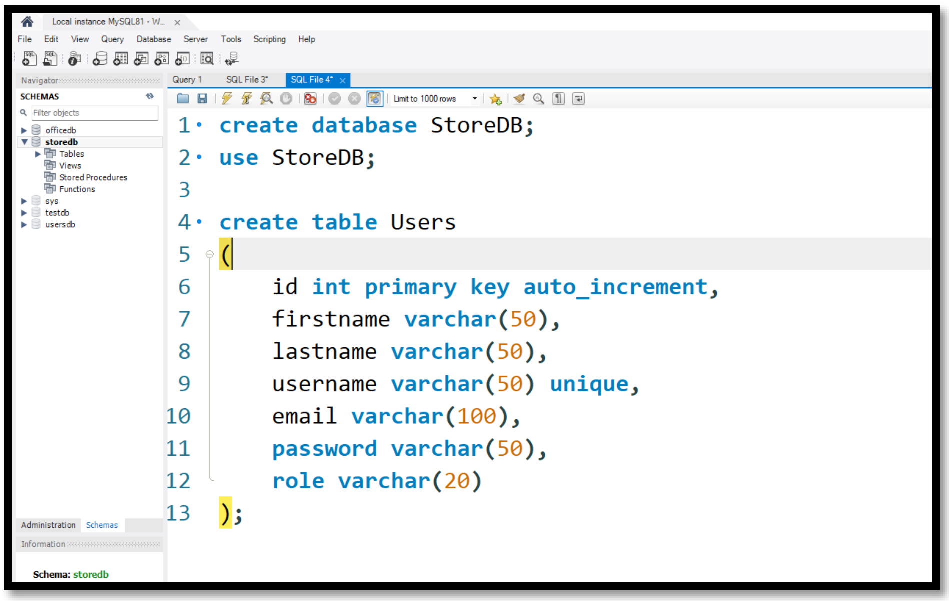
Rest API with JWT Token Authentication using OAuth2

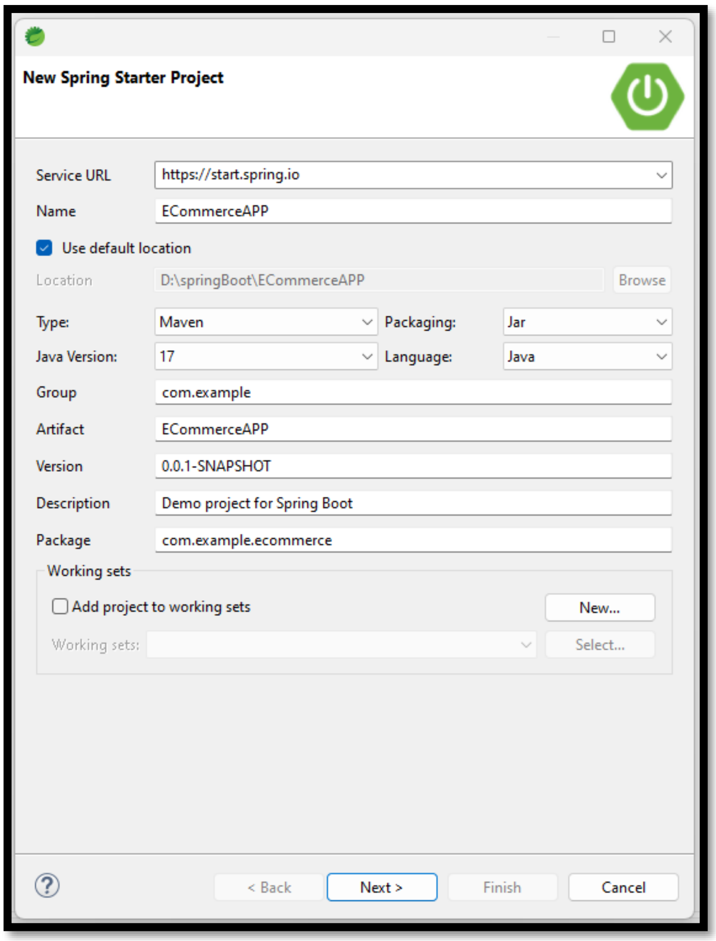
Here we are creating a REST API which allow users to Generate and Authenticate using JWT Tokens

1. Create SQL Table – (here we used MySql as Database)



```sql
create database StoreDB;
use StoreDB;

create table Users
(
    id int primary key auto_increment,
    firstname varchar(50),
    lastname varchar(50),
    username varchar(50) unique,
    email varchar(100),
    password varchar(50),
    role varchar(20)
);
```

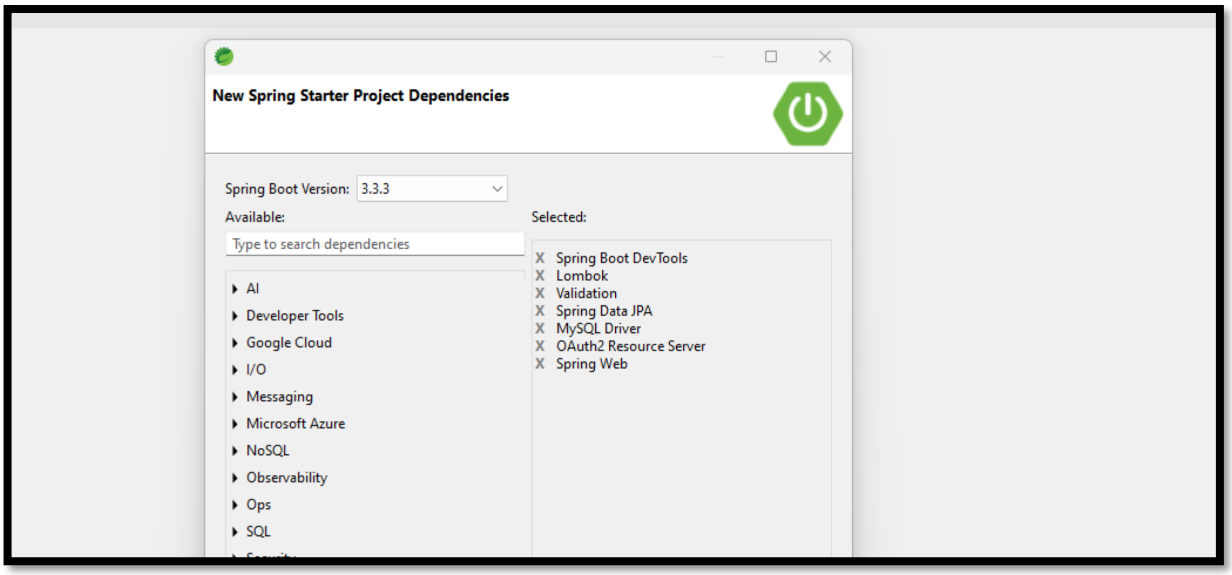
2. Create a Spring Boot Project in Spring Tool Suite



New Spring Starter Project

Service URL	https://start.spring.io
Name	ECommerceAPP
Use default location	✓
Location	D:\springBoot\ECommerceAPP
Type:	Maven
Packaging:	Jar
Java Version:	17
Language:	Java
Group	com.example
Artifact	ECommerceAPP
Version	0.0.1-SNAPSHOT
Description	Demo project for Spring Boot
Package	com.example.ecommerce
Add project to working sets	☐

3. Select the following dependencies


New Spring Starter Project Dependencies

Spring Boot Version: 3.3.3

Available:

Type to search dependencies

- AI
- Developer Tools
- Google Cloud
- I/O
- Messaging
- Microsoft Azure
- NoSQL
- Observability
- Ops
- SQL

Selected:

X Spring Boot DevTools
X Lombok
X Validation
X Spring Data JPA
X MySQL Driver
X OAuth2 Resource Server
X Spring Web

1. Spring Boot Dev Tools

- **Purpose**: This dependency provides various tools to enhance the development experience with Spring Boot applications. It includes features like automatic restarts, live reload, and configurations for development-time enhancements. It helps developers see changes instantly without needing to restart the application manually.

2. Lombok

- **Purpose**: Lombok is a library that helps reduce boilerplate code in Java. It uses annotations to generate common methods such as getters, setters, toString(), equals(), and hashCode() at compile time. This makes the code cleaner and easier to read. For example, by using the @Data annotation, you can automatically generate these methods for a class.

3. Validation

- **Purpose**: This dependency provides support for validating Java beans. It allows you to apply validation constraints (like @NotNull, @Size, etc.) to your model classes. This is useful for ensuring that incoming data meets specific criteria before processing it, which can prevent errors and improve application robustness.

4. Spring Data JPA

- **Purpose**: This dependency simplifies data access and manipulation in Spring applications. It provides an abstraction layer over JPA (Java Persistence API) and allows developers to interact with relational databases using repositories. It enables CRUD (Create, Read, Update, Delete) operations with minimal boilerplate code and provides features like pagination and sorting.

5. MySQL Driver

- **Purpose**: This dependency is the JDBC (Java Database Connectivity) driver for MySQL databases. It allows your Spring Boot application to connect to and interact with a MySQL database. It is necessary for performing database operations, such as executing queries and managing transactions.

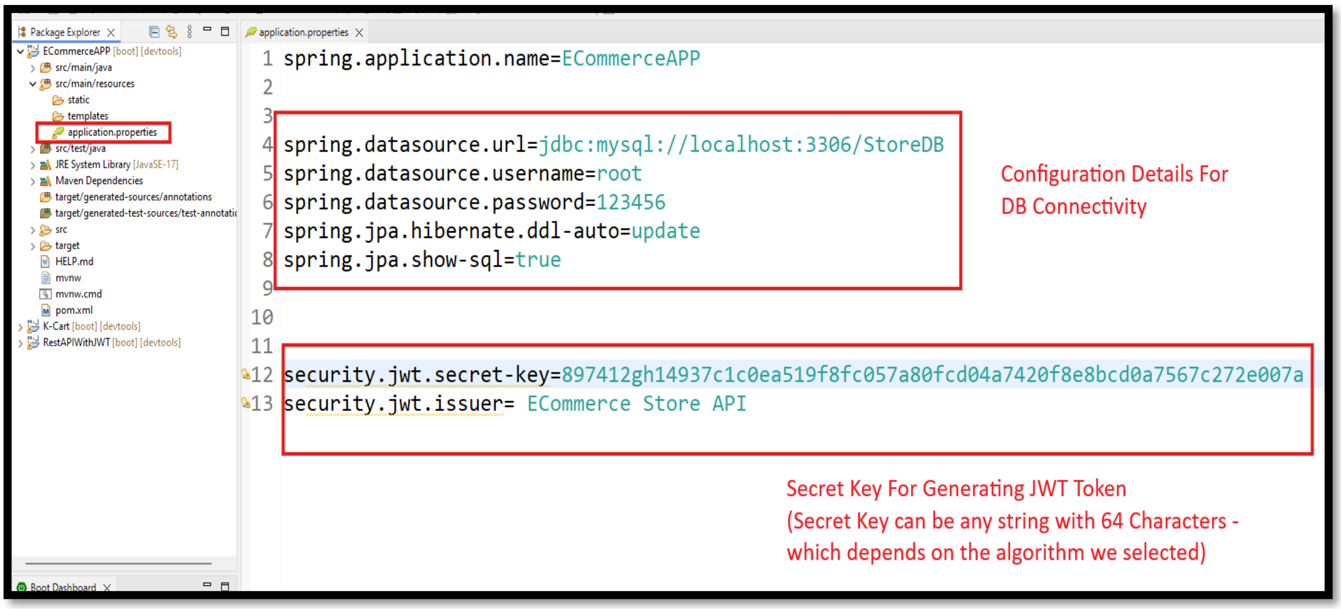
6. OAuth2 Resource Server

- **Purpose**: This dependency provides support for securing REST APIs with OAuth2. It allows your application to validate incoming JWT tokens and ensure that only authorized users can access specific endpoints. This is crucial for implementing secure authentication and authorization mechanisms in applications that require user identity verification.

7. Spring Web

- **Purpose**: This dependency includes all the necessary components for building web applications with Spring, such as RESTful APIs. It provides support for MVC (Model-View-Controller) architecture, including controllers, view resolvers, and RESTful services. This is essential for creating web applications and exposing APIs that clients can consume.

4. In Application Properties configure the necessary details

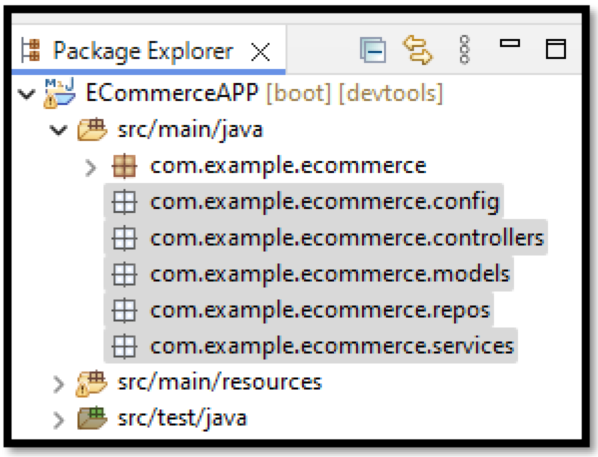


```
spring.application.name=ECommerceAPP


spring.datasource.url=jdbc:mysql://localhost:3306/StoreDB
spring.datasource.username=root
spring.datasource.password=123456
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true



security.jwt.secret-key=897412gh14937c1c0ea519f8fc057a80fcd04a7420f8e8bcd0a7567c272e007a
security.jwt.issuer= ECommerce Store API
```

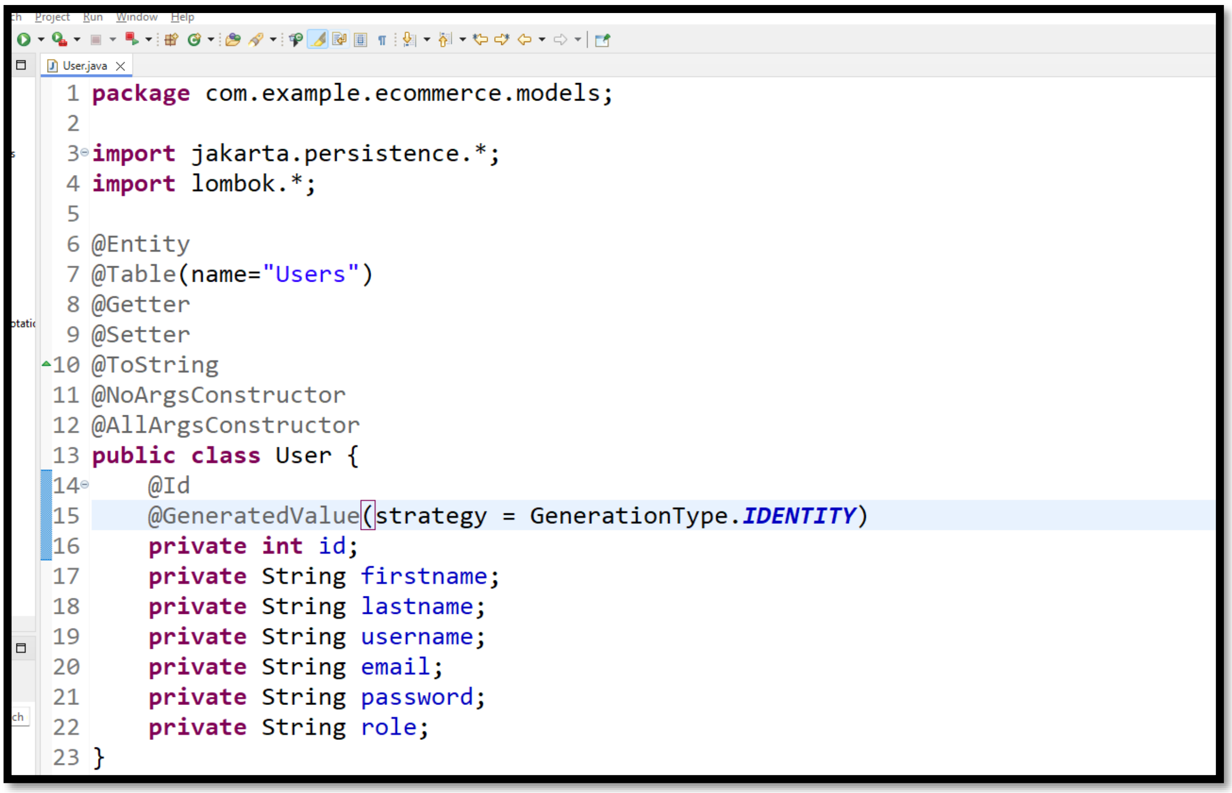
Configuration Details For DB Connectivity

Secret Key For Generating JWT Token (Secret Key can be any string with 64 Characters - which depends on the algorithm we selected)

5. Create necessary packages

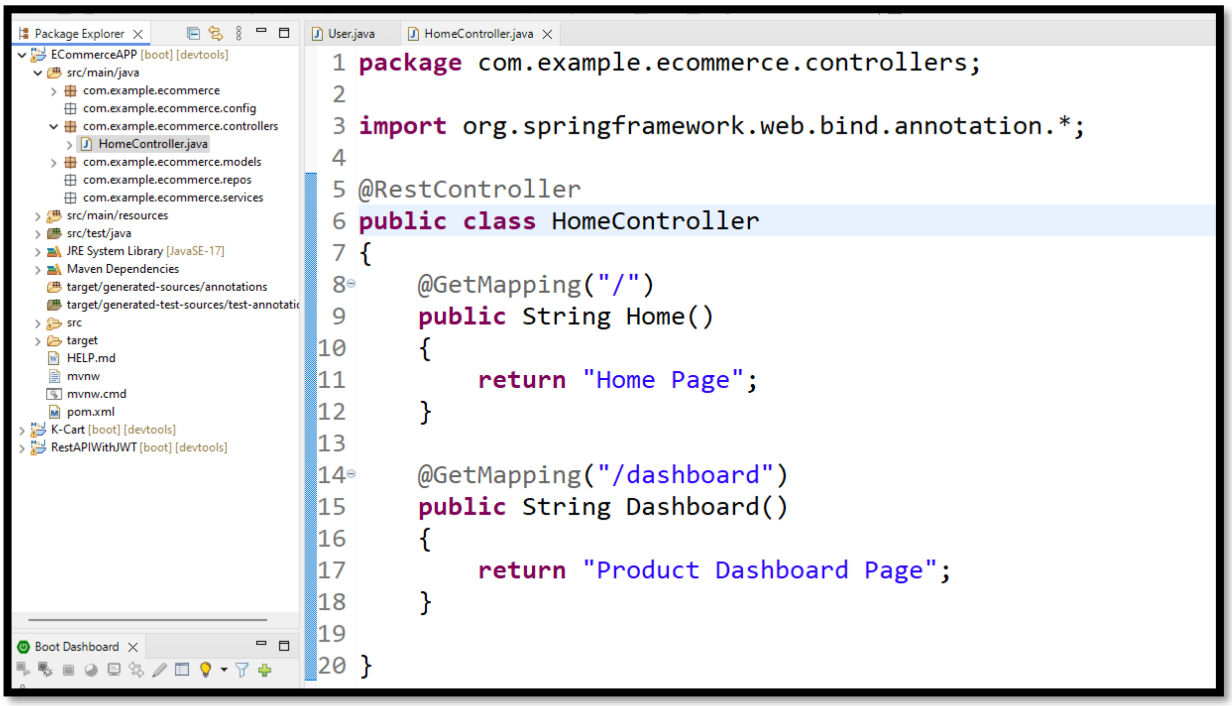

Package Explorer
ECommerceAPP [boot] [devtools]
src/main/java
com.example.ecommerce
com.example.ecommerce.config
com.example.ecommerce.controllers
com.example.ecommerce.models
com.example.ecommerce.repos
com.example.ecommerce.services
src/main/resources
src/test/java

6. In com.example.ecommerce.models package create a class "User.java"



```java
package com.example.ecommerce.models;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Table(name="Users")
@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String firstname;
    private String lastname;
    private String username;
    private String email;
    private String password;
    private String role;
}
```

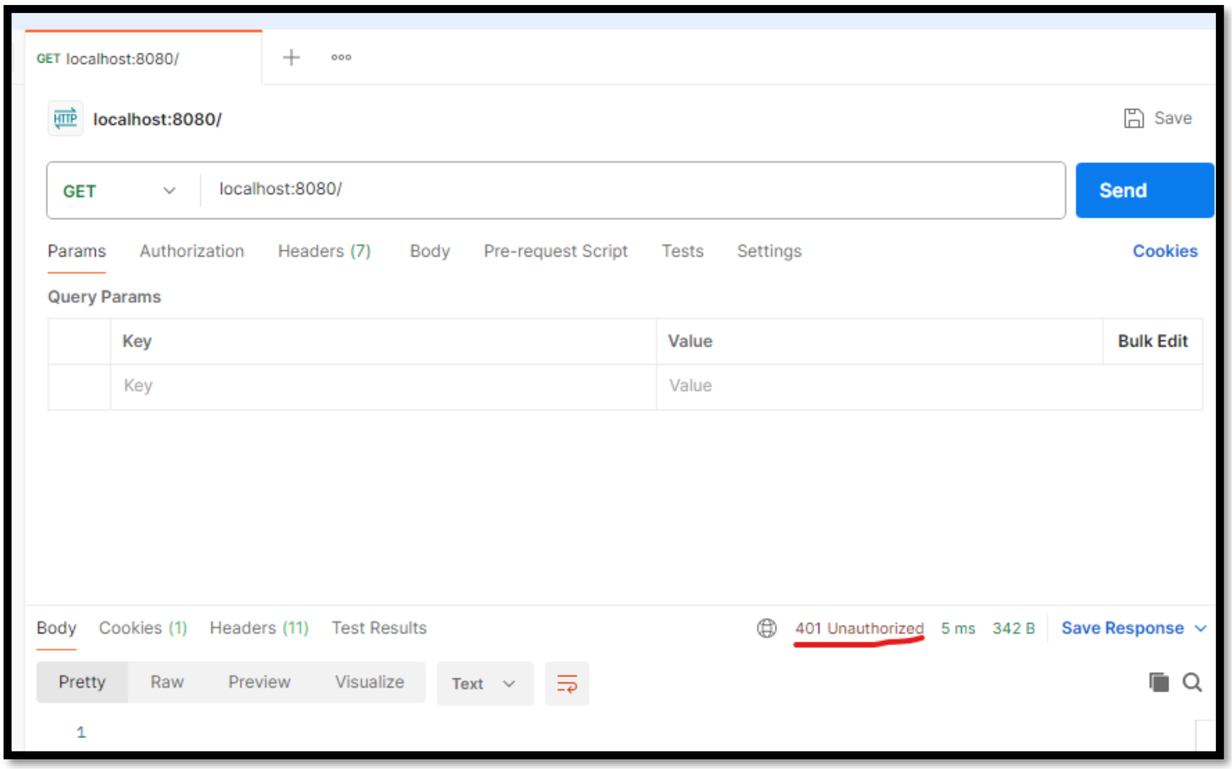
7. In com.example.ecommerce.controllers package create a class "HomeController.java"


Package Explorer
EcommerceAPP [boot] [devtools]
src/main/java
com.example.ecommerce
com.example.ecommerce.config
com.example.ecommerce.controllers
HomeController.java
com.example.ecommerce.models
com.example.ecommerce.repos
com.example.ecommerce.services
src/main/resources
src/test/java
JRE System Library [JavaSE-17]
Maven Dependencies
target/generated-sources/annotations
target/generated-test-sources/test-annotatic
src
target
HELP.md
mvnw
mvnw.cmd
pom.xml
K-Cart [boot] [devtools]
RestAPIWithJWT [boot] [devtools]
Boot Dashboard
User.java
HomeController.java

```java
package com.example.ecommerce.controllers;

import org.springframework.web.bind.annotation.*;

@RestController
public class HomeController
{
    @GetMapping("/")
    public String Home()
    {
        return "Home Page";
    }

    @GetMapping("/dashboard")
    public String Dashboard()
    {
        return "Product Dashboard Page";
    }
}
```

Here I can created some sample endpoints like '/' and '/dashboard'. Now I am running the application and test them using postman whether they are working or not

8. Test the enpoints in postman after running the app



GET localhost:8080/

localhost:8080/

GET	localhost:8080/

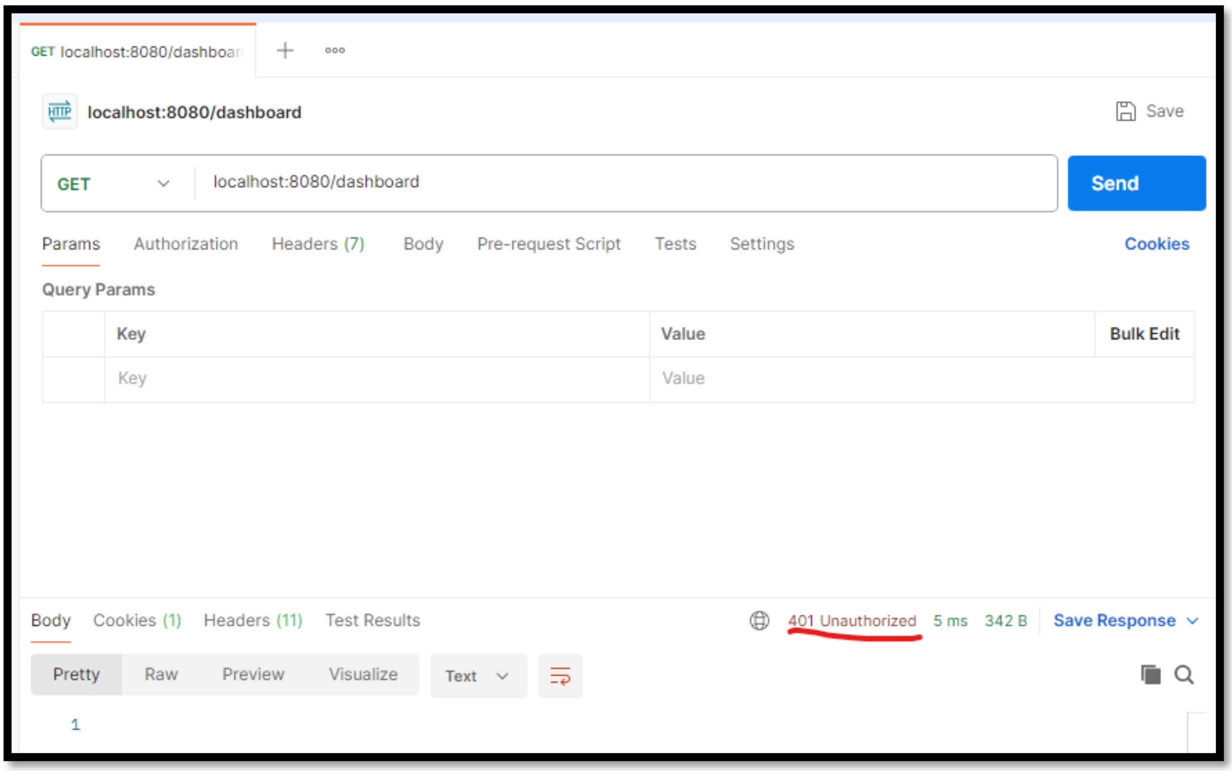
Params | Authorization | Headers (7) | Body | Pre-request Script | Tests | Settings | Cookies

Query Params

Key	Value	Bulk Edit
Key	Value	

Body | Cookies (1) | Headers (11) | Test Results — 401 Unauthorized 5 ms 342 B

```
1
```



GET localhost:8080/dashboar

localhost:8080/dashboard

GET	localhost:8080/dashboard

Params | Authorization | Headers (7) | Body | Pre-request Script | Tests | Settings | Cookies

Query Params

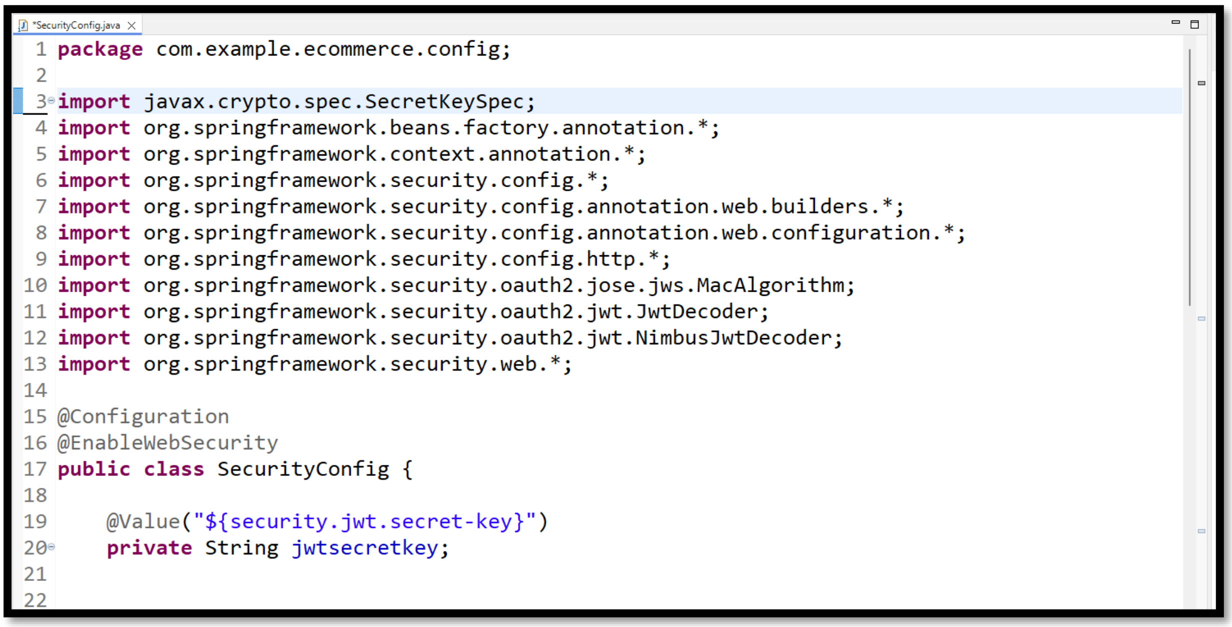
Key	Value	Bulk Edit
Key	Value	

Body | Cookies (1) | Headers (11) | Test Results — 401 Unauthorized 5 ms 342 B

```
1
```

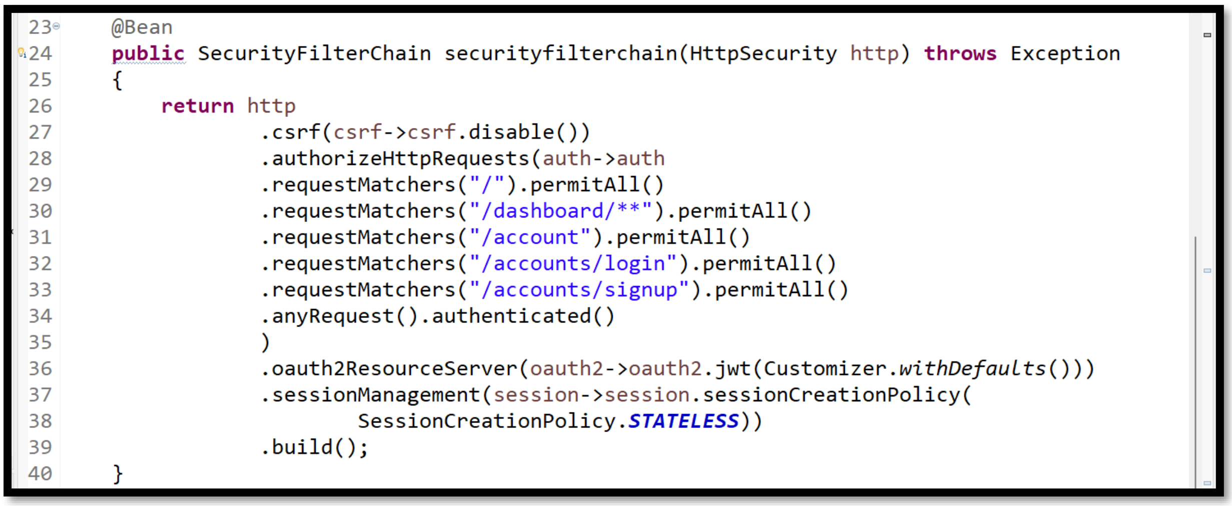
9. To Solve this 401 status code, we have to allow the specific url's from authorization filter

For that we are creating a file "SecurityConfig.java" inside the package com.example.ecommerce.config

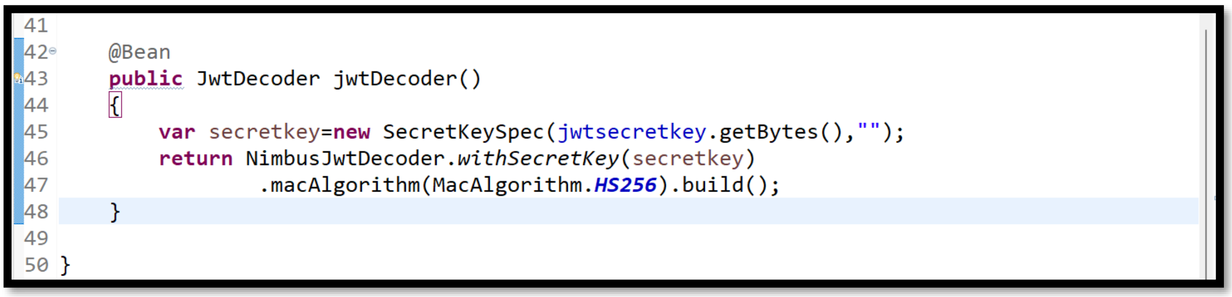


*SecurityConfig.java

```java
package com.example.ecommerce.config;

import javax.crypto.spec.SecretKeySpec;
import org.springframework.beans.factory.annotation.*;
import org.springframework.context.annotation.*;
import org.springframework.security.config.*;
import org.springframework.security.config.annotation.web.builders.*;
import org.springframework.security.config.annotation.web.configuration.*;
import org.springframework.security.config.http.*;
import org.springframework.security.oauth2.jose.jws.MacAlgorithm;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.NimbusJwtDecoder;
import org.springframework.security.web.*;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Value("${security.jwt.secret-key}")
    private String jwtsecretkey;


```



```java
    @Bean
    public SecurityFilterChain securityfilterchain(HttpSecurity http) throws Exception
    {
        return http
                .csrf(csrf->csrf.disable())
                .authorizeHttpRequests(auth->auth
                .requestMatchers("/").permitAll()
                .requestMatchers("/dashboard/**").permitAll()
                .requestMatchers("/account").permitAll()
                .requestMatchers("/accounts/login").permitAll()
                .requestMatchers("/accounts/signup").permitAll()
                .anyRequest().authenticated()
                )
                .oauth2ResourceServer(oauth2->oauth2.jwt(Customizer.withDefaults()))
                .sessionManagement(session->session.sessionCreationPolicy(
                        SessionCreationPolicy.STATELESS))
                .build();
    }
```



```java

    @Bean
    public JwtDecoder jwtDecoder()
    {
        var secretkey=new SecretKeySpec(jwtsecretkey.getBytes(),"");
        return NimbusJwtDecoder.withSecretKey(secretkey)
                .macAlgorithm(MacAlgorithm.HS256).build();
    }

}
```

Now This SecurityConfig permits the mentioned routes like "/","/dashboard" etc..

You can get the details of this class mentioned below

1. Class-Level Annotations

- **@Configuration**:
  - Indicates that this class is a Spring configuration class. It allows Spring to recognize this class as a source of bean definitions.
- **@EnableWebSecurity**:
  - Enables Spring Security's web security support. This annotation is used to configure security settings for web applications.

2. **Private Field: jwtsecretkey**

```
@Value("${security.jwt.secret-key}")

private String jwtsecretkey;
```

@Value("${security.jwt.secret-key}"):

- Injects the value of security.jwt.secret-key from the application's properties file into the jwtsecretkey variable. This key will be used to sign and verify JWT tokens.

3. **Security Filter Chain: securityfilterchain Method**

@Bean:

- Marks this method as a bean provider, meaning the returned SecurityFilterChain will be managed by the Spring container.

SecurityFilterChain:

- The SecurityFilterChain defines the security configurations applied to incoming HTTP requests. This is where you configure how requests should be secured.

HttpSecurity:

- A configuration object that allows you to specify how web-based security is configured for specific HTTP requests.

csrf(csrf -> csrf.disable()):

- Disables Cross-Site Request Forgery (CSRF) protection. This is common in stateless REST APIs since they don't maintain a user session.

authorizeHttpRequests(auth -> auth...):

- Defines the authorization rules for incoming HTTP requests.
- **requestMatchers**: Specifies which paths are permitted without authentication.
  - The permitAll() method allows any user to access these endpoints.
  - **anyRequest().authenticated()**: Requires authentication for any other requests not explicitly allowed.

oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults())):

- Configures the application as an OAuth2 resource server that uses JWT for authentication.
- The Customizer.withDefaults() sets up the JWT decoder with default settings.

sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)):

- Configures the session management policy.
- **SessionCreationPolicy.STATELESS**: Indicates that the application will not maintain a session for users. Every request must be authenticated independently, suitable for stateless REST APIs.

build():

- Finalizes the security filter chain configuration and returns the SecurityFilterChain bean.

4. **JWT Decoder: jwtDecoder Method**

@Bean:

- Indicates that this method returns a bean to be managed by the Spring container.

JwtDecoder:

- The JwtDecoder is responsible for decoding and validating the JWT tokens sent by clients.

SecretKeySpec:

- Converts the raw jwtsecretkey string (which is retrieved from application properties) into a SecretKeySpec object. This key is used for verifying the signature of JWT tokens.

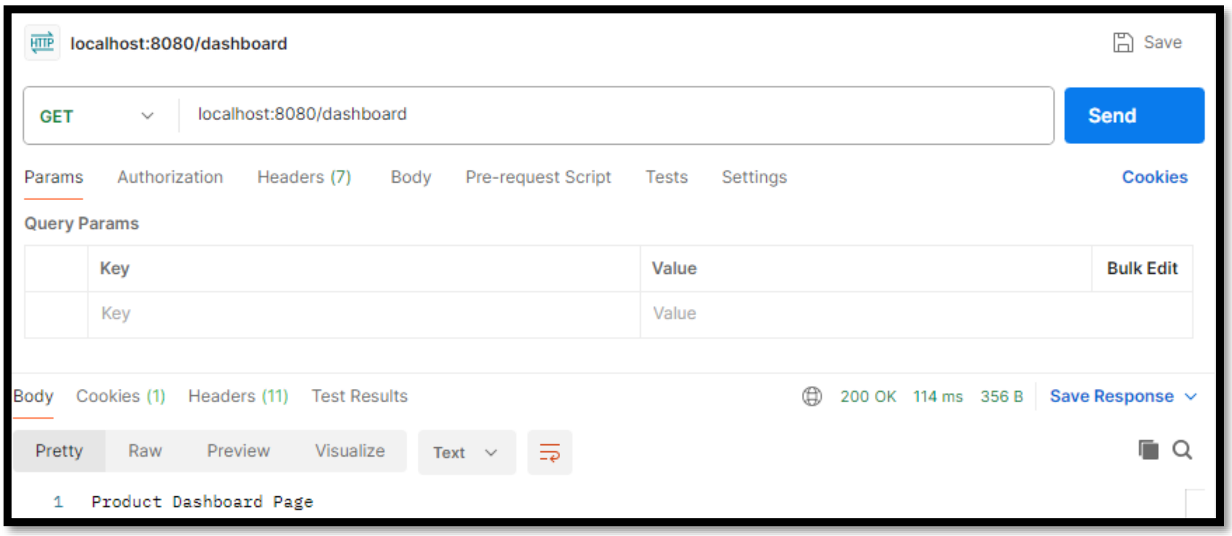
NimbusJwtDecoder.withSecretKey(secretkey):

- Creates an instance of NimbusJwtDecoder using the provided secret key for token decoding.

macAlgorithm(MacAlgorithm.HS256):

- Specifies the algorithm used to sign the JWT token. In this case, it's HS256, which is a HMAC-based algorithm.

10. Test the end points again with Postman now:

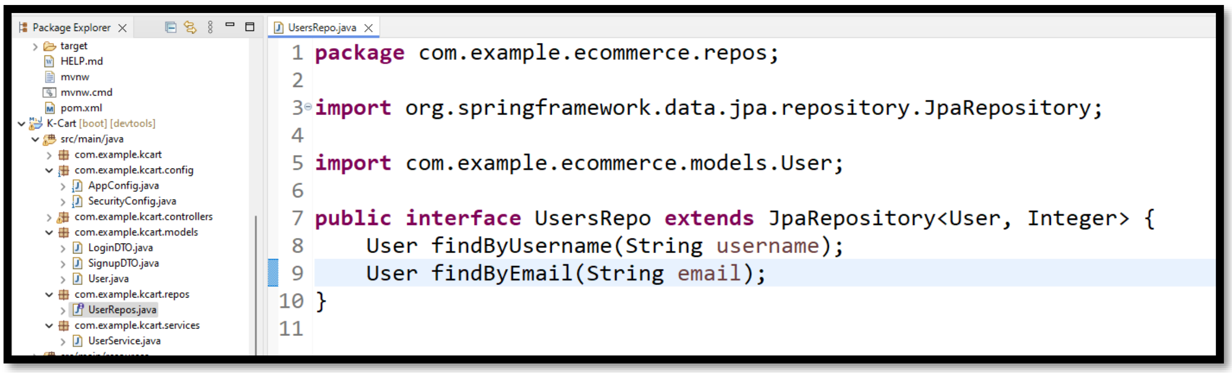


```
GET localhost:8080/dashboard
```

```
1  Product Dashboard Page
```

Its working and All End-Points are accessible. Now we are implementing Signup and Login functionalities along with JWT Token generation

11. Create an interface “UsersRepo” in com.example.ecommerce.repos



```java
package com.example.ecommerce.repos;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.ecommerce.models.User;

public interface UsersRepo extends JpaRepository<User, Integer> {
    User findByUsername(String username);
    User findByEmail(String email);
}

```

Note: Here the methods we created are case-sensitive

org.springframework.data.jpa.repository.JpaRepository:

- This import brings in the JpaRepository interface from Spring Data JPA, which provides built-in methods for database operations such as saving, finding, deleting, and updating entities.

com.example.ecommerce.models.User:

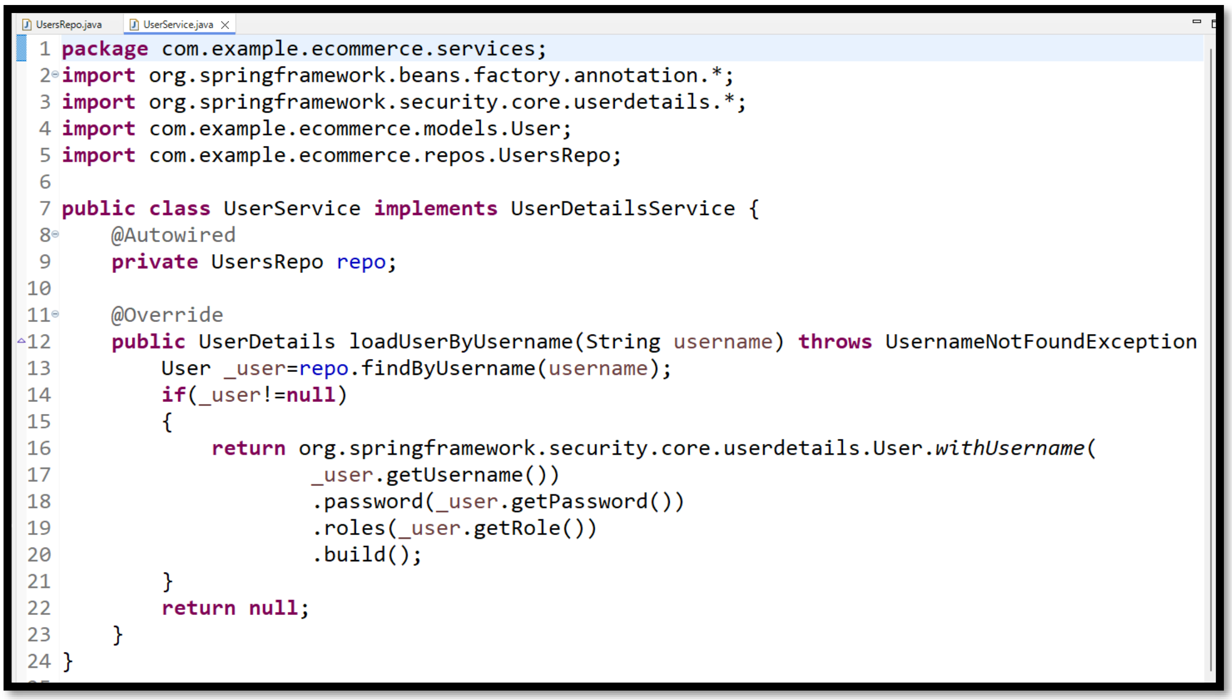
- This import brings in the User class from the models package. The User class represents the entity that will be managed by this repository.

- **public interface UsersRepo**:
    - This line defines the UsersRepo interface. Being an interface, it doesn't implement methods itself but instead extends another interface that provides implementation.
- **extends JpaRepository<User, Integer>**:
    - The UsersRepo interface extends JpaRepository, which is a JPA-specific extension of the CrudRepository interface. By extending JpaRepository, UsersRepo inherits several methods to perform CRUD operations.
    - **<User, Integer>**: This specifies the type of the entity and the type of the entity's primary key:
        - **User**: The entity type that this repository manages.
        - **Integer**: The data type of the primary key (id) in the User entity.

Custom Query Methods

- **User findByUsername(String username);**:
    - This method defines a query that finds a User entity based on the username field. Spring Data JPA automatically generates the necessary query behind the scenes.
    - **findByUsername**: The method name follows a specific convention (findBy) that Spring Data JPA recognizes and uses to create the query.
    - **String username**: The method takes a username as a parameter and returns the corresponding User entity if found.
- **User findByEmail(String email);**:
    - Similar to findByUsername, this method defines a query that finds a User entity based on the email field.
    - **String email**: The method takes an email as a parameter and returns the corresponding User entity if found.

12. Create “UserService.java” in the package com.example.ecommerce.services



```java
package com.example.ecommerce.services;
import org.springframework.beans.factory.annotation.*;
import org.springframework.security.core.userdetails.*;
import com.example.ecommerce.models.User;
import com.example.ecommerce.repos.UsersRepo;

public class UserService implements UserDetailsService {
    @Autowired
    private UsersRepo repo;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
        User _user=repo.findByUsername(username);
        if(_user!=null)
        {
            return org.springframework.security.core.userdetails.User.withUsername(
                    _user.getUsername())
                    .password(_user.getPassword())
                    .roles(_user.getRole())
                    .build();
        }
        return null;
    }
}
```

org.springframework.beans.factory.annotation.Autowired:

- This annotation is used for dependency injection. It automatically injects the UsersRepo bean into the UserService class.

org.springframework.security.core.userdetails.UserDetailsService:

- This is a core interface in Spring Security that provides a way to retrieve user-related data. It has a single method loadUserByUsername that Spring Security uses to load user-specific data during the authentication process.

org.springframework.security.core.userdetails.User:

- This class is a built-in implementation of the UserDetails interface. It represents a user in Spring Security, containing the necessary information (like username, password, and roles) for authentication and authorization.

com.example.ecommerce.models.User:

- The User class represents the user entity in your application, with fields like username, password, and role.

com.example.ecommerce.repos.UsersRepo:

- The UsersRepo is a repository interface used to interact with the User entity in the database. This is where user data is stored and retrieved.

Class Definition: UserService:

UserService:

- This class implements the UserDetailsService interface, which is required by Spring Security to fetch user details based on the username.

implements UserDetailsService:

- By implementing this interface, UserService must provide an implementation for the loadUserByUsername method, which Spring Security calls during the authentication process.

Dependency Injection: UsersRepo

@Autowired private

UsersRepo repo;

@Autowired:

- This annotation injects an instance of UsersRepo into UserService. The UsersRepo is used to retrieve user data from the database.

private UsersRepo repo;:

- Declares a private field repo to hold the injected repository.

Method: loadUserByUsername

@Override:

- Indicates that this method is overriding the loadUserByUsername method from the UserDetailsService interface.

UserDetails loadUserByUsername(String username):

- This method loads the user from the database based on the provided username. If the user is found, it returns a UserDetails object that Spring Security uses for authentication.

User _user = repo.findByUsername(username);:

- This line queries the UsersRepo to find a User entity by its username. The result is stored in the _user variable.

if (_user != null):

- Checks if the user was found in the database.

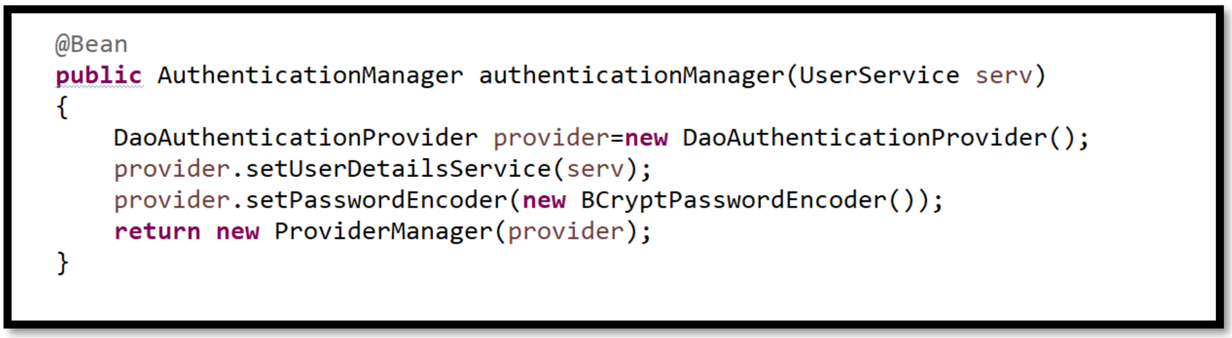
Returning UserDetails:

- If the user is found, a UserDetails object is created using the User.withUsername builder method.
- **_user.getUsername()**: Sets the username.
- **_user.getPassword()**: Sets the password.
- **_user.getRole()**: Sets the user roles. These roles determine the user's authorities (permissions).
- **.build()**: Constructs the UserDetails object.

return null;:

- If the user is not found, the method returns null. In a production environment, it is common to throw a UsernameNotFoundException instead of returning null to indicate that the user was not found.

13. Update “SecurityConfig.java” inside the package com.example.kcart.config



```
@Bean
public AuthenticationManager authenticationManager(UserService serv)
{
    DaoAuthenticationProvider provider=new DaoAuthenticationProvider();
    provider.setUserDetailsService(serv);
    provider.setPasswordEncoder(new BCryptPasswordEncoder());
    return new ProviderManager(provider);
}
```

Bean Definition:

@Bean:

- This annotation indicates that the method returns a Spring bean, which will be managed by the Spring container. The bean in this case is an AuthenticationManager.

AuthenticationManager authenticationManager(UserService serv):

- The method is named authenticationManager and returns an AuthenticationManager object.
- **UserService serv**: This is a parameter to the method. Spring will automatically inject an instance of UserService (which implements UserDetailsService) into this method. The UserService is used to load user details for authentication.

DaoAuthenticationProvider Setup:

DaoAuthenticationProvider provider:

- **This line creates a new instance of DaoAuthenticationProvider, which is a specific implementation of AuthenticationProvider that uses a UserDetailsService to retrieve user details for authentication.**
- **The DaoAuthenticationProvider is responsible for verifying the user's credentials (username and password) during authentication.**

Setting the UserDetailsService:

provider.setUserDetailsService(serv);:

- This line sets the UserDetailsService that the DaoAuthenticationProvider will use to load user details. Here, serv is the UserService instance passed to the method. This service is responsible for fetching the user details from the database.

Password Encoder Configuration:

provider.setPasswordEncoder(new BCryptPasswordEncoder());:

- This line configures the DaoAuthenticationProvider to use BCryptPasswordEncoder for encoding and matching passwords.
- **BCryptPasswordEncoder:**
    - This is a password encoder that uses the BCrypt hashing function to securely store passwords. During authentication, the raw password entered by the user is encoded and compared with the stored (hashed) password.

Returning AuthenticationManager:

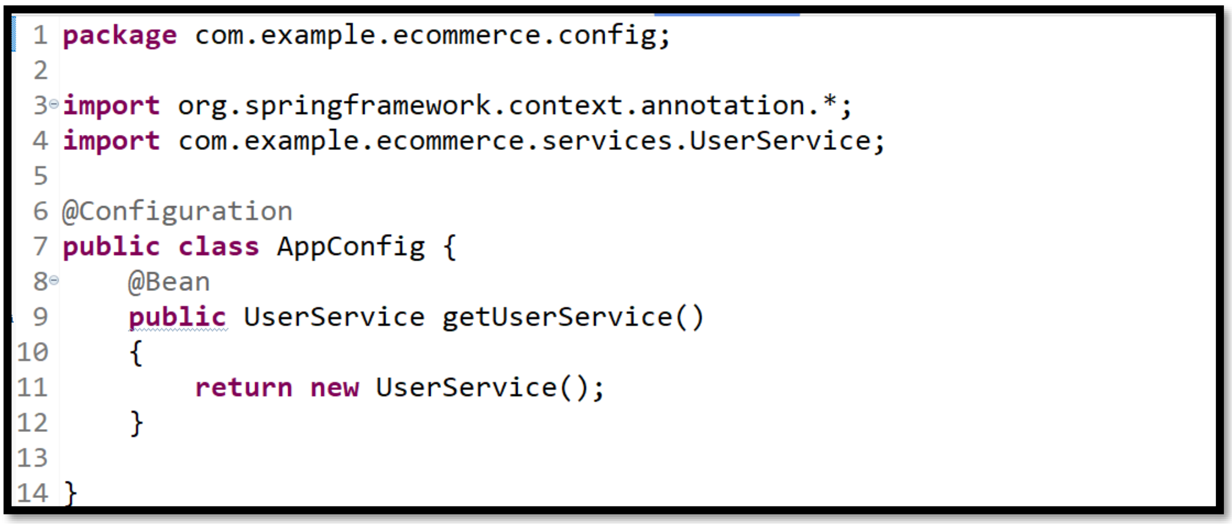
new ProviderManager(provider);:

- This line creates a new ProviderManager, which is an implementation of AuthenticationManager. The ProviderManager takes one or more AuthenticationProvider instances (in this case, DaoAuthenticationProvider) and delegates the authentication process to them.

return new ProviderManager(provider);:

- Finally, the method returns the configured AuthenticationManager bean. This AuthenticationManager will be used by Spring Security to handle authentication requests.

14. Create "**AppConfig.java**" inside the package com.example.ecommerce.config



```java
package com.example.ecommerce.config;

import org.springframework.context.annotation.*;
import com.example.ecommerce.services.UserService;

@Configuration
public class AppConfig {
    @Bean
    public UserService getUserService()
    {
        return new UserService();
    }

}
```

@Configuration:

- This annotation marks the class as a source of bean definitions. Spring will automatically scan this class for methods annotated with @Bean and register them as beans in the application context.

- Essentially, @Configuration tells Spring that this class contains configuration details and bean definitions that should be managed by the Spring container.

@Bean:

- This annotation is used to indicate that the method returns a bean to be managed by the Spring container. The method name (getUserService) becomes the name of the bean in the application context, although this can be customized.
- The UserService bean created here can then be injected into other components of the application where needed.

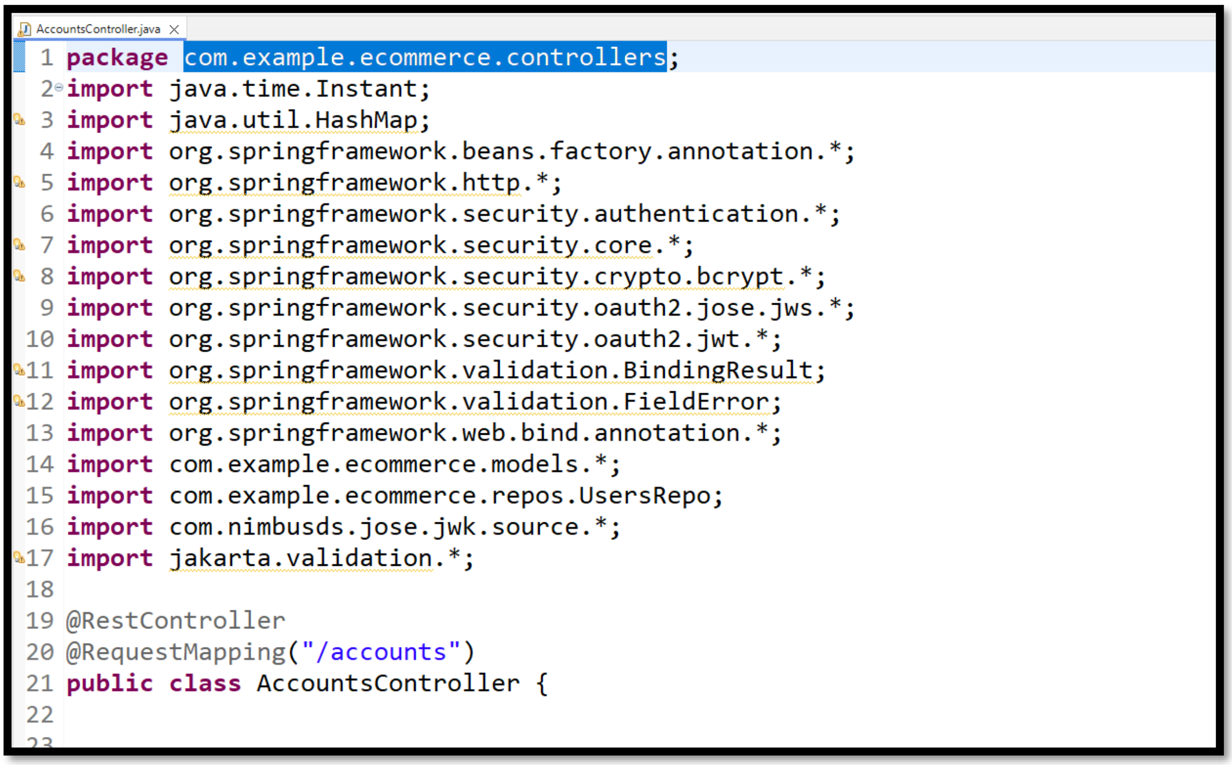
public UserService getUserService():

- This method returns a new instance of the UserService class. When Spring processes this configuration class, it will call this method and register the returned UserService instance as a bean.

return new UserService();:

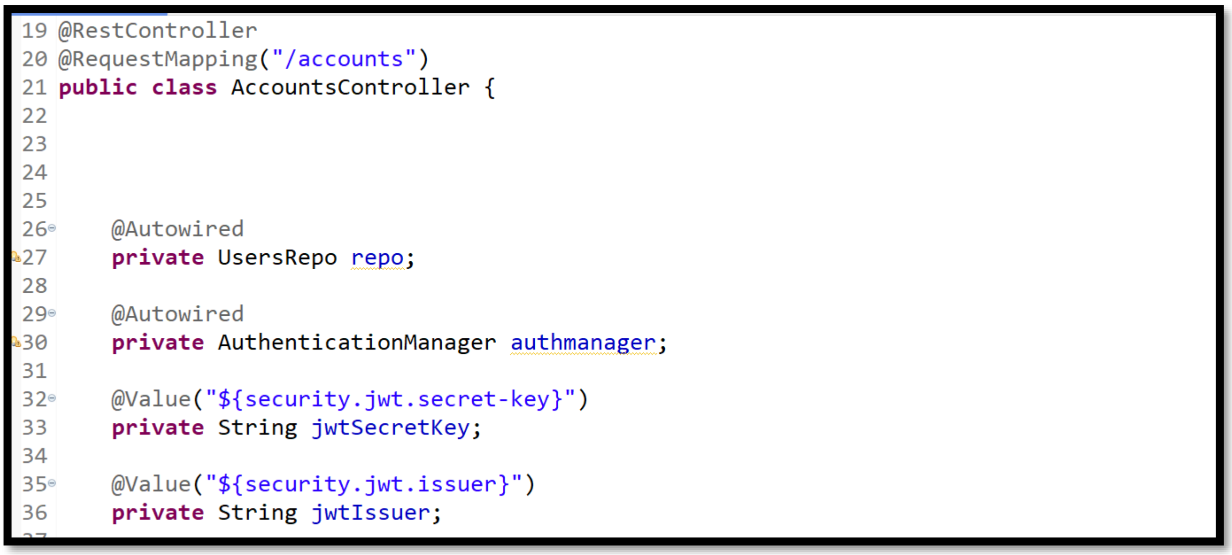
- A new instance of UserService is created and returned. This instance will be managed by the Spring container and can be injected wherever UserService is required.

15. Create "ActtountsController.java" in the package com.example.ecommerce.controllers

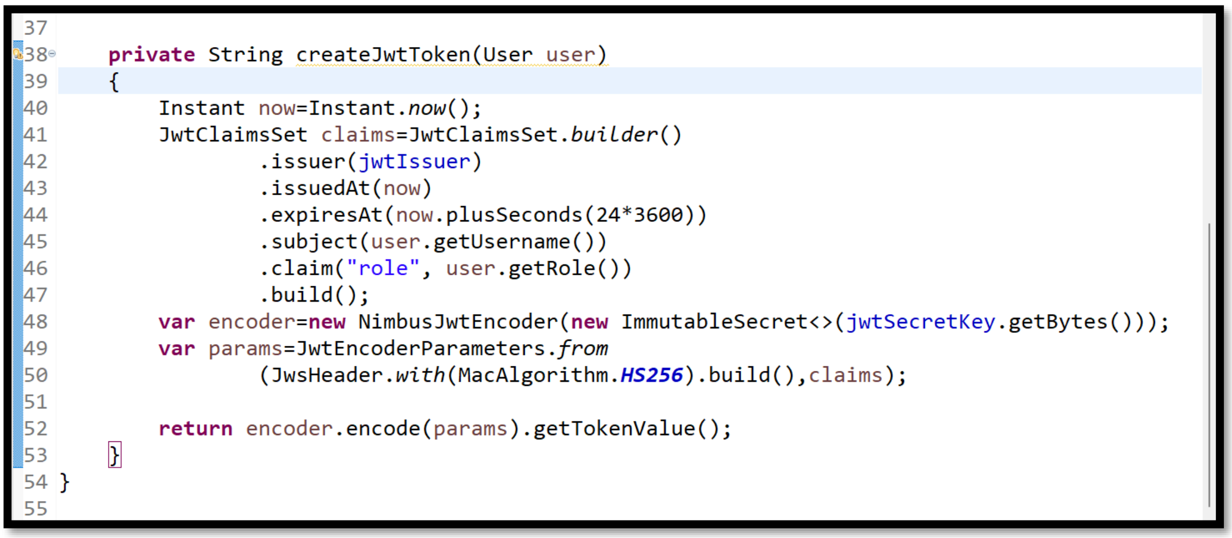


AccountsController.java

```java
package com.example.ecommerce.controllers;
import java.time.Instant;
import java.util.HashMap;
import org.springframework.beans.factory.annotation.*;
import org.springframework.http.*;
import org.springframework.security.authentication.*;
import org.springframework.security.core.*;
import org.springframework.security.crypto.bcrypt.*;
import org.springframework.security.oauth2.jose.jws.*;
import org.springframework.security.oauth2.jwt.*;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.annotation.*;
import com.example.ecommerce.models.*;
import com.example.ecommerce.repos.UsersRepo;
import com.nimbusds.jose.jwk.source.*;
import jakarta.validation.*;

@RestController
@RequestMapping("/accounts")
public class AccountsController {

```



```java
@RestController
@RequestMapping("/accounts")
public class AccountsController {




    @Autowired
    private UsersRepo repo;

    @Autowired
    private AuthenticationManager authmanager;

    @Value("${security.jwt.secret-key}")
    private String jwtSecretKey;

    @Value("${security.jwt.issuer}")
    private String jwtIssuer;
```

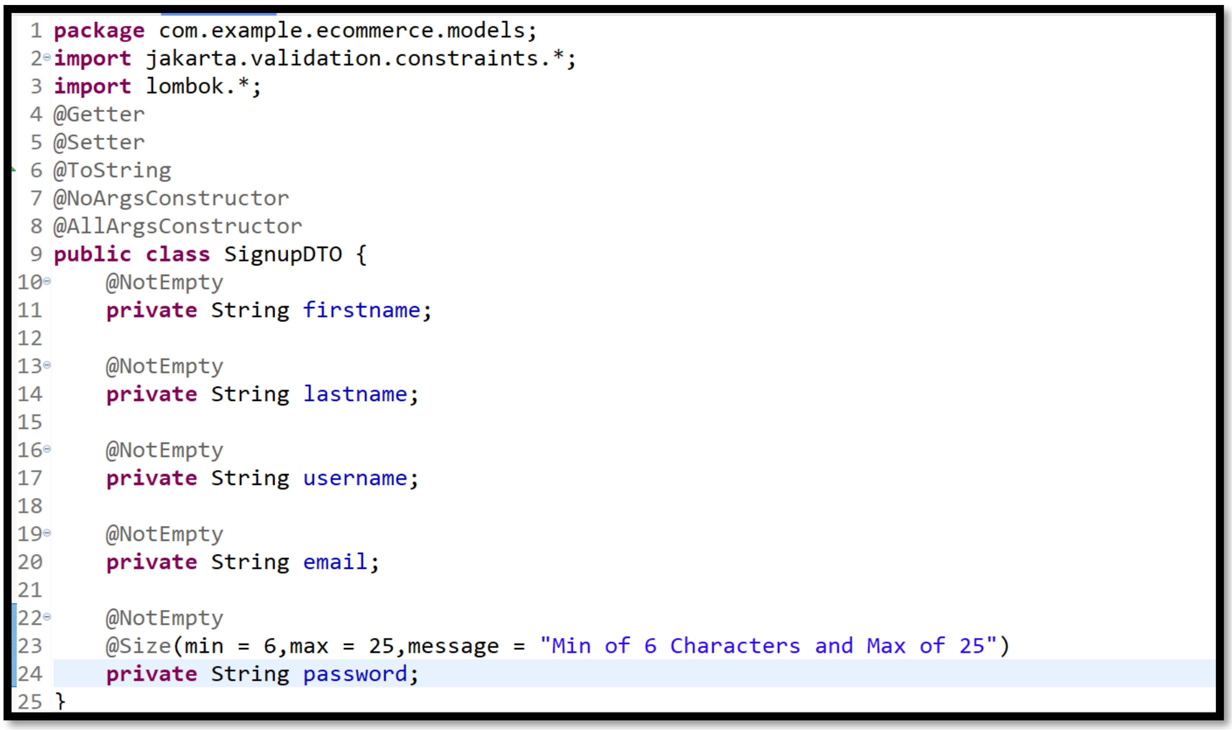


```java

    private String createJwtToken(User user)
    {
        Instant now=Instant.now();
        JwtClaimsSet claims=JwtClaimsSet.builder()
                .issuer(jwtIssuer)
                .issuedAt(now)
                .expiresAt(now.plusSeconds(24*3600))
                .subject(user.getUsername())
                .claim("role", user.getRole())
                .build();
        var encoder=new NimbusJwtEncoder(new ImmutableSecret<>(jwtSecretKey.getBytes()));
        var params=JwtEncoderParameters.from
                (JwsHeader.with(MacAlgorithm.HS256).build(),claims);

        return encoder.encode(params).getTokenValue();
    }
}

```

Here we have createdJwtToken method based on the User object details like firstname,lastname,username,password,email properties with validity for One Day

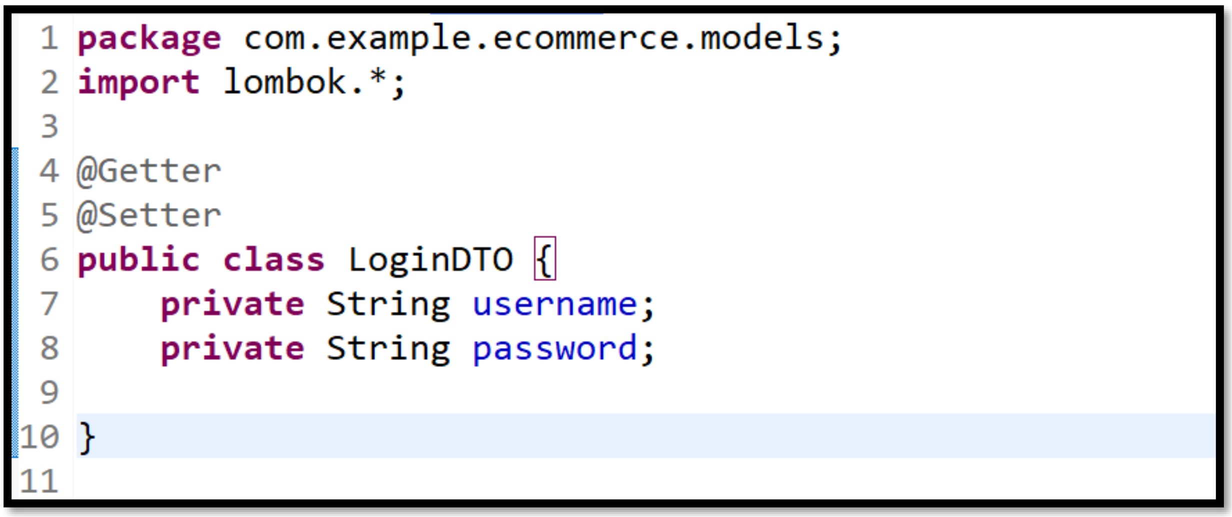
Now to implement endpoints like “Accounts/Signup”,”Accounts/Login”,”Accounts/Profile” we are creating 2 DTO models called SignupDTO.java and LoginDTO.java

16. Create “SignupDTO.java” inside the package com.example.ecommerce.models



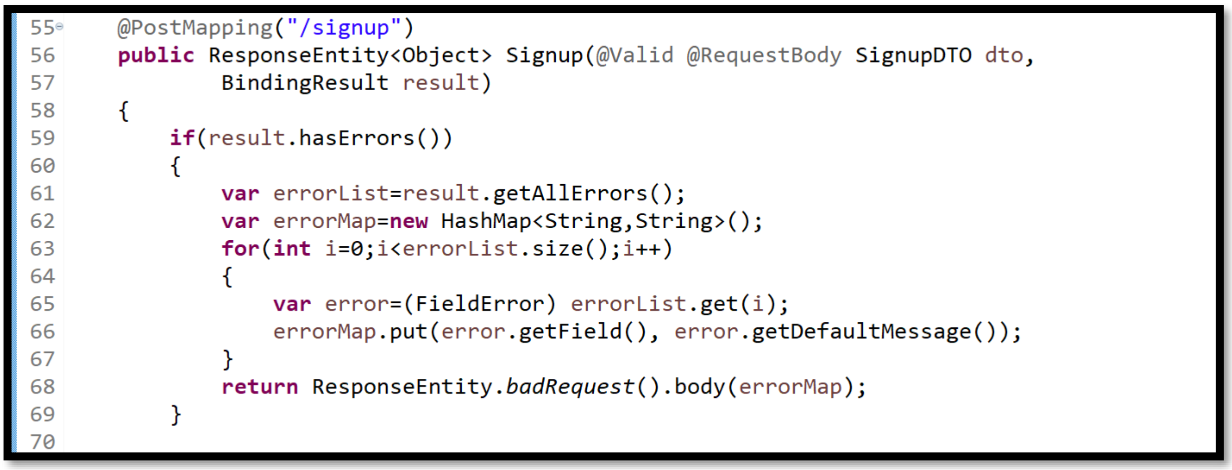
```java
package com.example.ecommerce.models;
import jakarta.validation.constraints.*;
import lombok.*;
@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class SignupDTO {
    @NotEmpty
    private String firstname;

    @NotEmpty
    private String lastname;

    @NotEmpty
    private String username;

    @NotEmpty
    private String email;

    @NotEmpty
    @Size(min = 6,max = 25,message = "Min of 6 Characters and Max of 25")
    private String password;
}
```

17. Create “LoginDTO.java” inside the package com.example.ecommerce.models

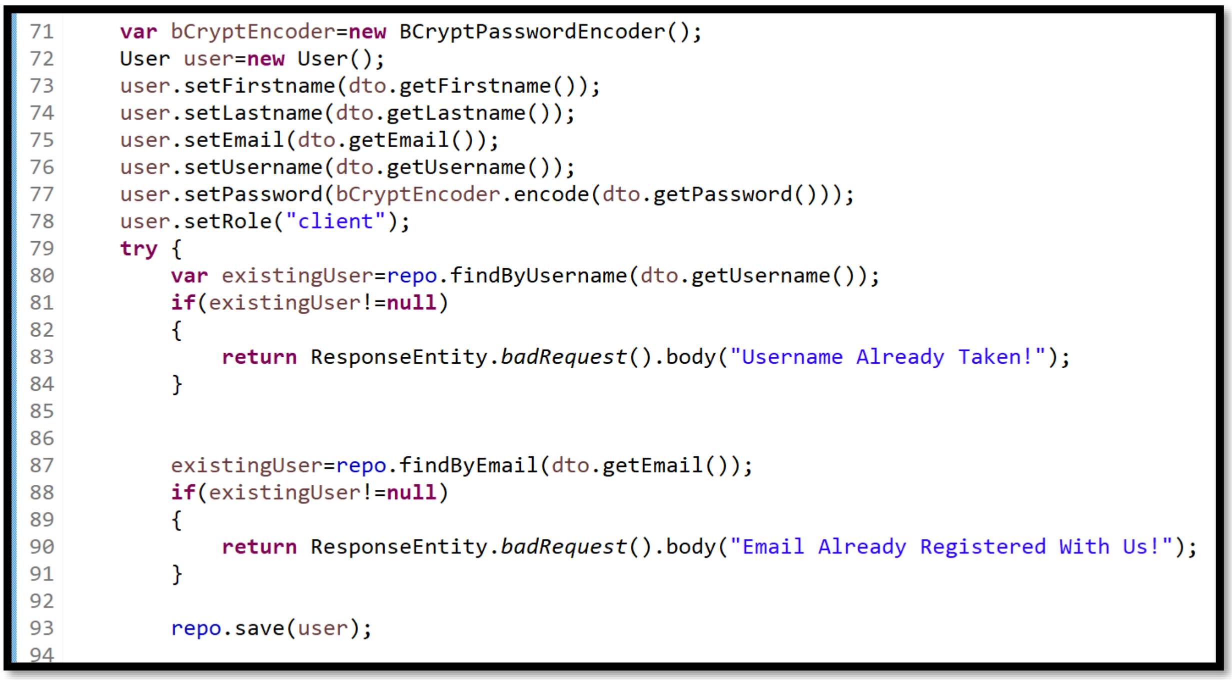


```java
package com.example.ecommerce.models;
import lombok.*;

@Getter
@Setter
public class LoginDTO {
    private String username;
    private String password;

}

```

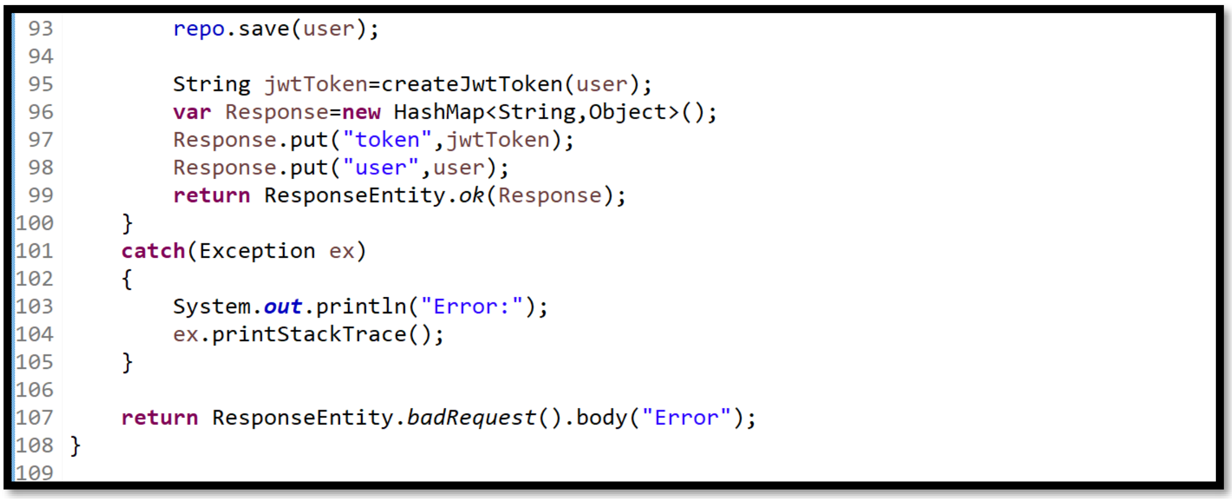
18. Inside "AccountsController.java" update the below endpoint method



```java
    @PostMapping("/signup")
    public ResponseEntity<Object> Signup(@Valid @RequestBody SignupDTO dto,
            BindingResult result)
    {
        if(result.hasErrors())
        {
            var errorList=result.getAllErrors();
            var errorMap=new HashMap<String,String>();
            for(int i=0;i<errorList.size();i++)
            {
                var error=(FieldError) errorList.get(i);
                errorMap.put(error.getField(), error.getDefaultMessage());
            }
            return ResponseEntity.badRequest().body(errorMap);
        }

```



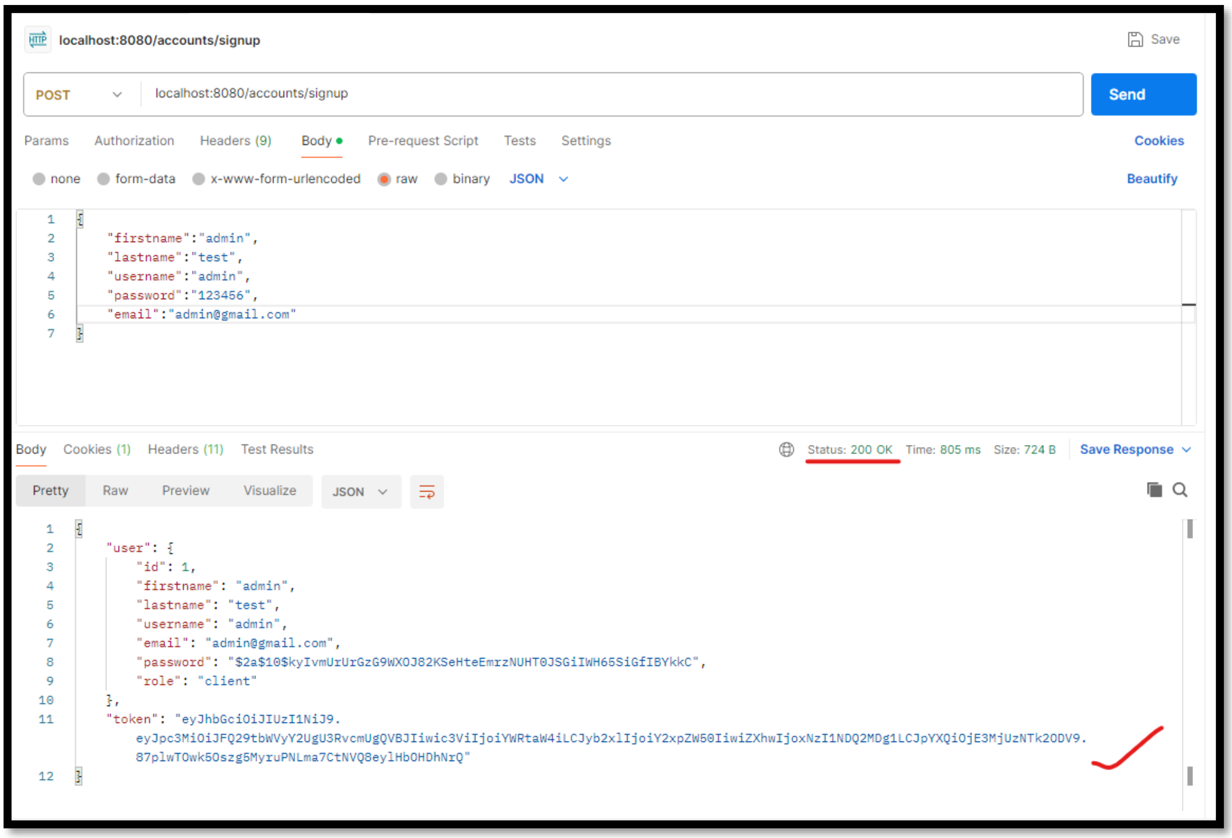
```java
        var bCryptEncoder=new BCryptPasswordEncoder();
        User user=new User();
        user.setFirstname(dto.getFirstname());
        user.setLastname(dto.getLastname());
        user.setEmail(dto.getEmail());
        user.setUsername(dto.getUsername());
        user.setPassword(bCryptEncoder.encode(dto.getPassword()));
        user.setRole("client");
        try {
            var existingUser=repo.findByUsername(dto.getUsername());
            if(existingUser!=null)
            {
                return ResponseEntity.badRequest().body("Username Already Taken!");
            }


            existingUser=repo.findByEmail(dto.getEmail());
            if(existingUser!=null)
            {
                return ResponseEntity.badRequest().body("Email Already Registered With Us!");
            }

            repo.save(user);

```



```java
            repo.save(user);

            String jwtToken=createJwtToken(user);
            var Response=new HashMap<String,Object>();
            Response.put("token",jwtToken);
            Response.put("user",user);
            return ResponseEntity.ok(Response);
        }
        catch(Exception ex)
        {
            System.out.println("Error:");
            ex.printStackTrace();
        }

        return ResponseEntity.badRequest().body("Error");
    }

```

Now Lets test signup endpoint and check whether jwt token was generating or not

19. Test signup endpoint in postman



POST localhost:8080/accounts/signup

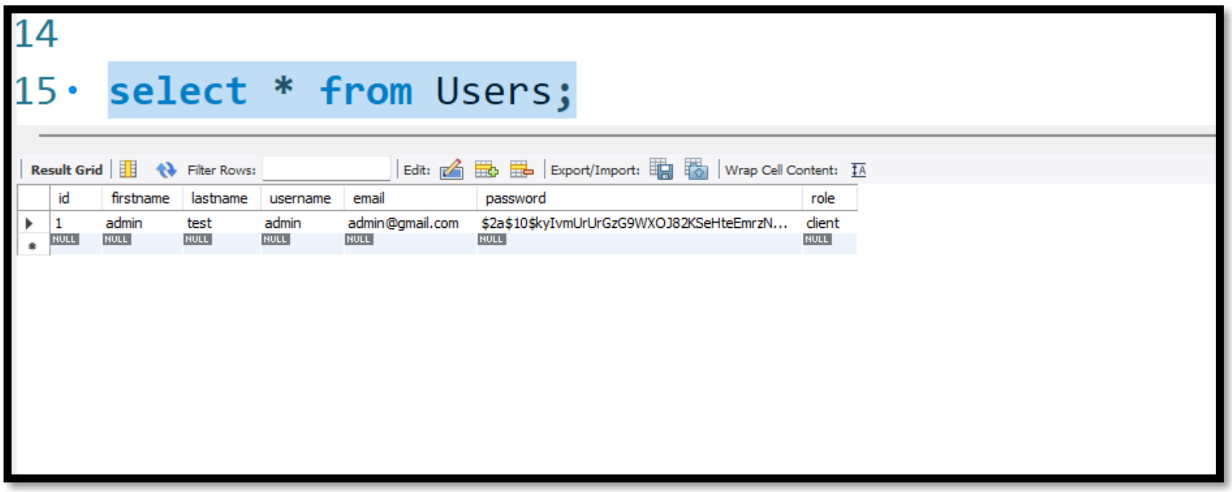
```
{
    "firstname":"admin",
    "lastname":"test",
    "username":"admin",
    "password":"123456",
    "email":"admin@gmail.com"
}
```

Status: 200 OK Time: 805 ms Size: 724 B

JSON

```
{
    "user": {
        "id": 1,
        "firstname": "admin",
        "lastname": "test",
        "username": "admin",
        "email": "admin@gmail.com",
        "password": "$2a$10$kyIvmUrUrGzG9WXOJ82KSeHteEmrzNUHT0JSGiIWH65SiGfIBYkkC",
        "role": "client"
    },
    "token": "eyJhbGciOiJIUzI1NiJ9.
        eyJpc3MiOiJFQ29tbWVyY2UgU3RvcmUgQVBJIiwic3ViIjoiYWRtaW4iLCJyb2xlIjoiY2xpZW50IiwiZXhwIjoxNzI1NDQ2MDg1LCJpYXQiOjE3MjUzNTk2ODV9.
        87plwTOwk5Oszg5MyruPNLma7CtNVQ8eylHbOHDhNrQ"
}
```

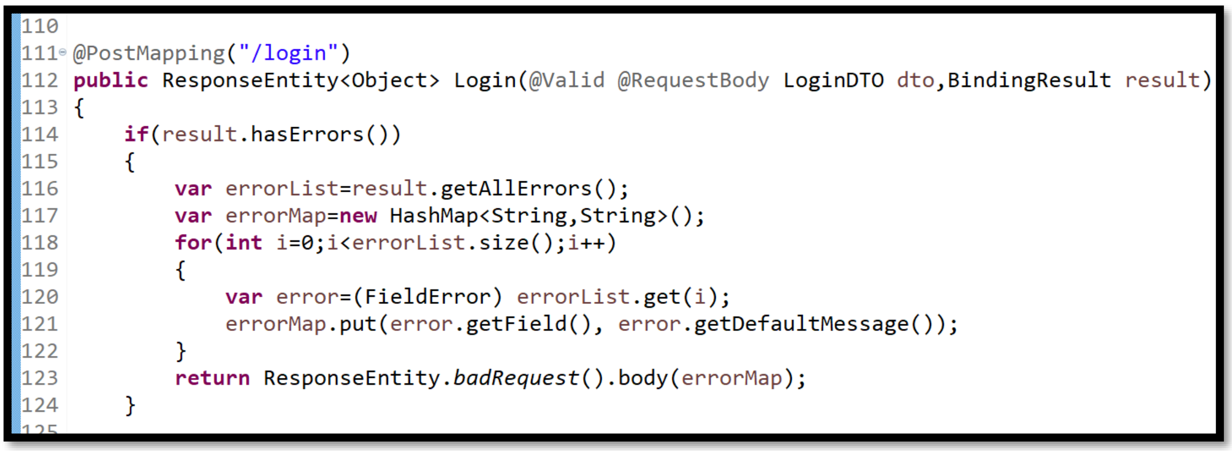
20. Check the record in mysql table



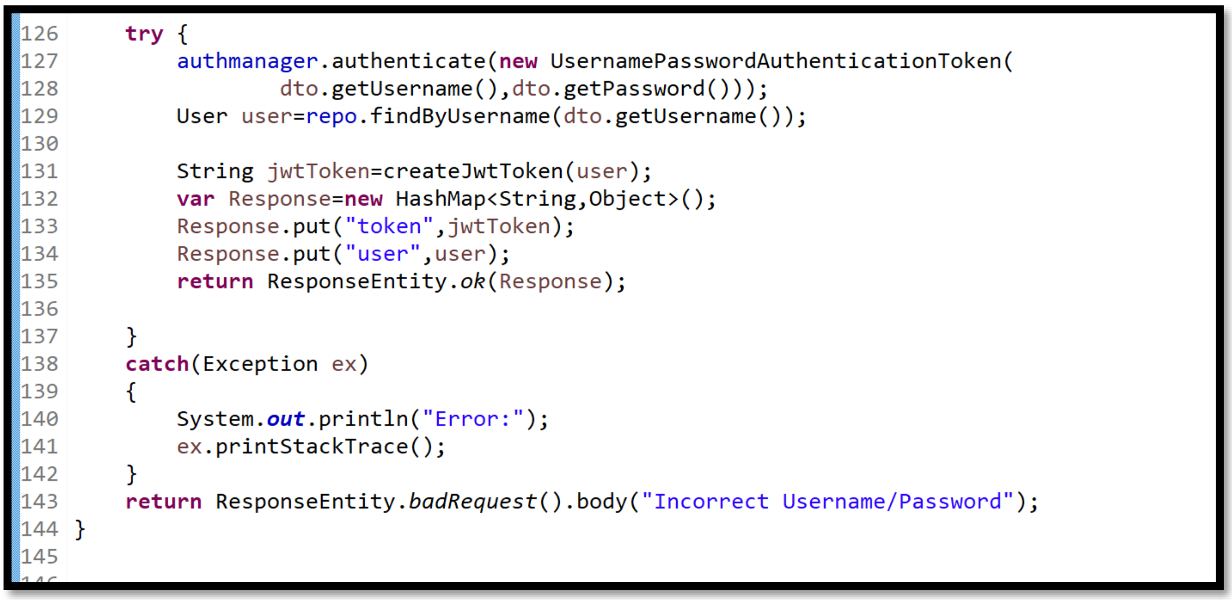
```
14
15 • select * from Users;
```

id	firstname	lastname	username	email	password	role
1	admin	test	admin	admin@gmail.com	$2a$10$kyIvmUrUrGzG9WXOJ82KSeHteEmrzN...	client
NULL	NULL	NULL	NULL	NULL	NULL	NULL

21. Inside “AccountsController.java” update the below endpoint method

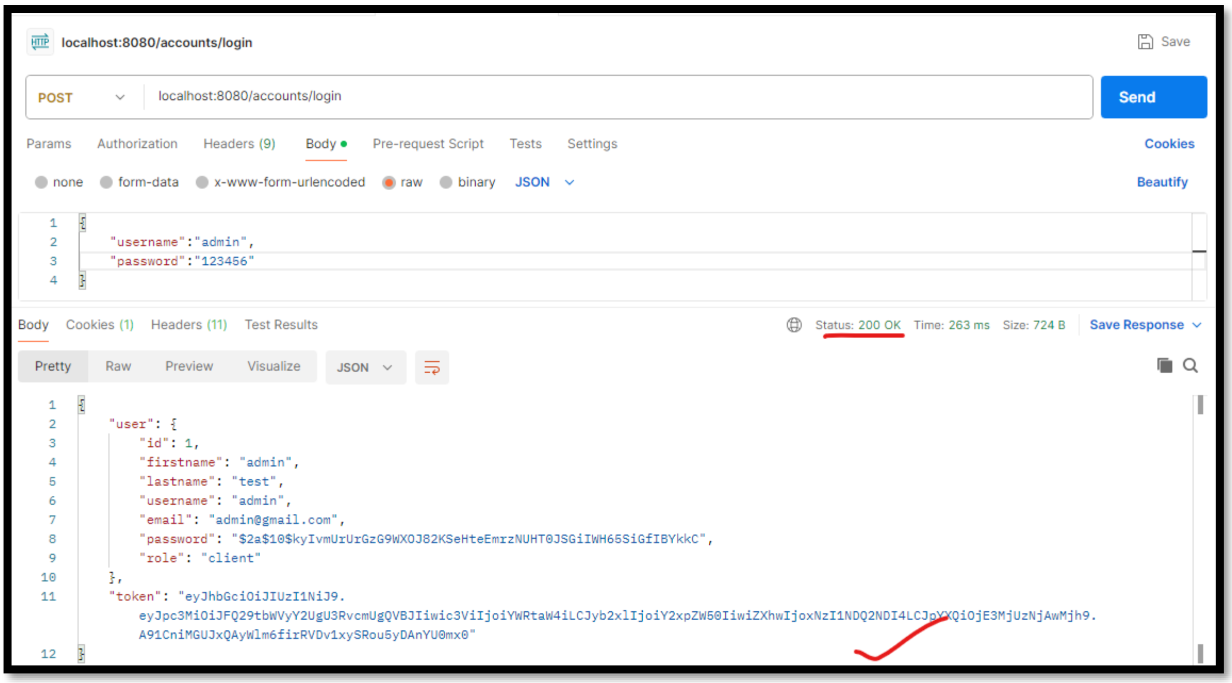


```java
@PostMapping("/login")
public ResponseEntity<Object> Login(@Valid @RequestBody LoginDTO dto,BindingResult result)
{
    if(result.hasErrors())
    {
        var errorList=result.getAllErrors();
        var errorMap=new HashMap<String,String>();
        for(int i=0;i<errorList.size();i++)
        {
            var error=(FieldError) errorList.get(i);
            errorMap.put(error.getField(), error.getDefaultMessage());
        }
        return ResponseEntity.badRequest().body(errorMap);
    }
```



```java
    try {
        authmanager.authenticate(new UsernamePasswordAuthenticationToken(
                dto.getUsername(),dto.getPassword()));
        User user=repo.findByUsername(dto.getUsername());

        String jwtToken=createJwtToken(user);
        var Response=new HashMap<String,Object>();
        Response.put("token",jwtToken);
        Response.put("user",user);
        return ResponseEntity.ok(Response);

    }
    catch(Exception ex)
    {
        System.out.println("Error:");
        ex.printStackTrace();
    }
    return ResponseEntity.badRequest().body("Incorrect Username/Password");
}
```

22. Check Login endpoint in postman



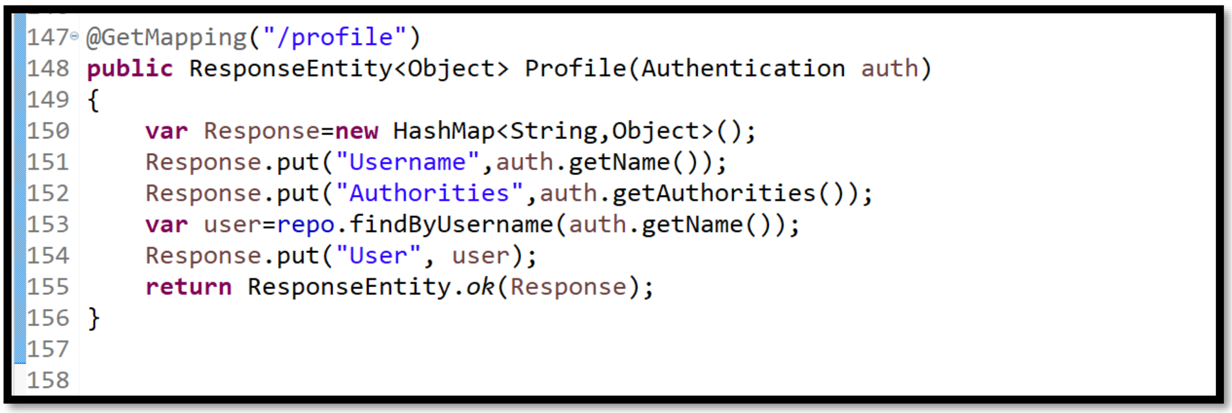
localhost:8080/accounts/login

```
{
    "username":"admin",
    "password":"123456"
}
```

Status: 200 OK Time: 263 ms Size: 724 B

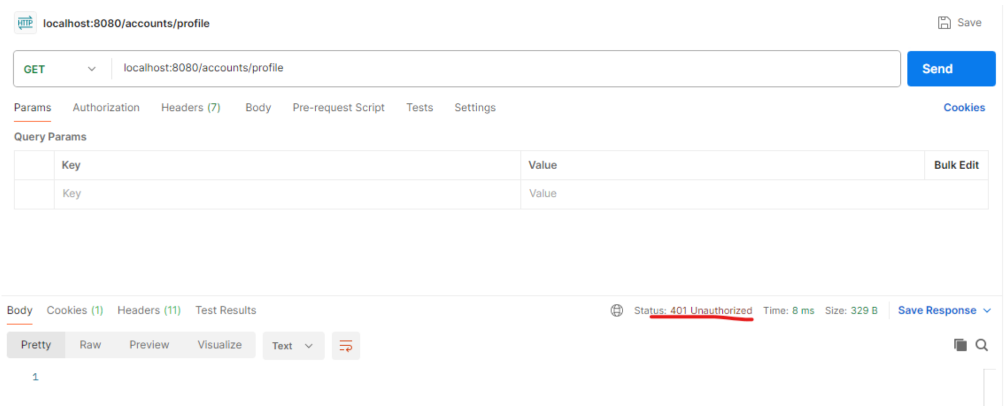
```
{
    "user": {
        "id": 1,
        "firstname": "admin",
        "lastname": "test",
        "username": "admin",
        "email": "admin@gmail.com",
        "password": "$2a$10$kyIvmUrUrGzG9WXOJ82KSeHteEmrzNUHT0JSGiIWH65SiGfIBYkkC",
        "role": "client"
    },
    "token": "eyJhbGciOiJIUzI1NiJ9.
        eyJpc3MiOiJFQ29tbWVyY2UgU3RvcmUgQVBJIiwic3ViIjoiYWRtaW4iLCJyb2xlIjoiY2xpZW50IiwiZXhwIjoxNzI1NDQ2NDI4LCJpYXQiOjE3MjUzNjAwMjh9.
        A91CniMGUJxQAyWlm6firRVDv1xySRou5yDAnYU0mx0"
}
```

23. create profile end-point “AccountsController”



```java
@GetMapping("/profile")
public ResponseEntity<Object> Profile(Authentication auth)
{
     var Response=new HashMap<String,Object>();
     Response.put("Username",auth.getName());
     Response.put("Authorities",auth.getAuthorities());
     var user=repo.findByUsername(auth.getName());
     Response.put("User", user);
     return ResponseEntity.ok(Response);
}
```

24. Check Profile end-point from “AccountsController” from postman

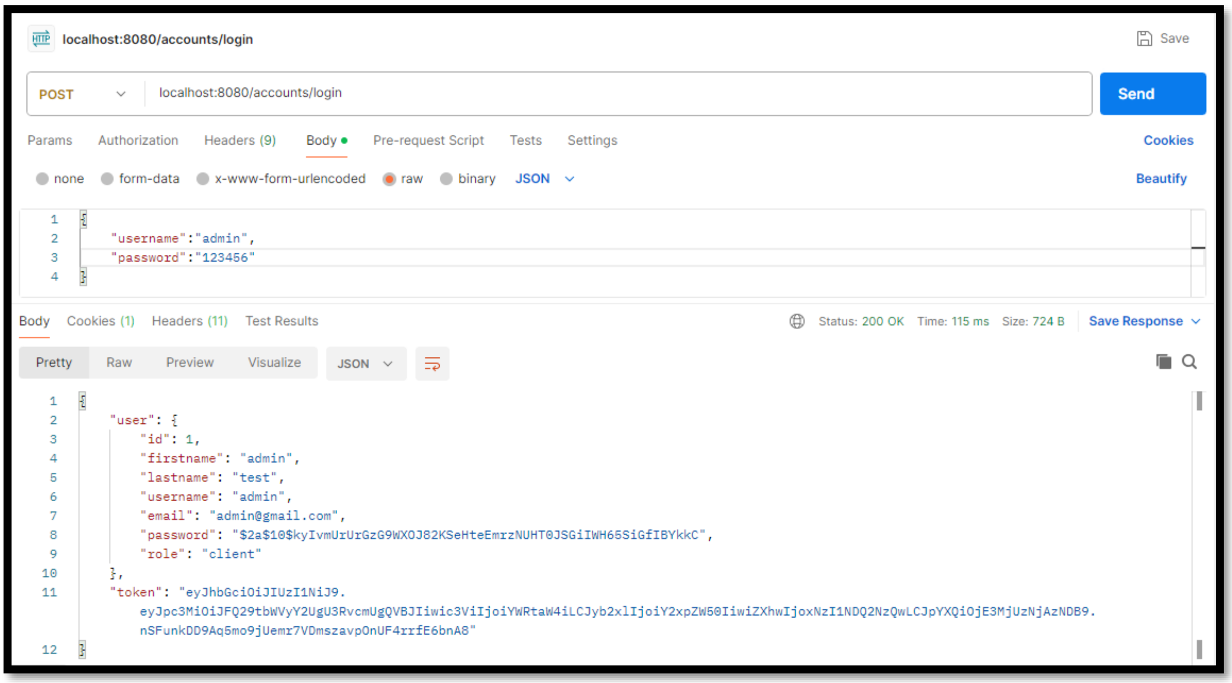


localhost:8080/accounts/profile

Status: 401 Unauthorized Time: 8 ms Size: 329 B

It returns 401-Status code bcz its mentioned under allowed requested urls. So that we have to pass jwt token along with every request other than mentioned in securityConfig

25. Get JET Token from 'Login" end point



localhost:8080/accounts/login

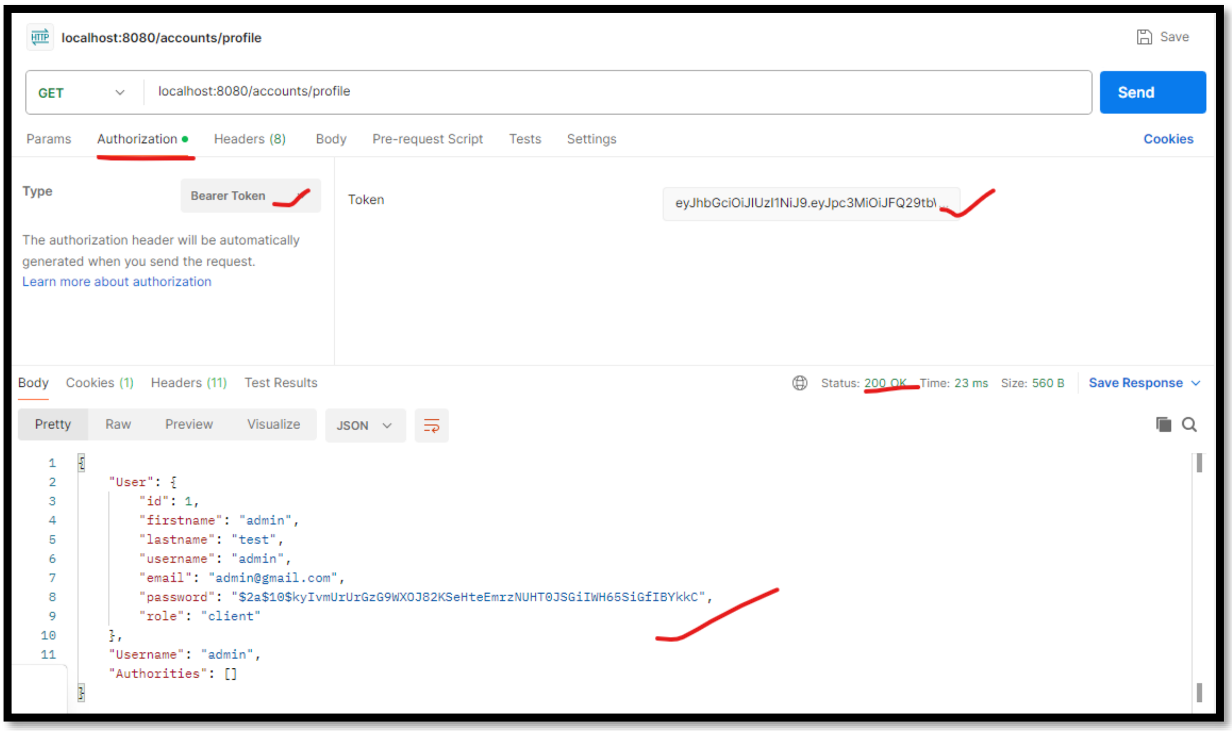
POST localhost:8080/accounts/login

```
{
    "username":"admin",
    "password":"123456"
}
```

Status: 200 OK Time: 115 ms Size: 724 B

```
{
    "user": {
        "id": 1,
        "firstname": "admin",
        "lastname": "test",
        "username": "admin",
        "email": "admin@gmail.com",
        "password": "$2a$10$kyIvmUrUrGzG9WXOJ82KSeHteEmrzNUHT0JSGiIWH65SiGfIBYkkC",
        "role": "client"
    },
    "token": "eyJhbGciOiJIUzI1NiJ9.
        eyJpc3MiOiJFQ29tbWVyY2UgU3RvcmUgQVBJIiwic3ViIjoiYWRtaW4iLCJyb2xlIjoiY2xpZW50IiwiZXhwIjoxNzI1NDQ2NzQwLCJpYXQiOjE3MjUzNjAzNDB9.
        nSFunkDD9Aq5mo9jUemr7VDmszavpOnUF4rrfE6bnA8"
}
```

26. Call the end-point profile along with the generated token



localhost:8080/accounts/profile

GET localhost:8080/accounts/profile

Authorization — Type: Bearer Token

The authorization header will be automatically generated when you send the request.
Learn more about authorization

Token: eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJFQ29tb\...

Status: 200 OK Time: 23 ms Size: 560 B

```
{
    "User": {
        "id": 1,
        "firstname": "admin",
        "lastname": "test",
        "username": "admin",
        "email": "admin@gmail.com",
        "password": "$2a$10$kyIvmUrUrGzG9WXOJ82KSeHteEmrzNUHT0JSGiIWH65SiGfIBYkkC",
        "role": "client"
    },
    "Username": "admin",
    "Authorities": []
}
```