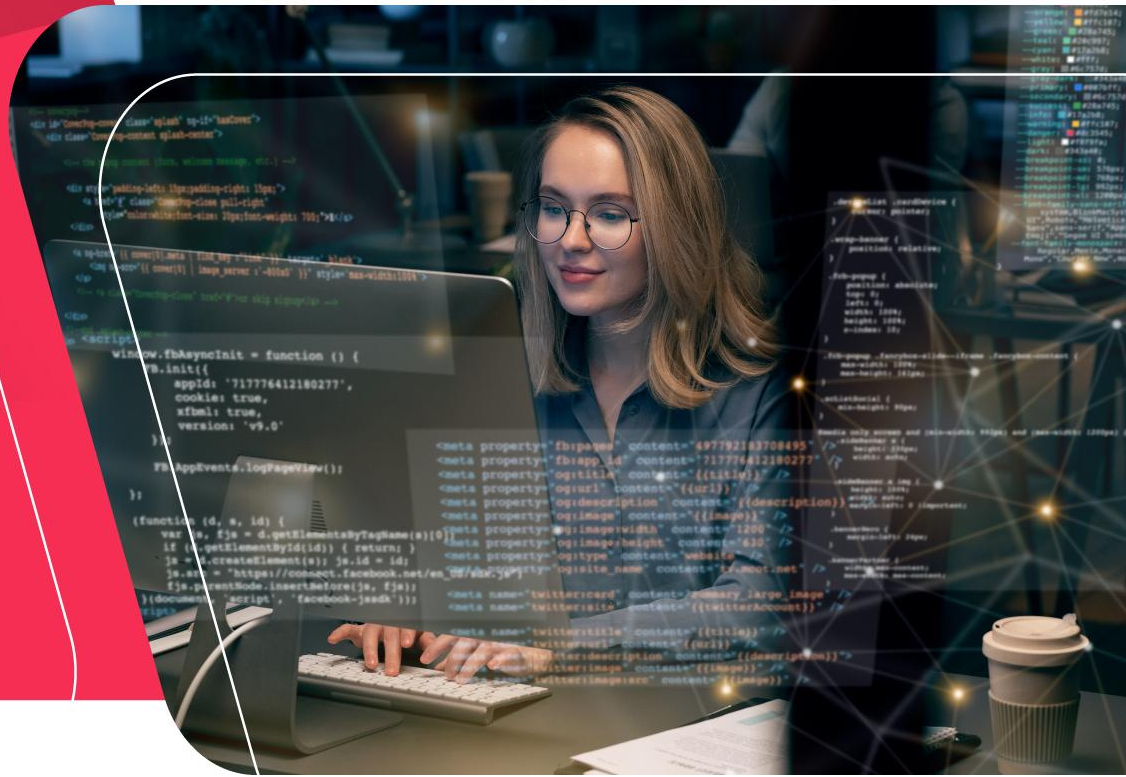
TIA-Academy

Java Bootcamp

Day 36



Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- **Intellij IDEA Community Edition**
- JAVA 3rd Party Library (Network, DB, etc)
- JAVA Framework (Spring Boot, Spring JPA & Spring JWT)
- **MySQL Server Community**

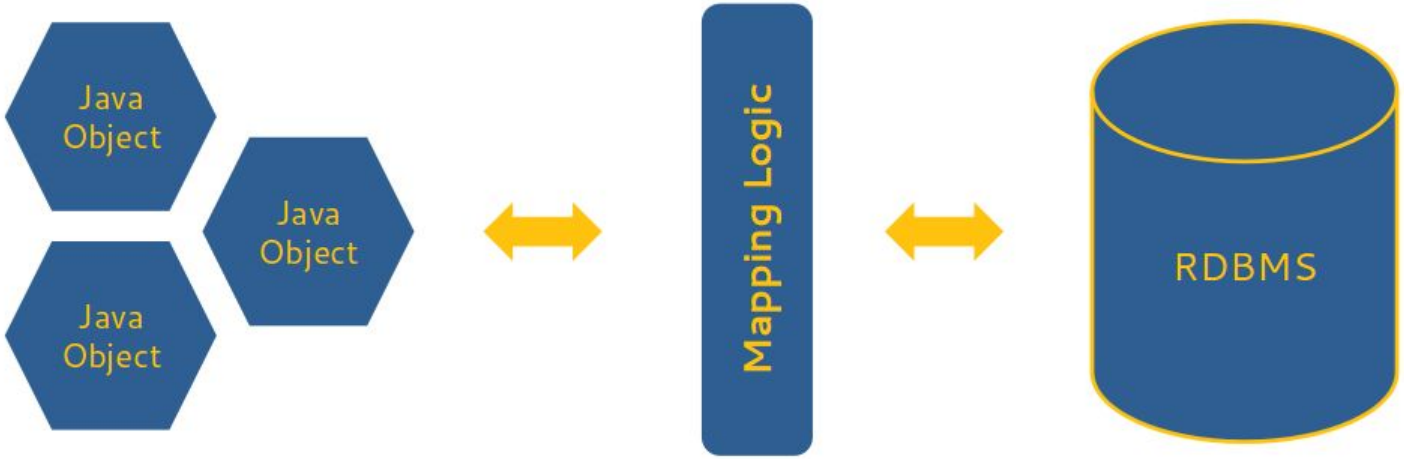
TIA-Academy

Java Persistence API

JPA and Hibernate

- As a specification, the Java Persistence API is concerned with *persistence*, which loosely means any mechanism by which Java objects outlive the application process that created them.
- Not all Java objects need to be persisted, but most applications persist key business objects.
- The JPA specification lets you define *which* objects should be persisted, and *how* those objects should be persisted in your Java applications.

JPA and Hibernate



JPA and Hibernate

- By itself, JPA is not a tool or framework; rather, it defines a set of concepts that can be implemented by any tool or framework.
- While JPA's object-relational mapping (ORM) model was originally based on Hibernate, it has since evolved. Likewise, while JPA was originally intended for use with relational/SQL databases, some JPA implementations have been extended for use with NoSQL datastores.

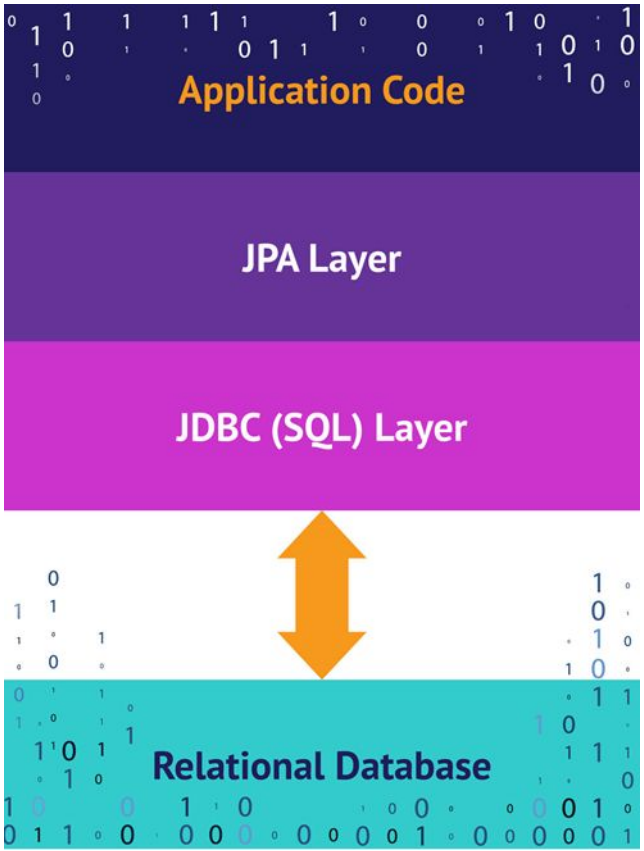
JPA and Hibernate

- Because of their intertwined history, Hibernate and JPA are frequently conflated. However, like the Java Servlet specification, JPA has spawned many compatible tools and frameworks; Hibernate is just one of them.
- Developed by Gavin King and released in early 2002, Hibernate is an ORM library for Java.
- King developed Hibernate as an alternative to entity beans for persistence. The framework was so popular, and so needed at the time, that many of its ideas were adopted and codified in the first JPA specification.
- Today, Hibernate ORM is one of the most mature JPA implementations, and still a popular option for ORM in Java. Hibernate ORM 5.3.8 (the current version as of this writing) implements JPA 2.2.

What is Java ORM?

- While they differ in execution, every JPA implementation provides some kind of ORM layer. In order to understand JPA and JPA-compatible tools, you need to have a good grasp on ORM.
- Object-relational mapping is a *task*—one that developers have good reason to avoid doing manually. A framework like Hibernate ORM or EclipseLink codifies that task into a library or framework, an *ORM layer*.
- As part of the application architecture, the ORM layer is responsible for managing the conversion of software objects to interact with the tables and columns in a relational database. In Java, the ORM layer converts Java classes and objects so that they can be stored and managed in a relational database.

What is Java ORM?



Why Do We Need Automated Persistence? – The Case for Hibernate

- Let's first answer the question why we would even use a framework to persist application data. Is JDBC (Java Database Connectivity) not good enough for us?
- Well, if we look at the data held by our applications, we'll usually see a complex network of objects, also known as the object graph, that we want to write to and read from some store, usually a relational database.
- Unfortunately, the persistence process is far from straightforward. Translating entire objects graphs to plain tabular data is everything but simple and storing objects using plain JDBC is painful and cumbersome, as the standard exposes a low-level API.

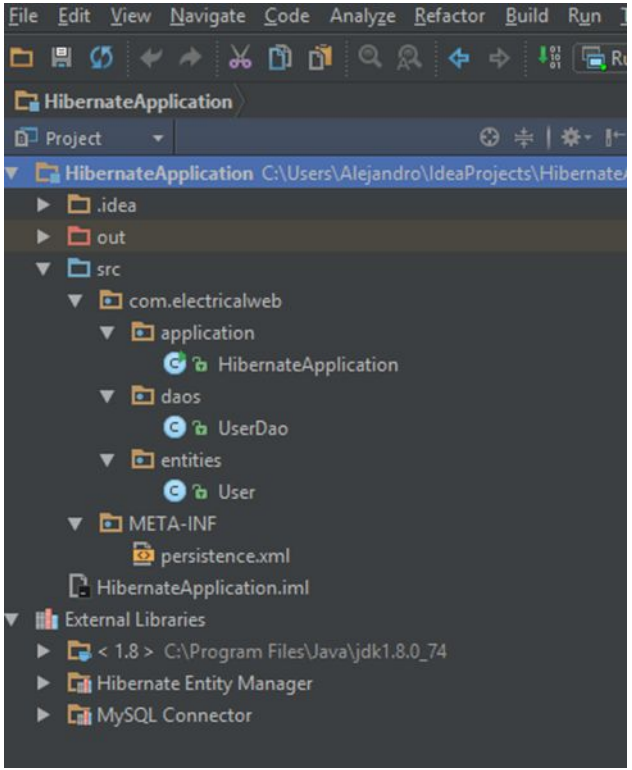
Why Do We Need Automated Persistence? – The Case for Hibernate

- You know, connecting to a database and writing SQL queries from scratch is all well and fine. But be honest with yourself: Do you want to do that over and over again? If you're anything like me, you don't.
- And this is where Hibernate comes in. Besides a lot of other things, it allows us to create the objects that hold our data, called entities, as we are used to and automates the process of reading and writing them.

- A nice way to start pulling the reins of Hibernate is to create a simple example. To do so, we are going to develop a Java application, whose functionality will boil down to performing CRUD operations on some user objects.
- In turn, the objects will be persisted, fetched, updated and removed from a MySQL database.
- To get things rolling as expected, we first need to grab the required Hibernate ORM JARs, along with the corresponding MySQL Java Connector. Maven will do that nicely for us, so we can focus on coding, rather than on dealing with dependency issues.

pom_jpa.xml

- With the POM file under the way, here's how this sample application will be laid out:
- The application's skeleton is structured in only three packages: **`com.electricalweb.entities`**, which contains the domain class that models users, **`com.electricalweb.daos`**, which houses a basic user DAO class, and lastly **`com.electricalweb.application`**, which is the application's entry point.
- Since the primary goal is to persist user objects in MySQL, let's define the User domain class.



- To keep things simple, the domain class will be extremely anemic, having only a couple of private properties, `name` and `email`, along with a parameterized constructor and the typical setters/getters. Here's the class in question:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;
    private String email;

    public User() {}

    public User(String name, String email) {
        this.setName(name);
        this.setEmail(email);
    }
    // getters and setters for name and email
}
```

- The above class is just a lightweight data container with no behavior attached to it. The only detail worth to stress here is that in order to make it persistable for Hibernate (or any other JPA implementation) it's been marked with an `@Entity` annotation. In addition,
- **Hibernate needs to know how to handle the entities' primary keys.** To tackle this, the `@Id` and `@GeneratedValue(strategy = GenerationType.AUTO)` annotations instruct Hibernate to automatically generate an ID for each user entity, which will be mapped to the primary key of the corresponding database entry.
- Finally, the `@Table(name = "users")` annotation tells Hibernate to map instances of the class to rows in a `users` table.
- With the domain class already defined, the final step to take in order to get the sample application up and running is create a configuration file, not surprisingly called persistence.xml, which will contain all the data required for connecting to the database and mapping entities to the database.

Accessing the Database – Creating a Persistence Unit

- Unquestionably, the most common way of setting up the connection parameters to the database, along with additional Hibernate proprietary options, is by defining the aforementioned **persistence.xml** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
    <persistence-unit name="user-unit" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.electricalweb.entities.User</class>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/usersdb" />
            <property name="javax.persistence.jdbc.user" value="username" />
            <property name="javax.persistence.jdbc.password" value="password" />
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>
</persistence>
```

Defining a Persistable Entity – Hibernate's Core Feature

- As you can see, the file is easily readable from top to bottom. It just sets up a few database connection parameters, which you should be familiar with if you've worked with JDBC before, including the database URL, the JDBC driver, the database username and the password (make sure to provide the right values for your own database server).

- Additionally, and this is by far the most interesting segment of the file, it defines a persistence-unit, that is a set of persistence entities (in this case just users, but it could be multiple entities wrapped inside several `<class>` tags) which are handled by a JPA entity manager. Considering that in this case Hibernate is the chosen JPA implementation, the entities will be managed by Hibernate's entity manager.

- Then you'll be eager to see how to use the entity manager for performing CRUD operations on a few user entities in a snap.

- In a nutshell, running CRUD operations on user objects is much easier than one might think.
- The entire process just boils down to spawning the **`EntityManager`** and consuming its API. It's really that easy.
- For instance, we could use the application's entry point, which is a simple class with a plain static main method, and call the entity manager in the following way:

Performing CRUD Operations on Entities – Working with Hibernate's Entity Manager

```
EntityManager entityManager = Persistence
        .createEntityManagerFactory("user-unit")
        .createEntityManager();
EntityTransaction entityTransaction =
entityManager.getTransaction();
```

```
/* Persist a User entity */
entityTransaction.begin();
User user = new User("Alejandro", "alejandro@domain.com");
entityManager.persist(user);
entityTransaction.commit();
```

```
/* Fetch a User entity */
entityManager.find(User.class, 1);
```

```
/* Update a User entity */
entityTransaction.begin();
User user = entityManager.find(User.class, 1);
user.setName("Alex");
user.setEmail("alex@domain.com");
entityManager.merge(user);
entityTransaction.commit();
```

```
/* Remove a User entity */
entityTransaction.begin();
User user = entityManager.find(User.class, 1);
entityManager.remove(user);
entityTransaction.commit();

entityManager.close();
```

- Quite simple, isn't it? Saving, fetching, updating and deleting user entities with a non-managed entity manager is a no-brainer process that can be mastered literally within minutes.
- The procedure is limited to first getting the manager via an **`EntityManagerFactory`**, then grabbing an instance of the **`EntityTransaction`** class, and finally calling the desired method of the entity manager.
- The above code snippet assumes there's a MySQL database table called users already defined, together with a MySQL instance running on a local server. So, make sure to set up that before testing the sample application.

- At this point, we have learned the basics behind persisting single entities with Hibernate. So, what comes up next?

  Well, a lot actually. As you may have noticed, something left out of the picture persisting entities that have a specific relationship with other entities (yes, the relational aspect).

- In order to keep the whole learning process free from additional, unnecessary complexities. After all, this is just an introduction to getting started using Hibernate. Still, Now to polish the demo a little bit, as in its current state it looks pretty monolithic and its execution flow can't be controlled through a user interface.

- Taking this into consideration, **what we'll do next will be defining a DAO class, which will encapsulate the handling of the entity manager and the entity transaction behind the fences of a declarative API.** Finally, we'll see how to use the class for selectively executing CRUD methods on user entities, based on options entered in the Java console.

Adding a Nice API – Defining a DAO Class

- Believe me. I'm not a blind worshiper of design patterns, using them everywhere without having strong reasons to do so. In this case, however, it's worth to appeal to the functionality provided by the DAO pattern, so you can give the previous example a try without much hassle.

- In a nutshell, all that the following **`UserDao`** class does is to define an easily consumable API, which allows to execute CRUD methods on a given user entity. Here's how the class looks:

UserDao.java

Adding a Nice API – Defining a DAO Class

- Honestly, this additional layer of abstraction (and complexity) wouldn't make much sense per see if I wouldn't show you how to use the class in a concrete case. So, check the following snippet, which exploits the benefits of dependency injection.

HibernateApplication.java

Adding a Nice API – Defining a DAO Class

- As shown above, the HibernateAplication class accepts a range of four different values entered in the console. According to the given input, it uses the DAO for running a specific CRUD operation on a user entity.
- Needless to say the logic of this revamped example is self-explanatory, so any further analysis would be redundant.
- Thus, make sure to give it a try on your own development platform and feel free to twist it at will to suit your personal needs.

TIA-Academy

ASSIGNMENT 01

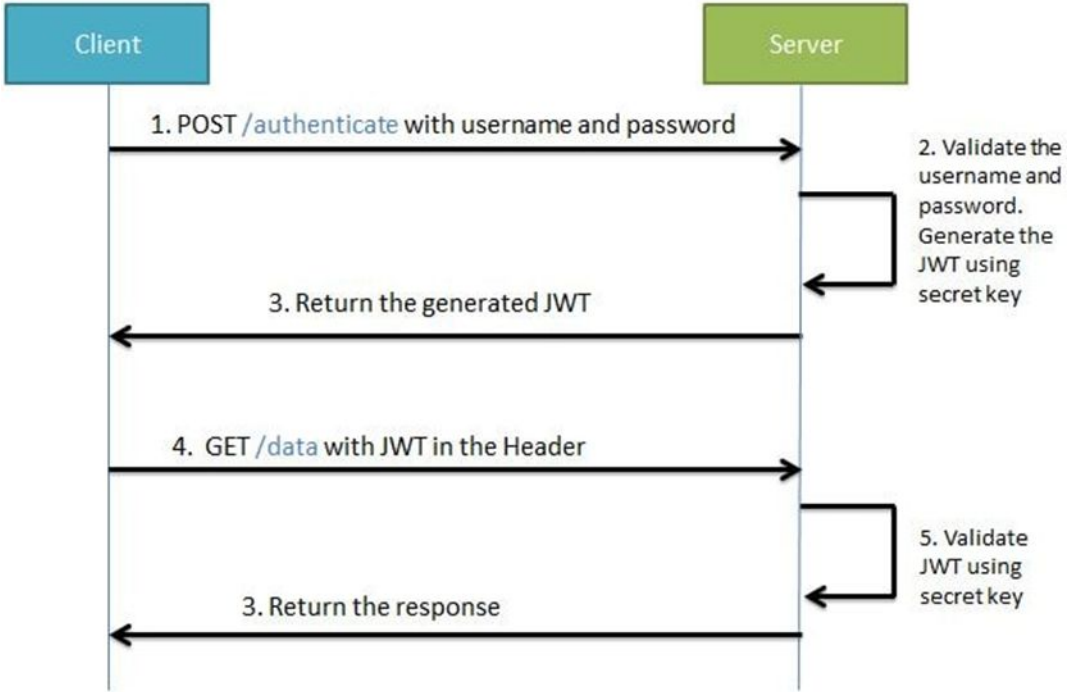
TIA-Academy

JWT

What is JWT?

- JWT stands for JSON Web Token. JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- This information can be verified and trusted because it is digitally signed.
- **The client will need to authenticate with the server using the credentials only once**. During this time the server validates the credentials and returns the client a JSON Web Token(JWT).
- For all future requests the client can authenticate itself to the server using this JSON Web Token(JWT) and so does not need to send the credentials like **username and password**.

Workflow of how JWT is used



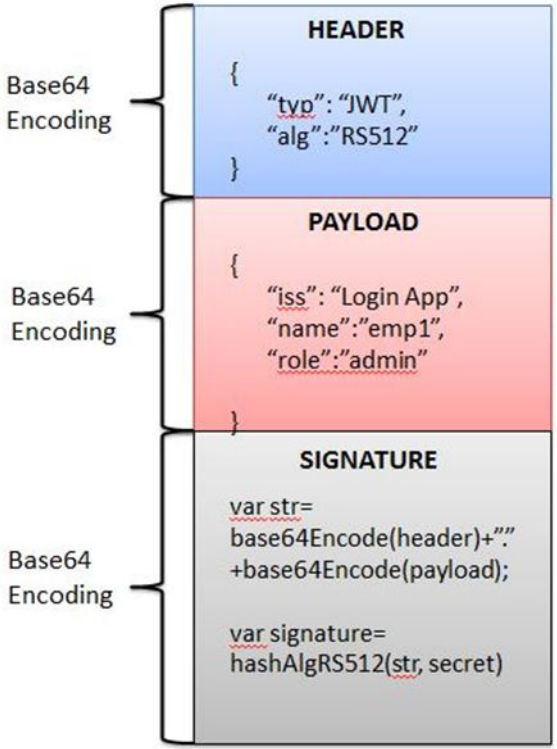
Workflow of how JWT is used

- During the first request the client sends a POST request with username and password.
- Upon successful authentication the server generates the JWT sends this JWT to the client. This JWT can contain a payload of data.
- On all subsequent requests the client sends this JWT token in the header. Using this token the server authenticates the user. So we don't need the client to send the user name and password to the server during each request for authentication, but only once after which the server issues a JWT to the client.
- A JWT payload can contain things like user ID so that when the client again sends the JWT, you can be sure that it is issued by you, and you can see to whom it was issued.

Structure of JWT

- JWT has the following format -**header.payload.signature**

Structure of JWT



The header consists of 2 fields. The typ whose value is JWT and the hashing algorithm to be used.

Payload consists of data to be exchanged securely This information represents claims. The Claim Names within this object MUST be unique

Claim Name
exp
nbf
iat
iss
aud
prn
jti
typ

Signature is the hash of the header and the payload using the hashing algorithm specified in the the Header and the secret which is known only to the server

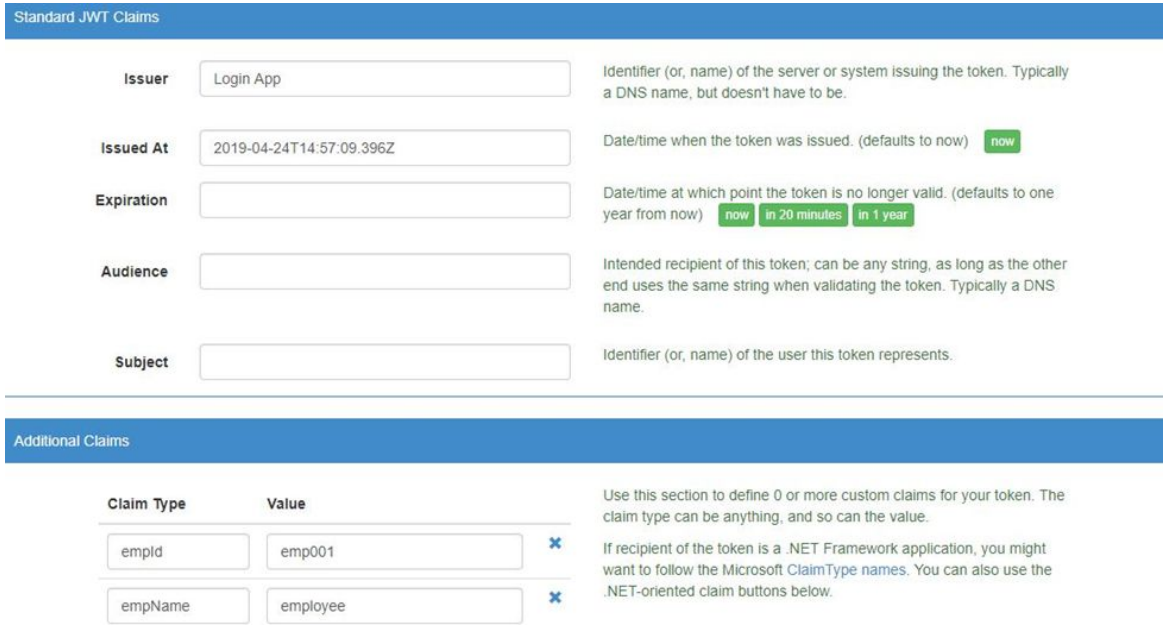
Structure of JWT

- An important point to remember about JWT is that the information in the **payload of the JWT is visible to everyone**. So we should not pass any sensitive information like passwords in the payload.
- We can encrypt the payload data if we want to make it more secure.
- However we can be sure that no one can tamper and change the payload information. If this is done the server will recognize it.

Creating a JWT Token

- We will be creating a JWT token using https://www.javainuse.com/jwtgenerator

  Specify the payload data as folows-



Creating a JWT Token

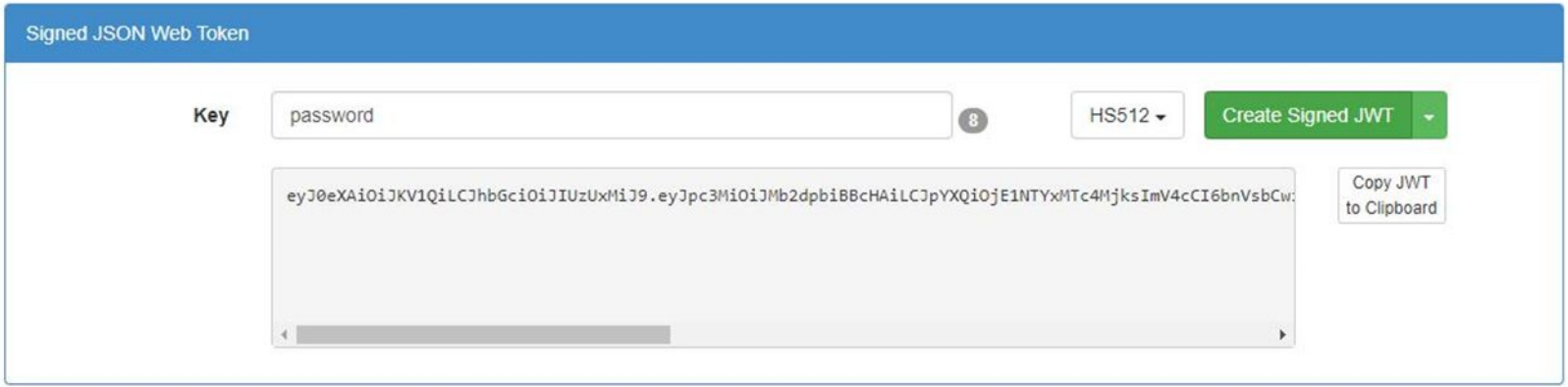
- We will be having following claims in the payload-

Generated Claim Set (plain text)

```
{
    "iss": "Login App",
    "iat": 1556117829,
    "exp": null,
    "aud": "",
    "sub": "",
    "empId": "emp001",
    "empName": "employee"
}
```

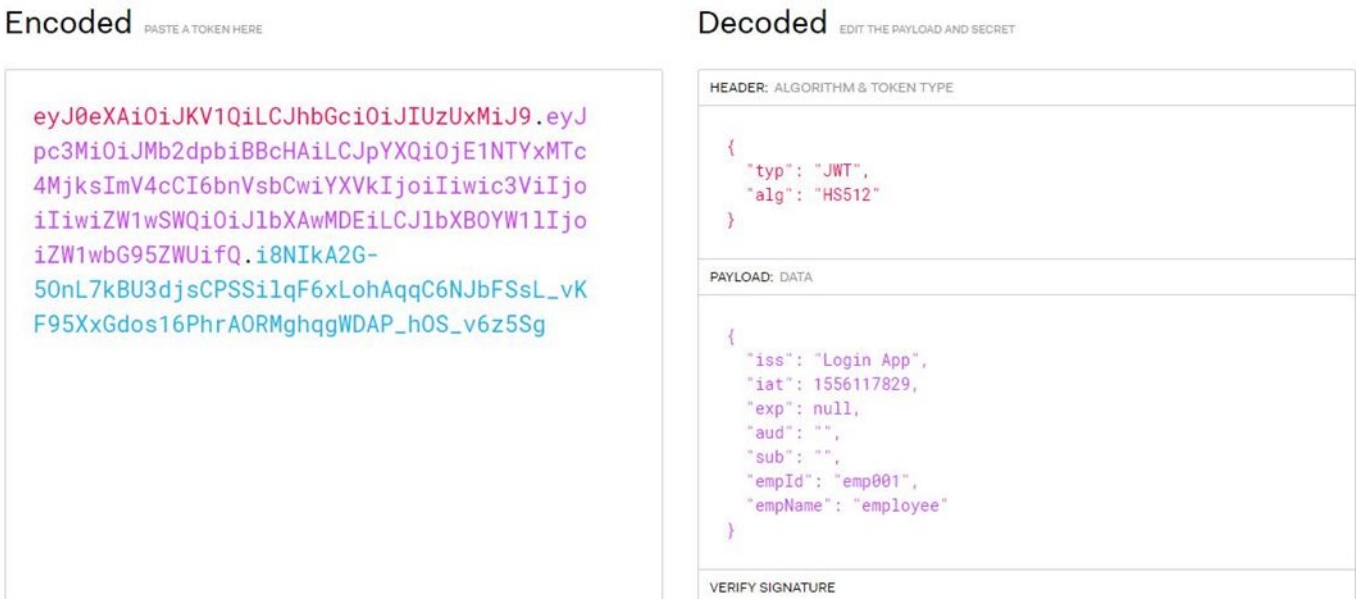
Creating a JWT Token

- Sign the payload using the hashing algorithm-



Inspect the contents of the created token

- We will be inspecting JWT token using [JWT Online Decoder]



TIA-Academy

Example Without JWT

Spring Boot Security + JWT Hello World Example

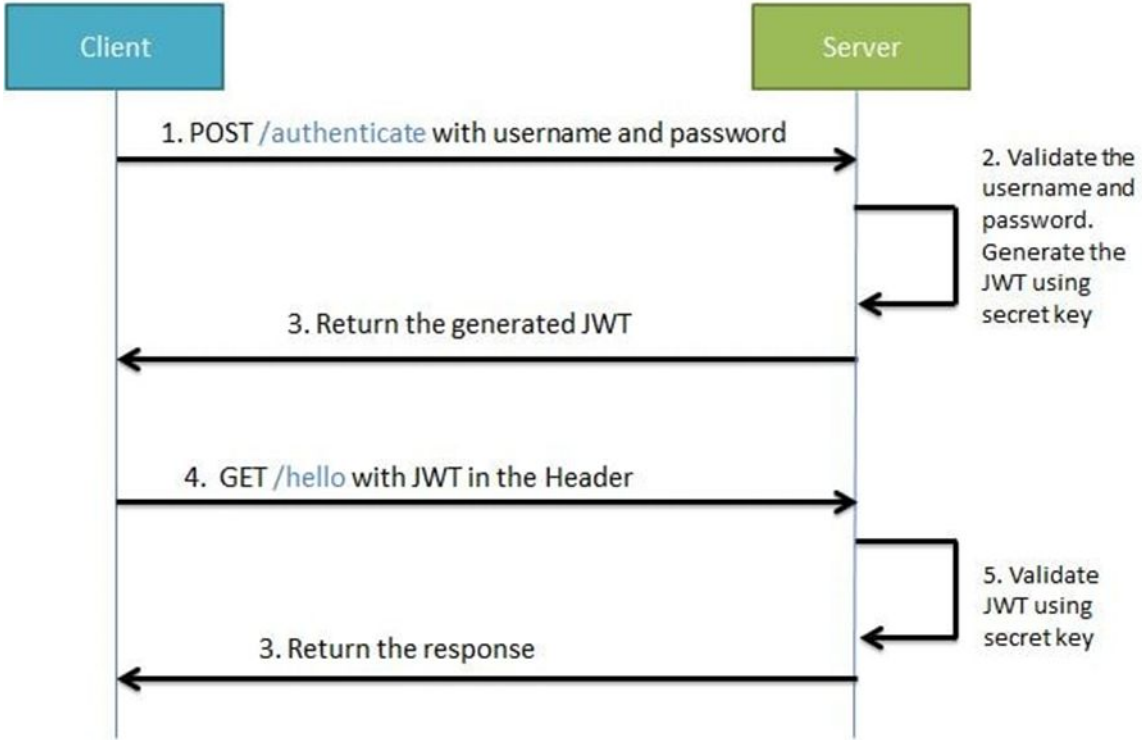
- In this tutorial we will be developing a Spring Boot Application that makes use of JWT authentication for securing an exposed REST API.
- In this example we will be making use of **hard coded user values for User Authentication.**

Spring Boot Security + JWT Hello World Example

For better understanding we will be developing the project in stages

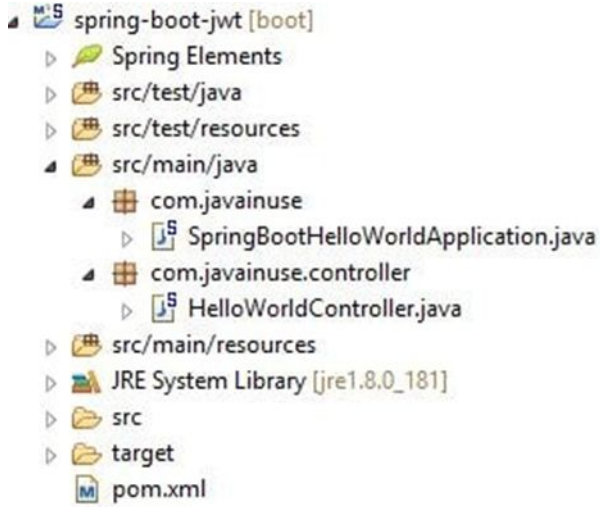
- Develop a Spring Boot Application to expose a Simple **REST GET API with mapping /hello.**
- Configure Spring Security for JWT. Expose **REST POST API with mapping /authenticate** using which User will get a valid JSON Web Token.
- And then **allow the user access to the api /hello only if it has a valid token.**

Spring Boot Security + JWT Hello World Example



Develop a Spring Boot Application to expose a GET REST API

- Maven Project will be as follows-



- The pom.xml is as follows-

pom.xml

Develop a Spring Boot Application to expose a GET REST API

- Create a Controller class for exposing a GET REST API-

```
package com.javainuse.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloWorldController
{
	@RequestMapping({ "/hello" })
	public String firstPage() {
		return "Hello World";
	}
}
```

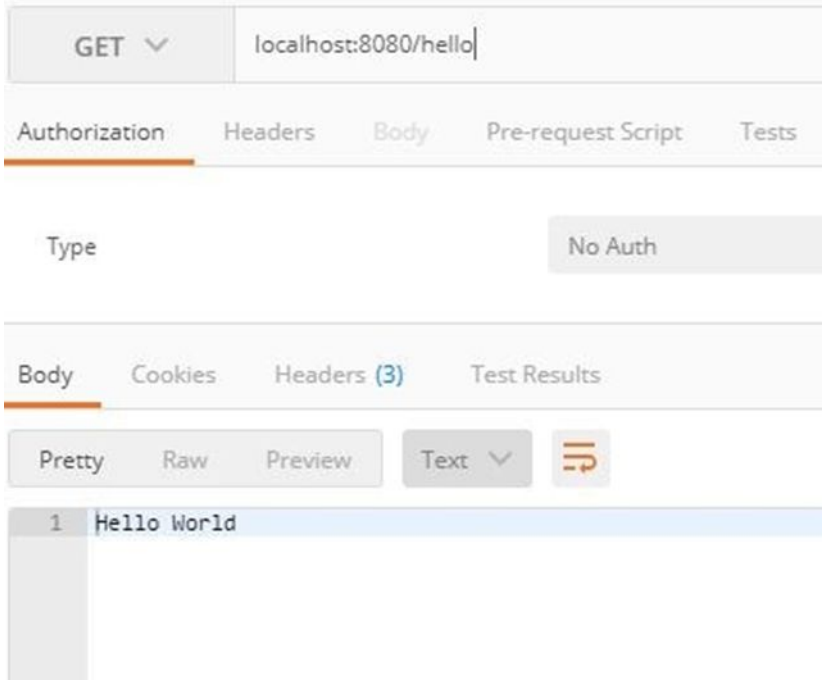
Develop a Spring Boot Application to expose a GET REST API

- Create the bootstrap class with SpringBoot Annotation

```
package com.javainuse;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootHelloWorldApplication {

        public static void main(String[] args) {
        SpringApplication.run(SpringBootHelloWorldApplication.class,
        args);
        }

}
```

Develop a Spring Boot Application to expose a GET REST API

- Compile and the run the SpringBootHelloWorldApplication.java as a Java application.
- Go to **localhost:8080/hello**



TIA-Academy

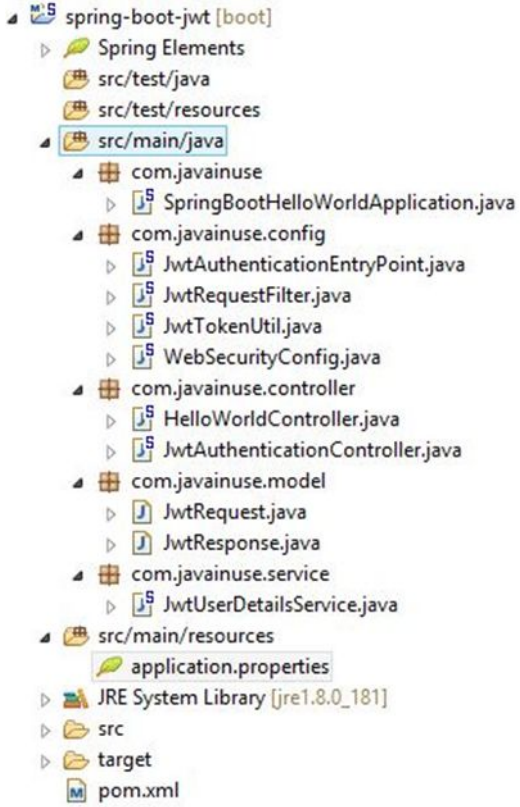
Applying JWT

Spring Security and JWT Configuration

- We will be configuring Spring Security and JWT for performing 2 operations-
  - **Generating JWT** - Expose a POST API with mapping **/authenticate**. On passing correct username and password it will generate a JSON Web Token(JWT)
  - **Validating JWT** - If user tries to access GET API with mapping **/hello**. It will allow access only if request has a valid JSON Web Token(JWT)

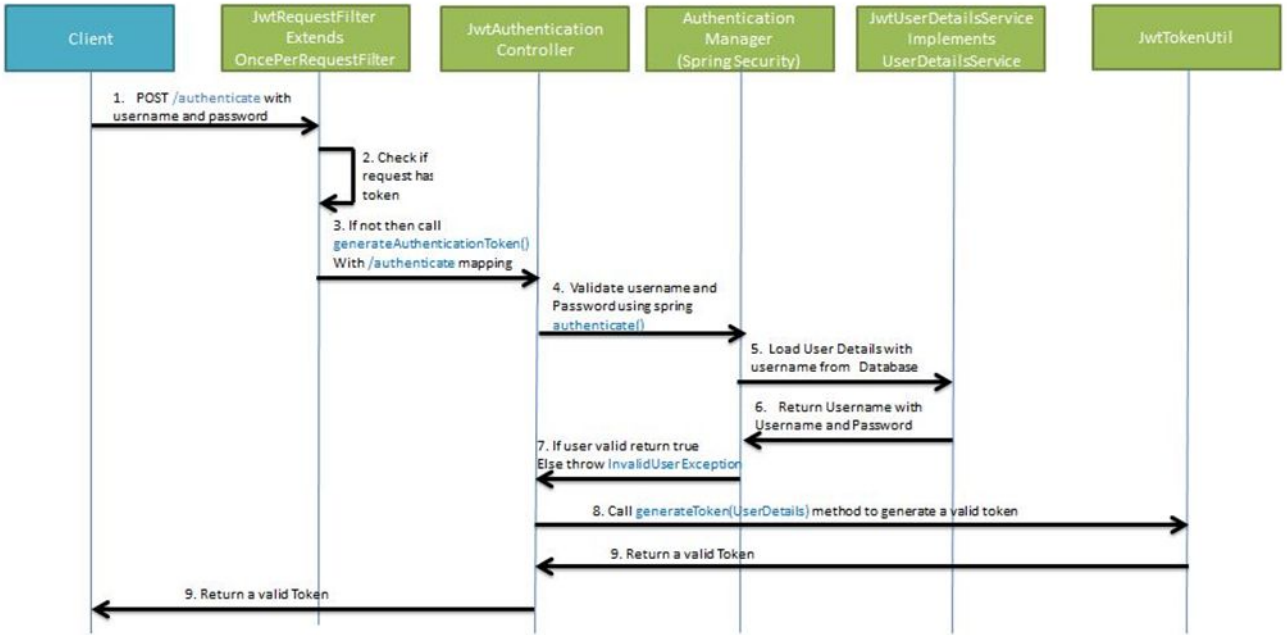
Spring Security and JWT Configuration

- Maven Project will be as follows-



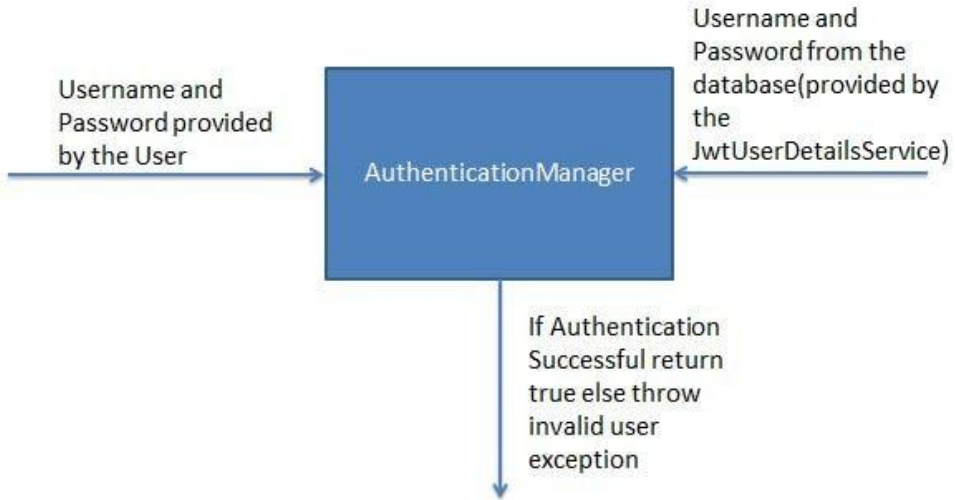
Spring Security and JWT Configuration

- Generating JWT



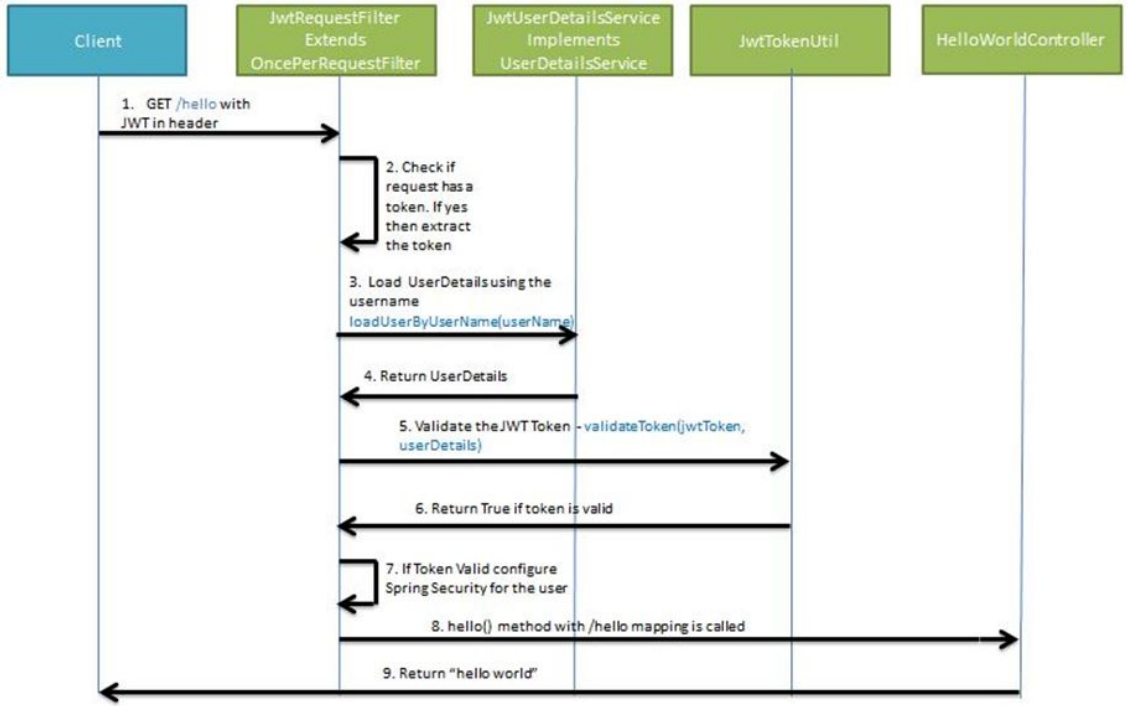
Spring Security and JWT Configuration

- Generating JWT



Spring Security and JWT Configuration

- Validating JWT



Spring Security and JWT Configuration

- Add the Spring Security and JWT dependencies on **Pom.xml**

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
                <groupId>io.jsonwebtoken</groupId>
                <artifactId>jjwt</artifactId>
                <version>0.9.1</version>
        </dependency>
</dependencies>
```

Spring Security and JWT Configuration

- Define the **application.properties**.
- The secret key is combined with the header and the payload to create a unique hash.

  We are only able to verify this hash if you have the secret key.

```
jwt.secret=javainuse
```

application.properties

Spring Security and JWT Configuration

- **JwtTokenUtil**, is responsible for performing JWT operations like creation and validation.
- It makes use of the io.jsonwebtoken.Jwts for achieving this.

JwtTokenUtil.java

Spring Security and JWT Configuration

- **JWTUserDetailsService** implements the Spring Security UserDetailsService interface.
- It overrides the loadUserByUsername for fetching user details from the database using the username.
- The Spring Security Authentication Manager calls this method for getting the user details from the database when authenticating the user details provided by the user. Here we are getting the **user details from a hardcoded User List**.
- Also the password for a user is stored in encrypted format using BCrypt. Here using the Online Bcrypt Generator you can generate the Bcrypt for a password.

JwtUserDetailsService.java

Spring Security and JWT Configuration

- Expose a POST API /authenticate using the **JwtAuthenticationController**.
- The POST API gets username and password in the body- Using Spring Authentication Manager we authenticate the username and password.
- If the credentials are valid, a JWT token is created using the **JWTTokenUtil** and provided to the client.

JwtAuthenticationController.java

Spring Security and JWT Configuration

- **JwtRequest**, this class is required for storing the username and password we recieve from the client.

JwtRequest.java

Spring Security and JWT Configuration

- **JwtResponse**, this is class is required for creating a response containing the JWT to be returned to the user.

JwtResponse.java

Spring Security and JWT Configuration

- The **JwtRequestFilter** extends the Spring Web Filter **OncePerRequestFilter** class. **For any incoming request this Filter class gets executed**.
- It checks if the request has a valid JWT token. **If it has a valid JWT Token then it sets the Authentication in the context**, to specify that the current user is authenticated.

JwtRequestFilter.java

Spring Security and JWT Configuration

- **JwtAuthenticationEntryPoint**, this class will extend Spring's **AuthenticationEntryPoint** class and override its method commence.
- **It rejects every unauthenticated request and send error code 401.**

JwtAuthenticationEntryPoint.java

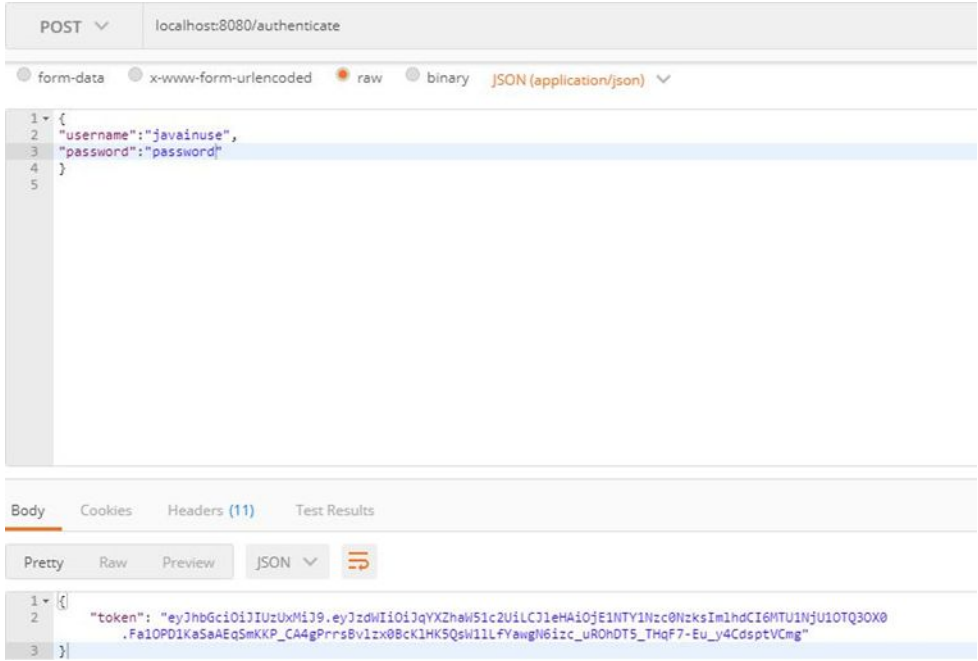
Spring Security and JWT Configuration

- **WebSecurityConfig**, this class extends the **WebSecurityConfigurerAdapter** is a convenience class that **allows customization to both WebSecurity and HttpSecurity.**

WebSecurityConfig.java

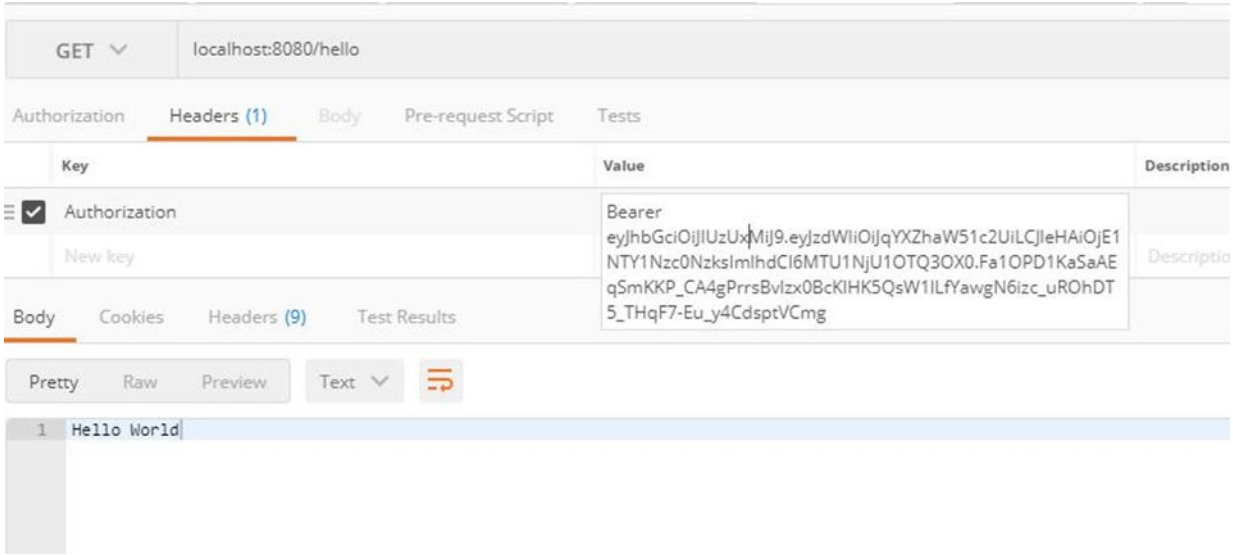
Spring Security and JWT Configuration

- Start the Spring Boot Application to generate a JSON Web Token –
- Create a POST request with url **localhost:8080/authenticate**. Body should have valid username and password. In our case **username is javainuse** and **password is password**.



Spring Security and JWT Configuration

- Validate the JSON Web Token by accessing the url **localhost:8080/hello** using the above generated token in the header as follows



TIA-Academy

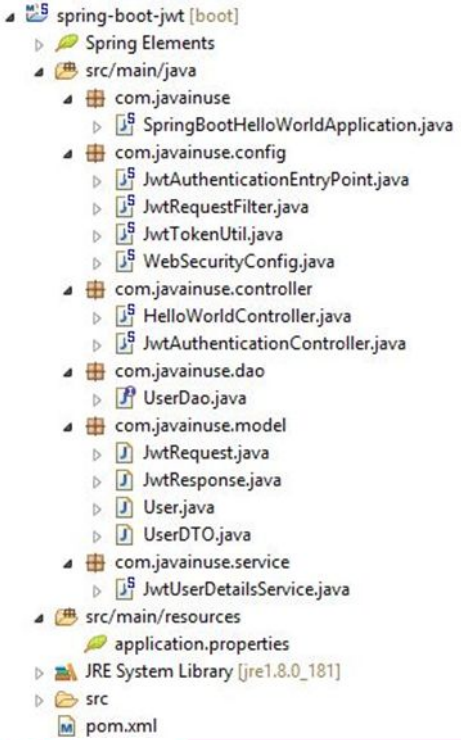
JWT + JPA

Spring Boot Security + JWT + MySQL Hello World Example

- In a previous tutorial we were making use of **hard coded user values** for User Authentication.
- In this tutorial we will be **implementing MYSQL JPA for storing and fetching user credentials.**
- The starting code for this tutorial will be the the one we had implemented previously.
- Currently using **JwtUserDetailsService** we are **validating the user**. We are doing this using **hard coded values for username and password**.
- Now we will be using Spring Data JPA to validate user credentials by fetching username and password from the mysql db.

Implement MYSQL JPA for storing and fetching user details

- Maven Project will be as follows-



- The pom.xml is as follows-

pom.xml

Implement MYSQL JPA for storing and fetching user details

- Inserting a user, define the database properties in **application.properties** as follows-

```
jwt.secret=javainuse
spring.datasource.url=jdbc:mysql://localhost/bootdb?createDatabaseIfNotExist=true&auto
Reconnect=true&useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.platform=mysql
spring.jpa.hibernate.ddl-auto=create-drop
```

application.properties

Implement MYSQL JPA for storing and fetching user details

- In a previous tutorial we had implemented **Spring Boot + JWT Hello World Example.**
- Create the Entity class **DAOUser** as follows. It will be used while performing database operations-

DAOUser.java

Implement MYSQL JPA for storing and fetching user details

- Define the **UserDTO** model class as follows.
- It is responsible for getting values from user and passing it to the **DAO layer** for inserting in database.

UserDTO.java

- For more information about **DAO** and **DTO**
  - https://stackoverflow.com/questions/14366001/dto-and-dao-concepts-and-mvc
  - https://www.javatpoint.com/q/2615/what-is-the-difference-between-dao-and-dto-files-----

Implement MYSQL JPA for storing and fetching user details

- Next we define the **UserDao** which is an interface that extends the Spring Framework class **CrudRepository**. **CrudRepository** class is a generics and takes the following two parameters as arguments-
- What type of Object will this repository be working with-
- In our case **DAOUser** and **Id** will be what type of object-
- Integer(since **id** defined in the **UserDao** class is **Integer**) Thats the only configuration required for the repository class.
- **The required operation of inserting user details in DB will now be handled.**

Implement MYSQL JPA for storing and fetching user details

- Define the **UserDao** class as follows.

UserDao.java

Implement MYSQL JPA for storing and fetching user details

- Update the **WebSecurityConfig** to allow the url **/register** to be **allowed without applying spring security-**

WebSecurityConfig.java

Implement MYSQL JPA for storing and fetching user details

- In the **JwtUserDetailsService**, autowire the **UserDao bean** and the **BcryptEncoder bean**.
- Also define the **saveUser function** for inserting user details-

JwtUserDetailsService.java

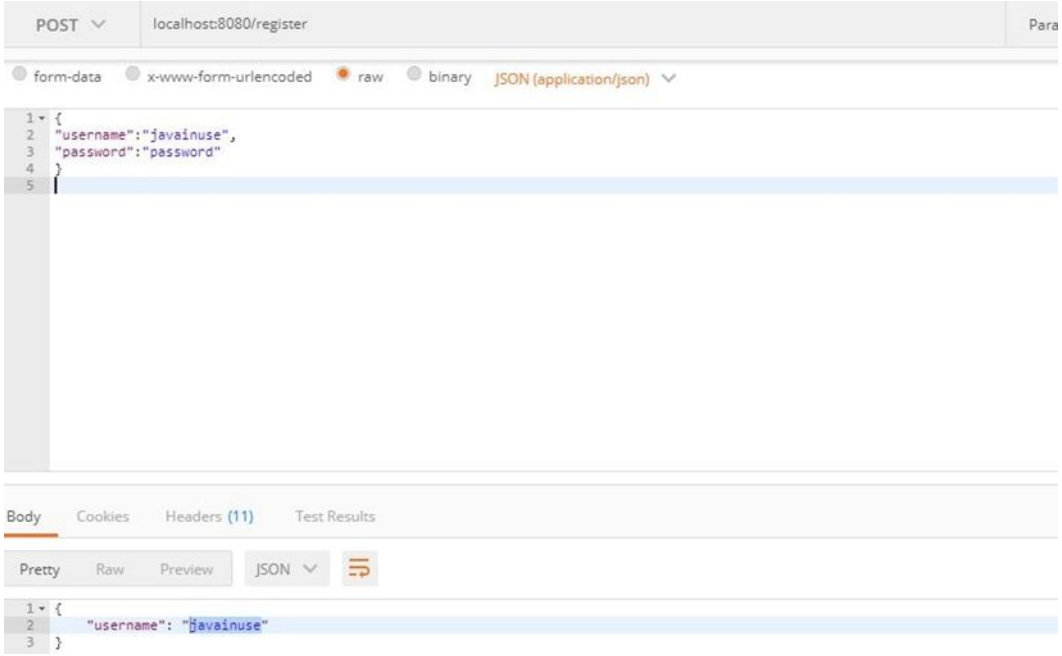
Implement MYSQL JPA for storing and fetching user details

- Finally modify the **JwtAuthenticationController** class for adding a POST request for adding user details to database.

JwtAuthenticationController.java

Implement MYSQL JPA for storing and fetching user details

- Start the Spring Boot Application- **Register a new user** by **creating a post request** to url **/register** and the body having username and password.



Make use of Database credentials for authentication

- In the UserDao interface add a method findByUsername(String username)

```
package com.javainuse.dao;

import
org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.javainuse.model.DAOUser;

@Repository
public interface UserDao extends
CrudRepository<DAOUser, Integer> {
     DAOUser findByUsername(String username);
}
```

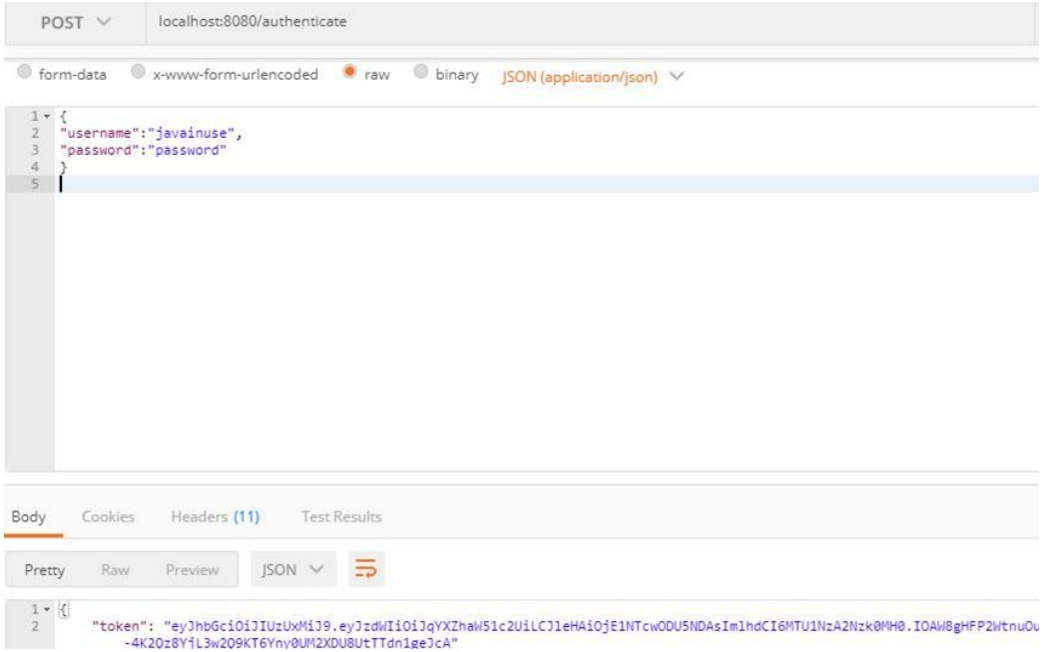
Make use of Database credentials for authentication

- In the **loadUserByUsername** method inside **JwtUserDetailsService** class, we fetch the user records from the database instead of using hardcoded value.

```
@Override
public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        DAOUser user = userDao.findByUsername(username);
        if (user == null) {
        throw new UsernameNotFoundException("User not found
        with username: " + username);
        }
        return new
        org.springframework.security.core.userdetails.User(u
        ser.getUsername(), user.getPassword(), new
        ArrayList<>());
}
```

Make use of Database credentials for authentication

- Generate a new Token by creating a **post request to url /authenticate** and the body having username and password.



TIA-Academy

ASSIGNMENT 00
(HOME ASSIGNMENT)

Sources

- https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html
- https://www.sitepoint.com/hibernate-introduction-persisting-java-objects/
- https://www.javainuse.com/spring/boot-jwt
- https://www.javainuse.com/spring/boot-jwt-mysql

TIA-Academy

Thank You