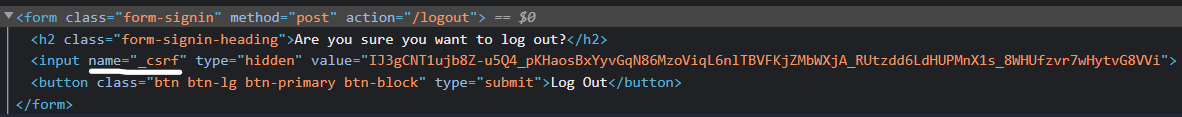


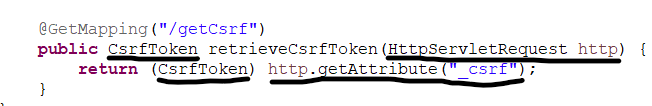


* **CSRF:** *Cross Site Request Forgery.* When we login to any (ex: bank) site a *cookie* is generated and if we change to any other malicious site, then other website can access the *cookie* and can use that *cookie* to send a *post* or *put* request to gather important details. By default *spring security* blocks the *post* and *put* requests and will ask for a token to be generated for each requests. However, it will allow *get* requests.



* **Getting CSRF token:** 
  + *CsrfToken* implementation provides CSRF token related information.
  + The *HttpServletRequest* provides methods for accessing parameters of a request. The type of the request determines where the parameters come from.
  + *http.getAttribute("\_csrf")* returns the value of the named attribute as an Object, or null if no attribute of the given name exists.

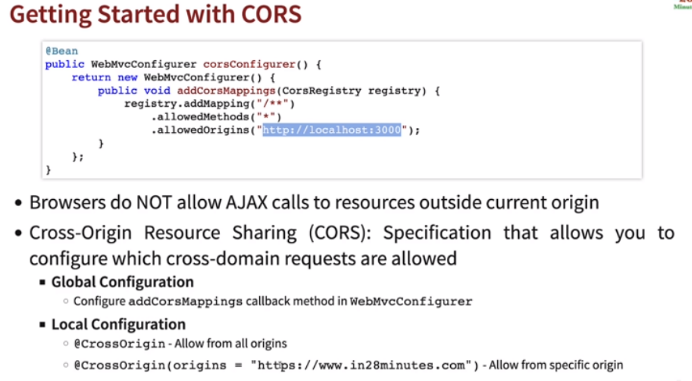


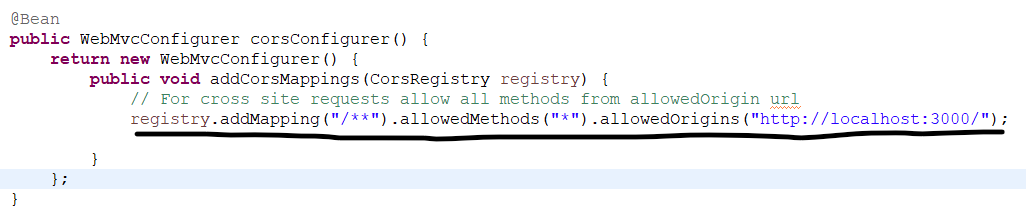


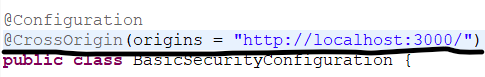
* **Creating Basic Auth:** 
  + Overriding the *SpringBootWebSecurityConfiguration(SecurityFilterChainConfiguration())* as it enable Spring security login.
  + Created API as stateless.
  + Disabled basic login form.
  + Disabled csrf.



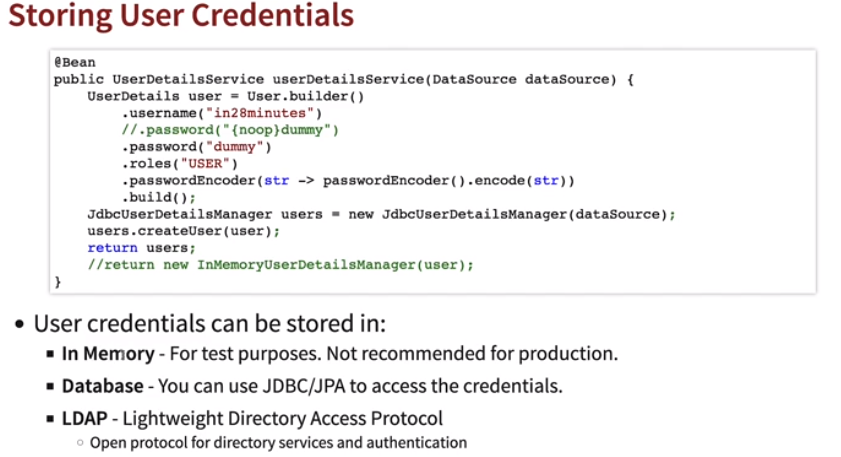
* **CORS:** Any calls outside domain is blocked by CORS.



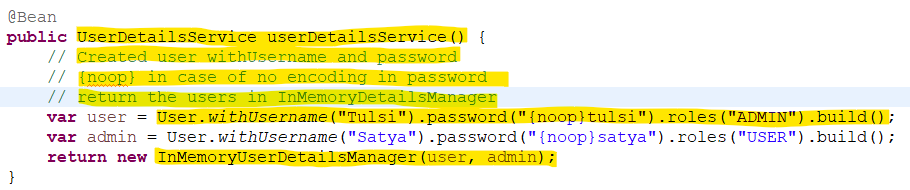




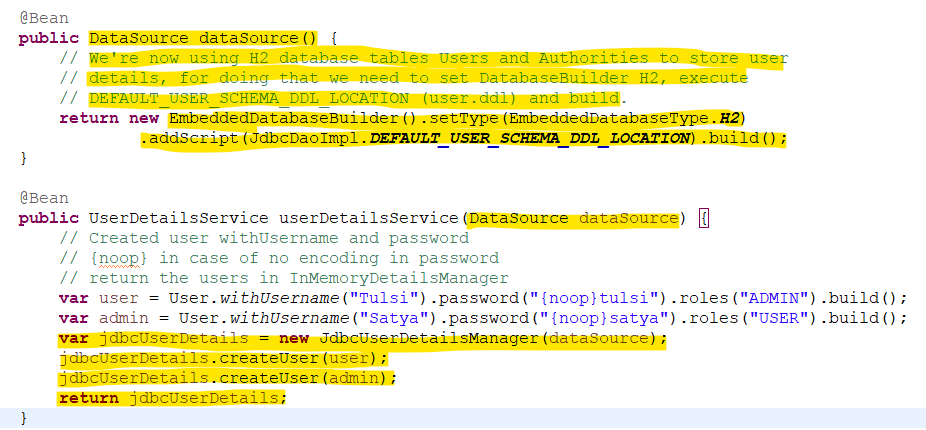
* **UserDetailsManager:** We can configure user, password and role using:

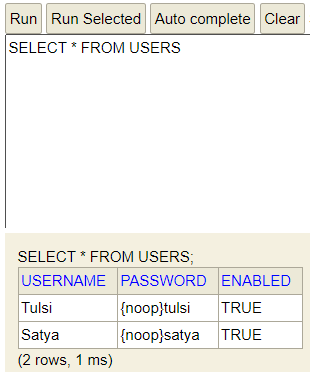
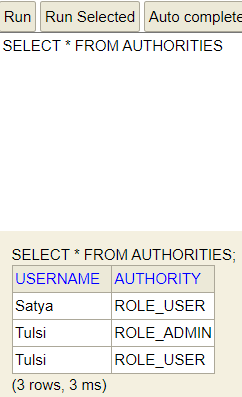
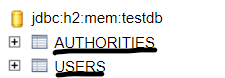


* + *InMemory:* For test purposes.

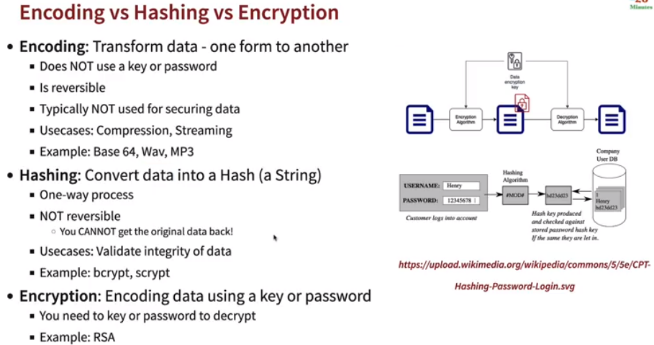


* + *Database:* Using JDBC/JPA.
    - In case we need not write our own tables in H2 db, we can use default *user.ddl* provided by the *JdbcDaoImpl.* Create new datasource to run the *DEFAULT\_USER\_SCHEMA\_DDL\_LOCATION (user.ddl)* to run as script at start.
    - In *UserDetailsManager,* inject *DataSource created* and using Object of *JdbcDetailsManager* create users and return.

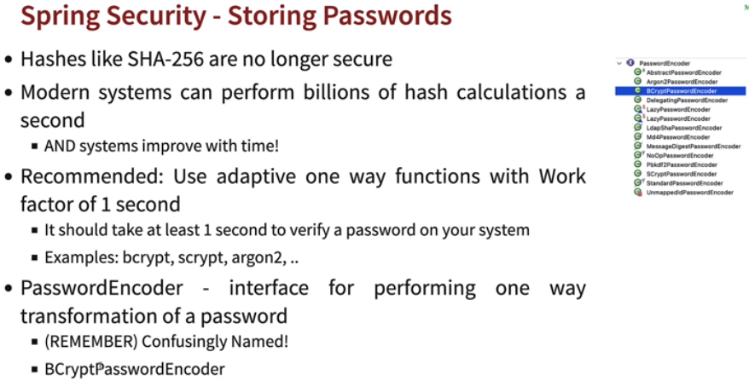




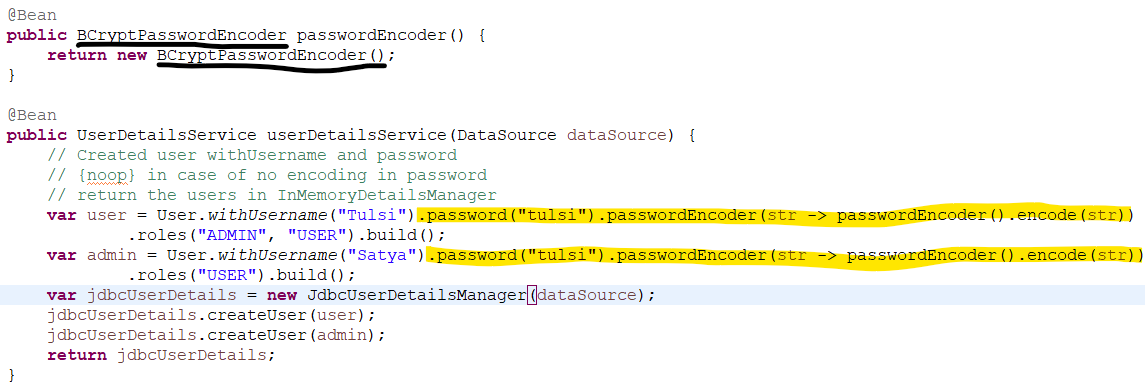
* + *LDAP:* LightWeight Directory Access Protocol.
* **Encoding vs Hashing vs Encryption:** 
  + **Encoding:** Transforms one data to other format like Base64 etc. It is reversible and not recommended.
  + **Hashing:** Convert data to Hash. It is one way process and any data transformed to hash cannot be converted back to original data. Used to data integrity. Let’s say in case we need to store password in databases and we can confirm if password is correct or not using hash function.
  + **Encryption:** Encoding data using a key or password and can only be decrypted using the key.

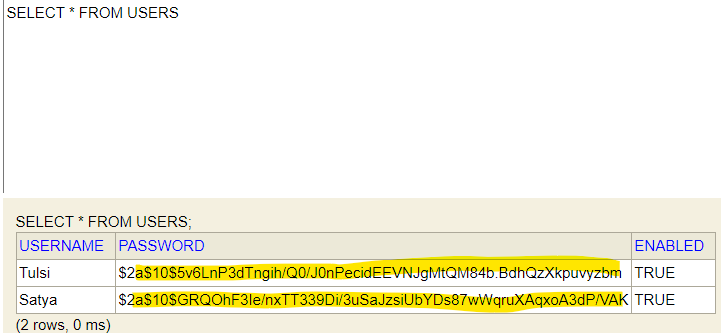


* **Encoding the password:** Modern systems can run multiple hash calculations in a second and hence Hashes are not so secure.

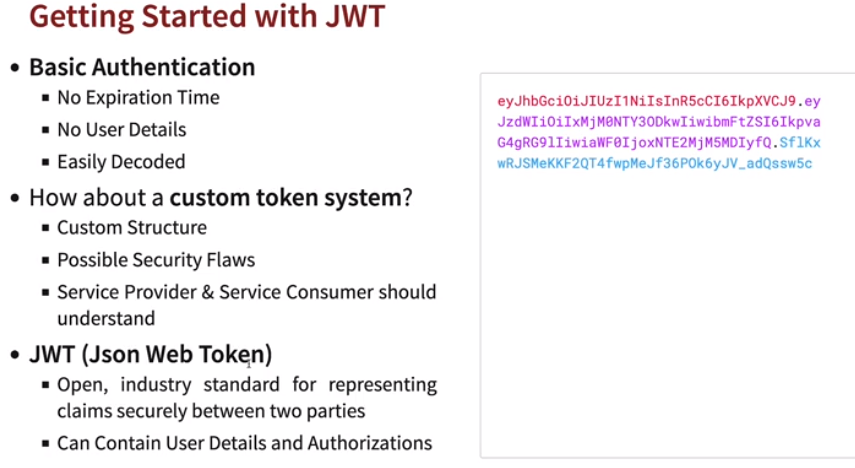


* + Created a *BCryptPasswordEncoder* object*.*





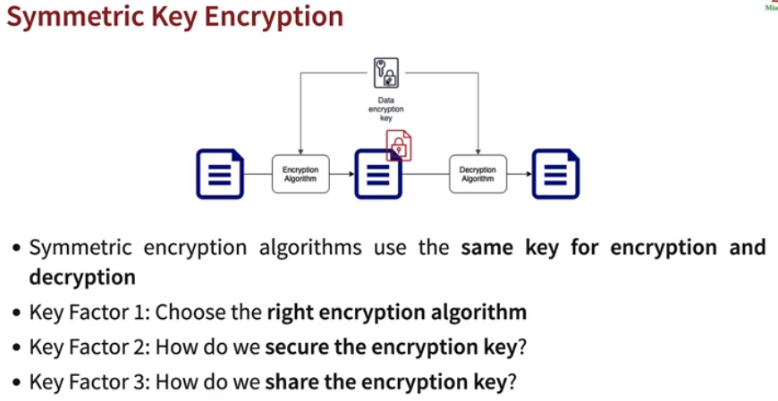
* **JWT:** *Json Web Token*
  + Basic authentication has no expiration time and can be easily decode.
  + Custom Token system has security flaws.
  + Open standard for securing exchange between two parties.



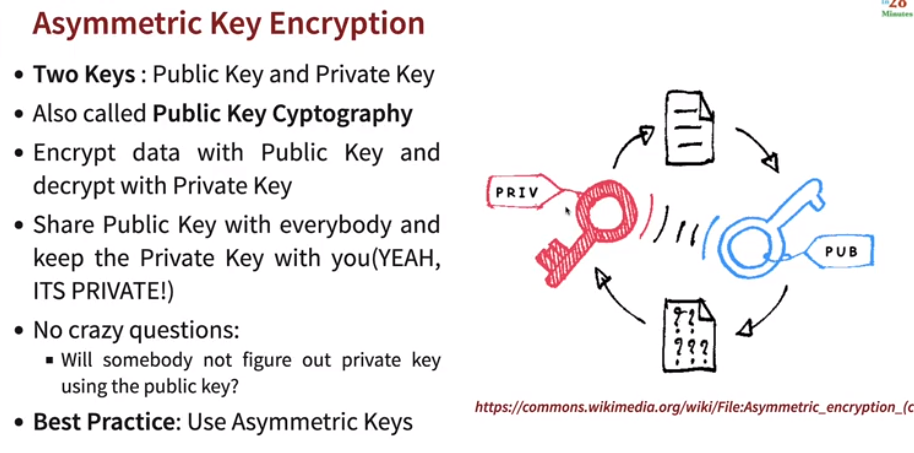
* + *JWT* contains.
    - *Header* using hash algorithm
    - *Payload* contains information like the *(iss)* issue, *(sub)* the subject, *(aud)* audience, *(exp)* expiration and *(iat)* when token issued.
    - *Signature* which includes a secret.



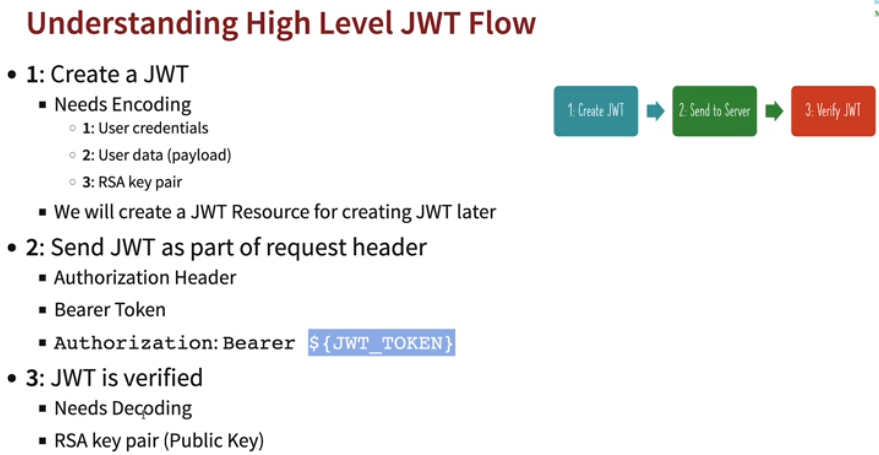
* **Symmetric vs Asymmetric Key Encryption:**
  + *Symmetric:* Same key is used for encryption and decryption.



* + *Asymmetric:* Two keys are there *Public Key* and *Private Key.* Also known as *Public Key Cryptography.* Encrypt data with *Public Key* and Decrypt with *Private Key.*



* **JWT High Level flow:**
  + *Create a JWT*
    - Needs Encoding of User Credentials, user Data *(payload),* RSA Key pair.
  + *Send JWT as Request Header*
    - Authorization header
    - Bearer Token
    - Authorization: *Bearer ${JWT\_TOKEN}*
  + *JWT Decoding*
    - Decoding
    - RSA Key Pair *(private key)*



* **Creating a JWT Authentication:**
  + Add below dependency in pom.xml.

*<dependency>*

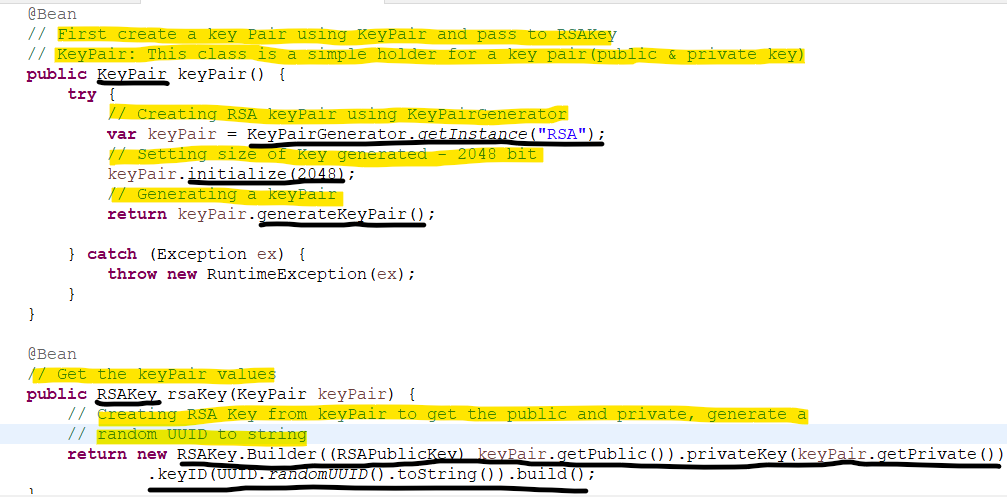
*<groupId>org.springframework.boot</groupId>*

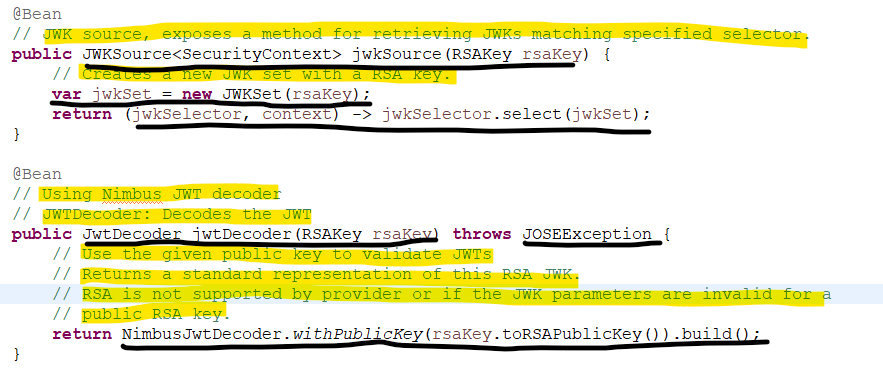
*<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>*

*</dependency>*



* + Create a *KeyPair (Class holding Public and Private Key)* and using an instance of *RSA (.getInstance(“RSA”)),* set size *(.intialize(2048))* and generate key pair *(.generateKeyPair()).*
  + Next is create a *RSAKey* using the *KeyPair* generated.
    - Get the *public key* from *keyPair (.getPublicKey()).*
    - Get the *private key* from *keyPair (.getPrivate()).*
    - Generate a UUID (Unique ID) to string. *(.keyID(UUID.randomUUID().toString()))*
  + Using *JWKSource (Json Format key)* to create new *keyset* from RSA key.
  + Create *JwtDecoder* to decode *RSAKey* and return exceptions if any issues.
  + Create *JwtEncoder* to encode *JWKSource*.



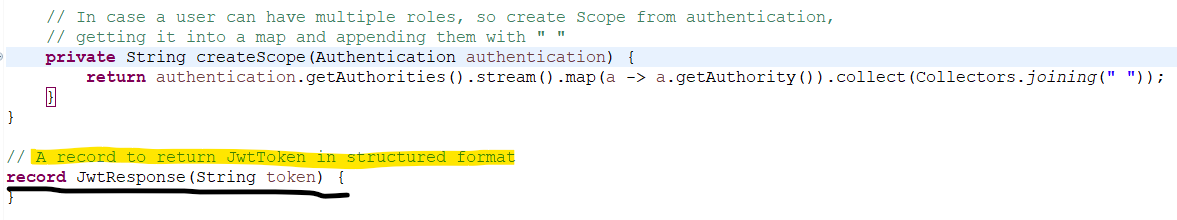


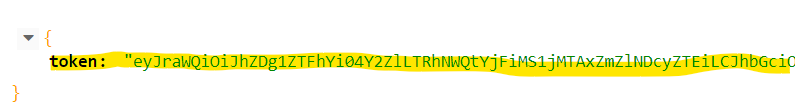
* **Returning JWT Token in response:**
  + *return authenticate;* --> Will return Authentication object details like user details, role etc
  + *Authentication* object will return below values.



* + *Authenticate* object is passed to *createToken* method for adding details like *issuer, issuedAt, expiresAt, subject* and *scope* *(Access type)* etc
  + Using *JwtClaimsSet* we're setting *issuer, issuedAt, expiresAt, subject* and *scope* and then encoding the values and getting token value.
  + In case a user can have multiple roles, so create *scope* from authentication, getting it into a map and appending them with *" "*.

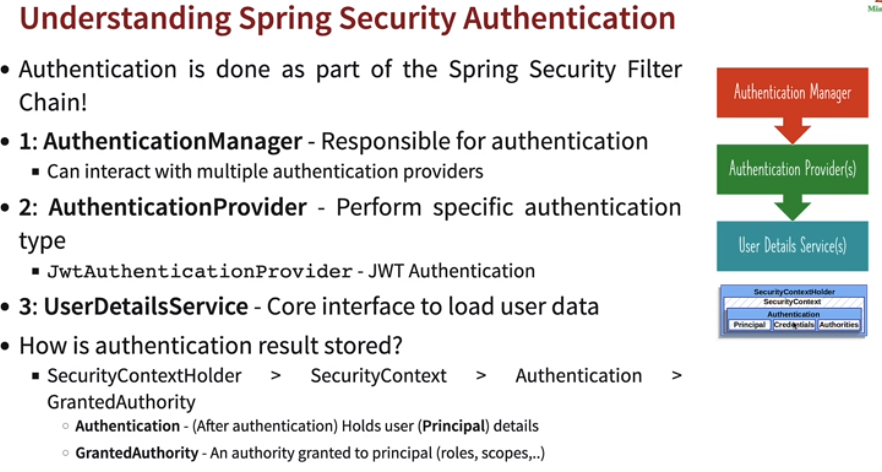






* **Spring Security Authentication on high level:** 
  + Authentication is done as part of *Spring Security Filter chain.*
  + *AuthenticationManager* is responsible for authentication.
    - Before authentication *SecurityContext* has *credentials* only.
    - Once authenticated *SecurityContext* will have *principal (user details)* and *Authorities (roles)* as well.
  + *AuthenticationManager* perform authentication type like JWT authentication *(JwtAuthenticationProvider).*
  + *UserDetailsService:* Interface to load user data.
  + Authentication result is stored in *(SecurityContext)* *SecurityContextHolder 🡪 SecurityContext 🡪 Authentication (holds user prinicipal details) 🡪 Granted Authority.*

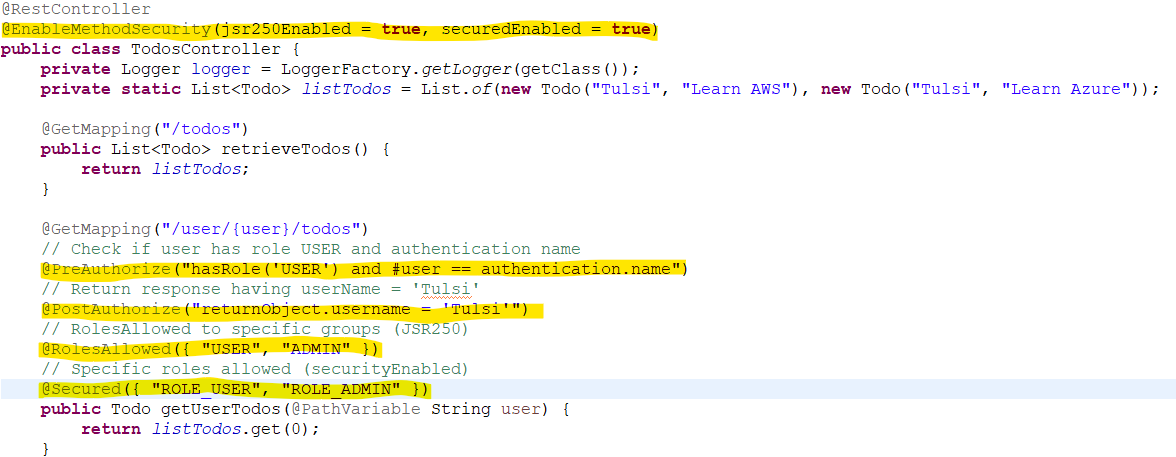




* **Spring security Authorization:** Used to configure Spring security authorizations. Used to match certain requests and decide authorization for methods.



* + *Global Security: authorizaHttpRequests*
    - *requestMatchers(“/users”).hasRole(“USER”)*
      * *hasRole, hasAuthority, hasAnyAuthority, isAuthenticated.*
  + *Method Security(@EnableMethodSecurity)*
    - *@Pre and Post Annotations*
      * *@PreAuthorize()*
      * *@PostAuthorize()*
    - *JSR-250 Annotations*
      * *@EnabledMethodSecurity(jsr250Enabled=true)*
      * *@RoleAllowed*
    - *@Secured Annotation*
      * *@EnabledMethodSecurity(securedEnabled=true)*
      * *@Secured({“ADMIN”,”USER”})*



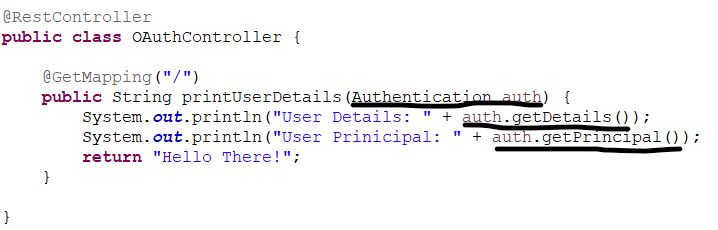
* **Secure connection with Google Drive:** When we want to connect to Google Drive, twitter, facebook etc and retrieve information using OAuth client ID and secret.



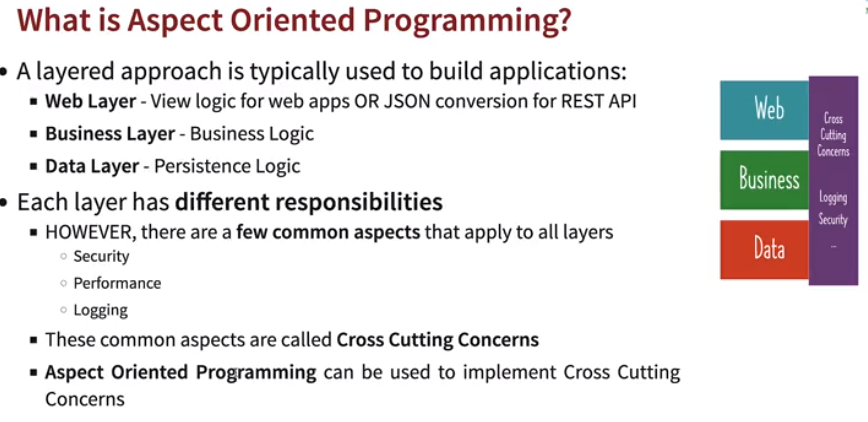
* **Generating Oauth Client ID and Secret from Google API console:**
  + Go to Google API Console.
  + APIs and Services 🡪 Credentials 🡪 Create New Credentials 🡪 Select OAuth Client ID
  + Configure Consent Screen 🡪 External 🡪 Enter Details 🡪 Save and Continue 🡪 Add scope (Details to be shared) 🡪 Back To Dashboard
  + Create Credentials 🡪 Select type of webapplications 🡪 Enter Domain of your application *(Authorised redirect URIs (http://localhost:8080/login/oauth2/code/google))* 🡪 Create 🡪 Get Client ID and secret.
  + In *application.properties* add below properties.

*spring.security.oauth2.client.registration.google.client-id=YOUR\_CLIENT\_ID*

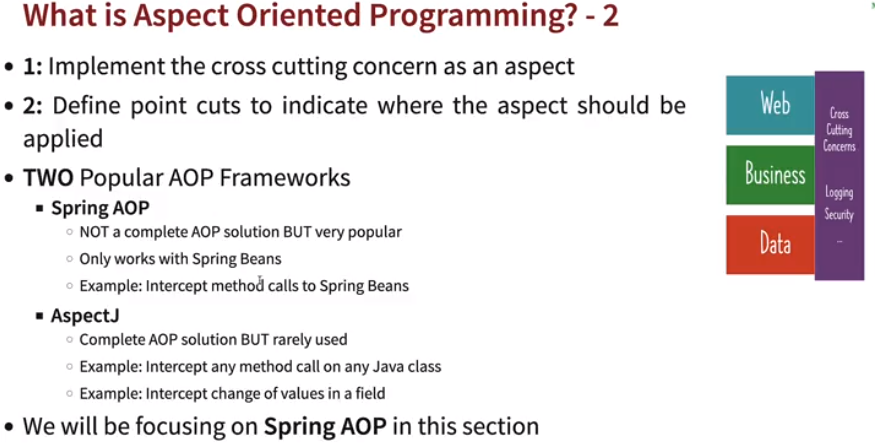
*spring.security.oauth2.client.registration.google.client-secret=YOUR\_SECRET*



* **Spring AOP:** Web layer is where weapps or JSON conversion of API happens, Business layer consists of Business logic to be performed and Data layer is persistence layer containing the interaction with Database code (SQL, Postgre, MongoDB etc). However, they have common aspects like logging, security and performance. To implement these common aspects we use *Aspect Oriented Programming.* The common aspects are known as *Cross cutting concerns. AOP* reduces the lines of code.



* **AOP frameworks:** Two popular *AOP* frameworks are:
  + *Spring AOP*
    - Mostly used by provides incomplete *AOP* solution.
    - Only works with Spring beans.
    - Intercept method calls to Spring Beans.
  + *AspectJ*
    - Rarely used but provides complete *AOP* solution.
    - Intercept any method in Java class and change of values in a field.



* **First AOP project:**
  + Add the below dependency in pom.xml.

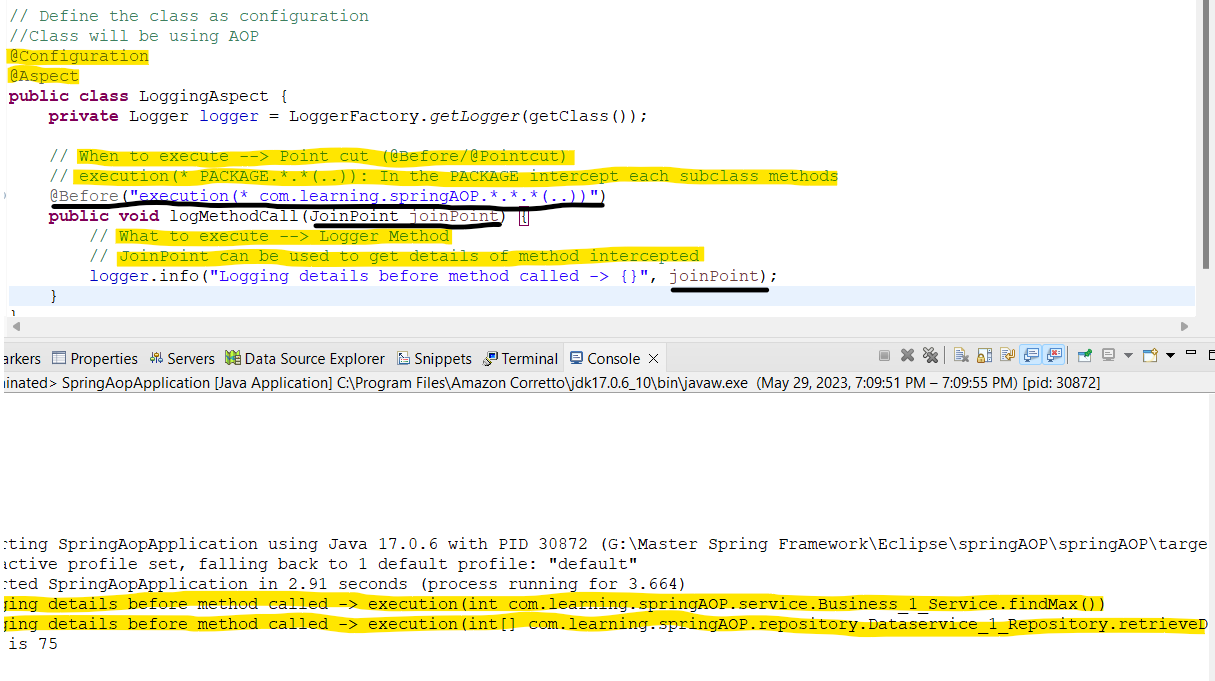
*<dependency>*

*<groupId>org.springframework.boot</groupId>*

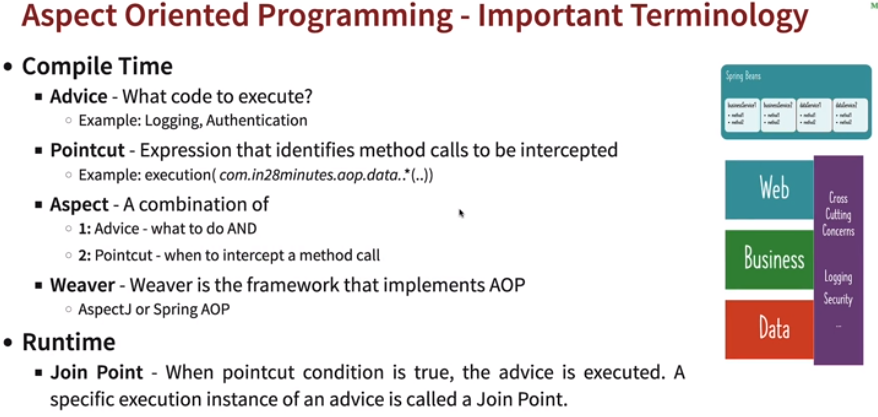
*<artifactId>spring-boot-starter-aop</artifactId>*

*</dependency>*

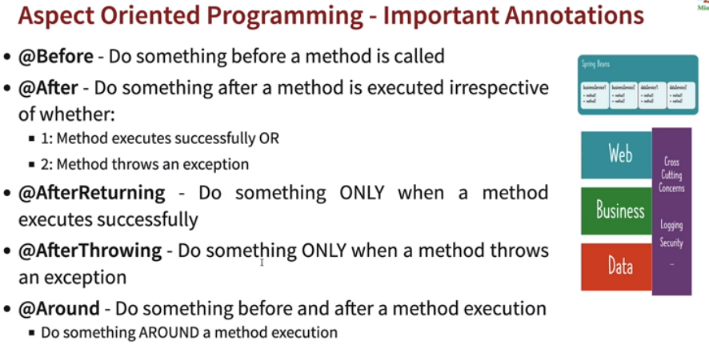
* + Create a class with *@Configuration* and annotate with *@Aspect* which declares class as aspect.
  + What kind of method can be intercepted are defined using *PointCut. @Pointcut()/@Before()*
  + Where to intercept methods: *execution(\* PACKAGE.\*.\*(..)) 🡪* In the package for all subclasses and methods.
  + *JoinPoint 🡪 JoinPoint* is a specification of when, in the corresponding program, the aspect code should be executed.



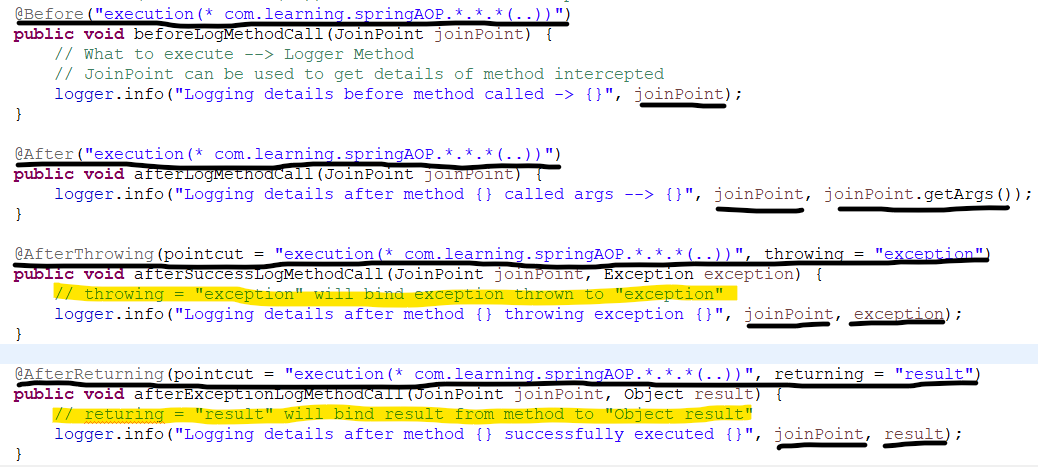
* **AOP Terms:**

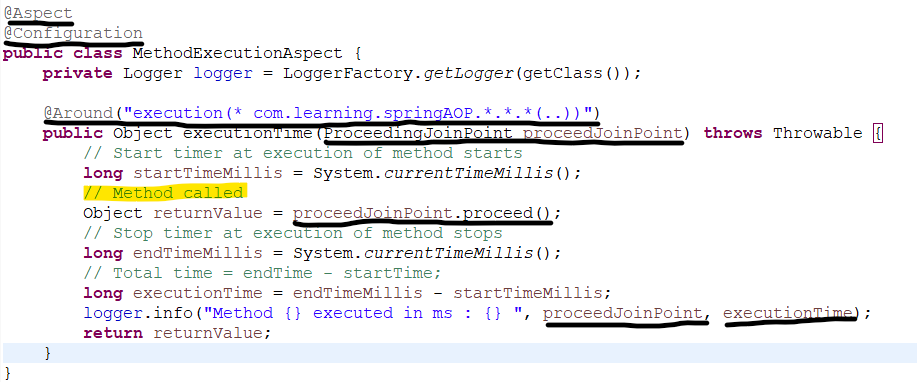


* + *Compile Time*
    - *Advice:* What to execute like logging, authentication etc.
    - *Pointcut:* Expression that identifies when methods to be intercepted.
    - *Aspect:* It is combination of *Advice* and *Pointcut.*
    - *Weaver: Spring AOP* and *AspectJ.*
  + *Run Time*
    - *Join Point:* When *Pointcut* condition is true, *Advice* is executed. A specific execution instance of an *Advice* is called *Joint Point.*

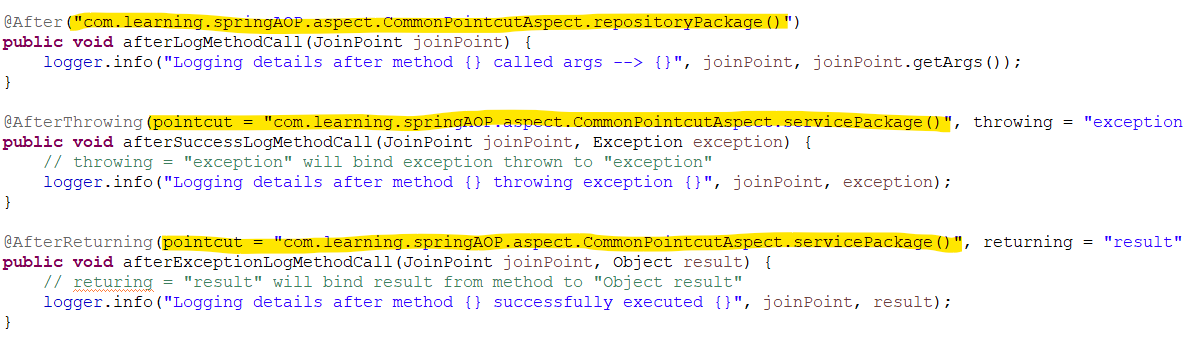


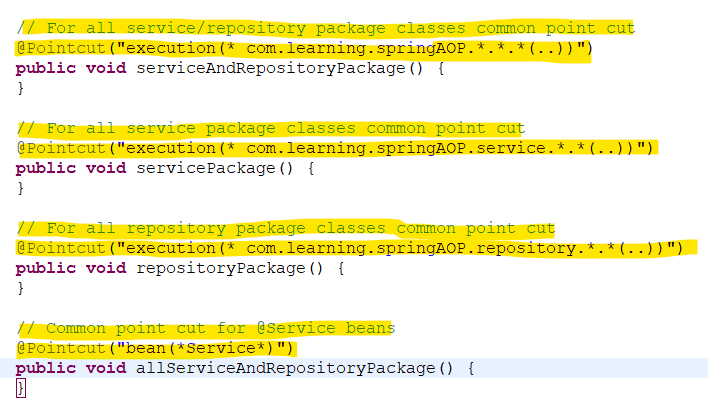
* **@Before:** Before method is called.
* **@After:** After method is called, whether method executed successfully or thrown an exception.
* **@AfterThrowing:** Only when method is throwing an exception.
* **@AfterReturning:** Only when method is executed successfully.
* **@After:** Before and after method called. Do something around method execution.
* **ProceedingJoinPoint:** *JoinPoint* can’t be used to called a method, so we can use *proceed(),* of *ProceedingJoinPoint* to called the method.
  + *ProceedingJoinPoint* is an extension of the *JoinPoint* that exposes the additional *proceed()* method. When invoked, the code execution jumps to the next advice or to the target method. It gives us the power to control the code flow and decide whether to proceed or not with further invocations.



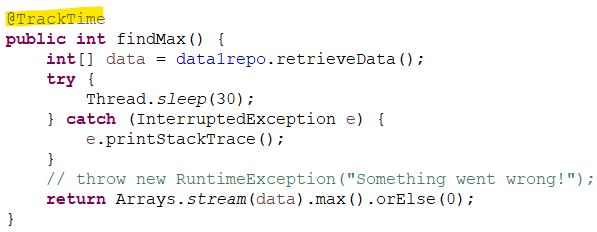


* **Creating a common pointcut:**
  + Create a class and define methods with *@Pointcut(“<Package\_name>”)*
  + In common aspects use the *copied qualified names* of *CommonAspect class.*

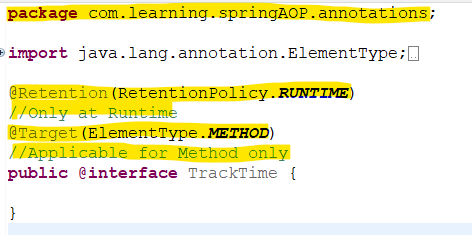




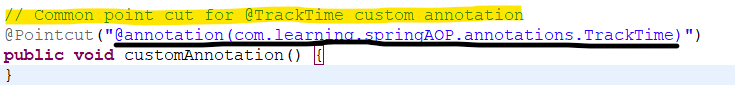
* **Using annotations as Pointcut:** 
  + Define a custom annotation in method. In this case *@TrackTime.*



* + Create a custom annotation.



* + Define a common point cut for annotation, *(“@annotation(“<annotation\_loc>”)”)*



* + Use this custom annotation point cut in other *AOPs.*

