Requirement 1 - Build REST APIs

Build Below REST APIs for Todo Management Module:

- Add Todo
- Get Todo
- Get All Todos
- Update Todo
- Delete Todo
- Complete Todo
- In-complete Todo

Requirement 2 - Build Frontend React App

Build Frontend React App (Consume REST APIs) for Todo Management Module

User should able perform below operations:

- Add new todo
- List all todos in a table
- Update particular todo
- Delete particular todo
- Mark todo as complete
- Mark todo as incomplete

Requirement 3 - Secure REST APIs

Secure all the Todo REST API using Spring Security and implement Role-Based Authorization

Requirement 4 - Build Register and Login REST APIs

1. Build Register REST API
2. Exception Handling in Register REST API
3. Build Login REST API

Requirement 5 - Register and Login Implementation in React App

Build Frontend React App (Consume REST APIs) for Registration and Login Module

User should able perform below operations:

- Register to Todo App
- Login to Todo App using Registered Credentials
- Logout from Todo App

Requirement 6 - Secure REST APIs using JWT (JSON Web Token)

- Secure Spring Boot REST APIs using Spring Security and JWT
- Use JWT in React App

Configure MySQL Database

```
spring.application.name=todo-management
spring.datasource.url=jdbc:mysql://localhost:3306/todo_management
spring.datasoruce.username=root
spring.datasource.password=root

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
```

Create Todo JPA Entity

```
@Setter
@Getter
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "todos")
public class Todo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "title", nullable = false)
    private String title;

    @Column(nullable = false)
    private String description;
    private boolean completed;
}
```

TodoRepository

```
public interface TodoRepository extends JpaRepository<Todo, Long> {
}
```

TodoDto

```
@Setter
@Getter
@NoArgsConstructor
@AllArgsConstructor
public class TodoDto {
    private Long id;
    private String title;
    private String description;
    private boolean completed;
}
```

Build Add Todo REST API

```java
@Service
@AllArgsConstructor
public class TodoServiceImpl implements TodoService{
    private TodoRepository todoRepository;

    @Override
    public TodoDto addTodo(TodoDto todoDto) {
        // convert TodoDto into Todo Jpa entity
        Todo todo = new Todo();
        todo.setTitle(todoDto.getTitle());
        todo.setDescription(todo.getDescription());
        todo.setCompleted(todoDto.isCompleted());

        //Todo Jpa entity
        Todo savedTodo = todoRepository.save(todo);

        // convert saved Todo Jpa entity to TodoDto object
        TodoDto savedTodoDto = new TodoDto();
        savedTodoDto.setId(savedTodo.getId());
        savedTodoDto.setTitle(savedTodo.getTitle());
        savedTodoDto.setDescription(savedTodo.getDescription());
        savedTodoDto.setCompleted(savedTodo.isCompleted());

        return savedTodoDto;
    }
```

ModelMapper library is used to convert dto into jpa entity and vice versa.

Steps – include its dependency in pom.xml, register its bean, inject it in serviceImpl class and then use it.

```java
@SpringBootApplication
public class TodoManagementApplication {

    @Bean
    public ModelMapper modelMapper(){
        return new ModelMapper();
    }

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

}
```

```java
@Service
@AllArgsConstructor
public class TodoServiceImpl implements TodoService{
    private TodoRepository todoRepository;
    private ModelMapper modelMapper;

    @Override
    public TodoDto addTodo(TodoDto todoDto) {
        // convert TodoDto into Todo Jpa entity
        Todo todo = modelMapper.map(todoDto, Todo.class);

        //Todo Jpa entity
        Todo savedTodo = todoRepository.save(todo);

        // convert saved Todo Jpa entity to TodoDto object
        TodoDto savedTodoDto = modelMapper.map(savedTodo, TodoDto.class);
        return savedTodoDto;
    }
}
```

```java
@RestController
@RequestMapping("api/todos")
@AllArgsConstructor
public class TodoController {
    private TodoService todoService;

    // Build Add Todo REST API
    @PostMapping
    public ResponseEntity<TodoDto> addTodo(@RequestBody TodoDto todoDto){
        TodoDto savedTodo = todoService.addTodo(todoDto);
        return new ResponseEntity<>(savedTodo, HttpStatus.CREATED);
    };
}
```

Build Get Todo REST API

```java
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException{
    public ResourceNotFoundException(String message){
        super(message);
    }
}
```

```java
public interface TodoService {
    TodoDto addTodo(TodoDto todoDto);

    TodoDto getTodo(Long id);
}
```

```java
@Override
public TodoDto getTodo(Long id){
    Todo todo = todoRepository.findById(id).orElseThrow(
            ()->new ResourceNotFoundException("Todo not found with id : " + id));
    return modelMapper.map(todo, TodoDto.class);
}
```

```java
@GetMapping("{id}")
public ResponseEntity<TodoDto> getTodo(@PathVariable("id") Long todoId){
    TodoDto todoDto = todoService.getTodo(todoId);
    return new ResponseEntity<>(todoDto, HttpStatus.OK);
}
```

Build Get All Todos REST API

TodoService.java | TodoServiceImpl.java | TodoController.java | ResourceNotFoun

```java
@Override
public List<TodoDto> getAllTodos(){
    List<Todo> todos = todoRepository.findAll();
    return todos.stream() Stream<Todo>
            .map((todo) -> modelMapper.map(todo, TodoDto.class)) Stream<TodoDto>
            .collect(Collectors.toList());
}
```

```java
@GetMapping("/all")
public ResponseEntity<List<TodoDto>> getAllTodos(){
    List<TodoDto> todos = todoService.getAllTodos();
//      return new ResponseEntity<>(todos, HttpStatus.OK);
    return ResponseEntity.ok(todos);
}
```

Build Update Todo REST API

va | Todo.java | TodoService.java | TodoServiceImpl.java | TodoController.jav

```java
@Override
public TodoDto updateTodo(TodoDto todoDto, Long id){
    Todo todo = todoRepository.findById(id).orElseThrow(
            ()->new ResourceNotFoundException("Todo not found with id : " + id));
    todo.setTitle(todoDto.getTitle());
    todo.setDescription(todoDto.getDescription());
    todo.setCompleted(todoDto.isCompleted());

    Todo updatedTodo = todoRepository.save(todo);
    return modelMapper.map(updatedTodo, TodoDto.class);
}
```

```java
@PutMapping("{id}")
 public ResponseEntity<TodoDto> updateTodo(@RequestBody TodoDto todoDto,
                                           @PathVariable("id") Long todoId){
    TodoDto updatedTodo = todoService.updateTodo(todoDto, todoId);
    return ResponseEntity.ok(updatedTodo);
}
```

Build Delete Todo REST API

ava × TodoService.java TodoServiceImpl.java × TodoController.java Resourc

```java
@Override
public void deleteTodo(Long id){
    Todo todo = todoRepository.findById(id).orElseThrow(
            ()->new ResourceNotFoundException("Todo not found with id : " + id));
    todoRepository.deleteById(id);
}
```

```java
@DeleteMapping("{id}")
 public ResponseEntity<String> deleteTodo(@PathVariable("id") Long id){
    todoService.deleteTodo(id);
    return ResponseEntity.ok( body: "Todo deleted successfully");
}
```

Build Complete Todo & InComplete Todo REST API

We use PutMapping to update the resource completely and PatchMapping to update it partially.

TodoServiceImpl.java × Boolean.java TodoController.java ResourceNotFour

```java
@Override
public TodoDto completeTodo(Long id){
    Todo todo = todoRepository.findById(id).orElseThrow(
            ()->new ResourceNotFoundException("Todo not found with id : " + id));
    todo.setCompleted(Boolean.TRUE);
    Todo updatedTodo = todoRepository.save(todo);
    return modelMapper.map(updatedTodo, TodoDto.class);
}
```

```java
@PatchMapping("{id}/complete")
public ResponseEntity<TodoDto> completeTodo(@PathVariable("id") Long todoId){
    TodoDto updatedTodo = todoService.completeTodo(todoId);
    return ResponseEntity.ok(updatedTodo);
}
```

Authentication

Process of verifying the identify of a user or system attempting to access a resource. Ex- password based, biometric, multifactor authentication etc.

Authorization

Process of determing what actions an authenticatied user or system is allowed to perform on a resource. In simple term, authorization determines what actions they are allowed to perform once their identity has been verified.

Ex- role based authentication : A user with a “manager” role can approve or reject employee leave requests, while a user with an “employee” role can only submit leave requests.

Spring security provides built-in support for authentication, authorization and projection against common attacks. To use it add spring-boot-starter-security dependency in ur project.

springSecurityFilterChain bean is responsible for all the security (protecting the application URLs, validating submitted username and passwords, redirecting to the login form, and so on) within your application. When an HTTP request comes in the springSecurityFilterChain delegates the request to the appropriate SecurityFilterChain based on matching conditions.

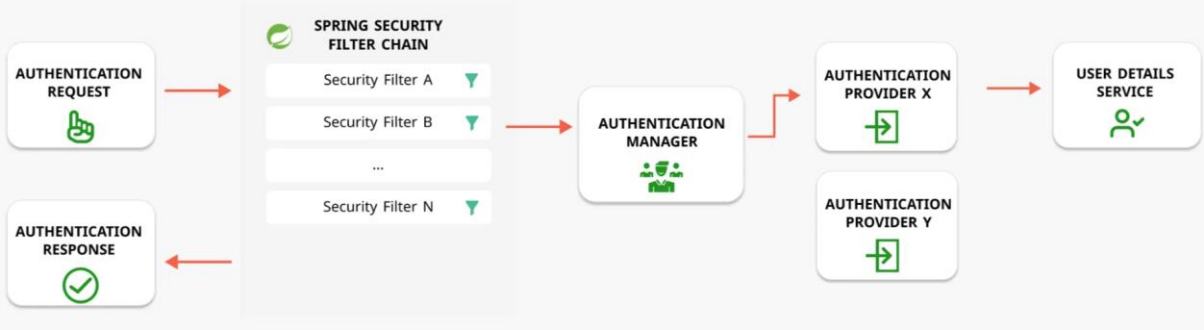
SecurityFilterChain: Represents individual security rules for specific request patterns.

Spring Boot Auto Configuration for Spring Security

Spring Boot auto configures below features:

- **spring-boot-starter-security** starter that aggregates Spring Security-related dependencies together.
- Enables Spring Security’s default configuration, which creates a servlet Filter as a bean named **springSecurityFilterChain**. Provides default login form for you.
- Creates default user with a username as **user** and a randomly generated password that is logged to the console (Ex: 8e557245-73e2-4286-969a-ff57fe326336).
- Spring boot provides properties to cusomize default user’s username and password
- Protects the password storage with **BCrypt** algorithm
- Lets the user log out (default logout feature)
- CSRF attack prevention (enabled by default)
- If Spring Security is on the classpath, Spring Boot automatically secures all HTTP endpoints with “basic” authentication.

In postman, we have to pass username & password for basic authentication.



Configure credentials like this -

TodoService.java application.properties m pon

```
spring.security.user.name=arjun
spring.security.user.password=password
spring.security.user.roles=ADMIN, USER, MANAGER
```

Configure Basic Authentication

In project we don't use form based authentication (where u get spring security default form) but we have used basic authentication.

HTTP Basic authentication involves sending a verified username and password with your request. In the request Headers, the Authorization header passes the API a Base64 encoded string representing your username:password. Browser gives us popup to enter credentials.

Inorder to configure spring security we have to create a SecurityFilterChain bean. You can define multiple SecurityFilterChain beans for different security requirements in your application. Bean name is that of method name or can be given using @Bean(name = "……")

Http.build() method return DefaultSecurityFilterChain class object which is implementation of interface SecurityFilterChain.

.authorizeHttpRequests() is used to authorize all the incoming http requests.

REST API - specific type of web API that follow the principles of REST, an architectural style. Like its stateless which means Each request from a client to the server must contain all the information the server needs to fulfill the request. The server does not store any client state between requests. Follow standard HTTP methods such as GET, PUT, POST, etc.

Normal API – any api which may not adhere to REST principles. Like it can be stateful or stateless. May use any protocol – HTTP, SOAP etc. Stateful means server maintains info about client's state across multiple requests typically using sessions. Once authenticated, a session ID (or similar identifier) is sent back to the client, typically in a cookie. Client sends the session ID with every subsequent request, usually as a cookie/header and server relies on it to authenticate requests. No need for the client to re-authenticate as long as the session is active.

How CSRF Works in Stateful APIs

1. **Initial Authentication:**
   - A user logs into a web application (e.g., a bank) and their session is created.
   - The server sets a session cookie, which is sent automatically with every request from the client.
2. **Attacker's Strategy:**
   - The attacker tricks the user into visiting a malicious website (or clicking on a malicious link) while they are still logged into the target application (e.g., a banking website).
   - The malicious site sends a request to the target application using the user's credentials (because the session cookie is automatically included by the browser in the request).
3. **Executing the Attack:**
   - Since the user is logged in, the request appears legitimate to the target application because it includes the valid session cookie.
   - The server processes the request and performs actions (e.g., transferring money, changing account details) without the user's consent.

Example of a CSRF Attack

1. **User Logs into Target Application:**
   - A user logs into their bank account at `bank.com`. The server creates a session and sends a session cookie `sessionId=abcd1234` to the browser.
2. **Malicious Website:**
   - The user visits a malicious website, `attacker.com`, which contains the following HTML code:

```html
<img src="https://bank.com/transfer?amount=1000&toAccount=attackerAccount" />
```

   - The malicious website causes the browser to send a request to `bank.com`, using the session cookie `sessionId=abcd1234` (which is automatically included by the browser).
3. **Request is Processed:**
   - The request is sent to the bank's server, which recognizes the session and assumes the request is legitimate.
   - The bank transfers $1000 to the attacker's account without the user's knowledge or consent.

In a **CSRF attack**, the attacker **does not directly get** the session ID but exploits the fact that the **browser automatically sends the session cookie** to the server with any request to the same domain. The server trusts the session cookie and processes the request as if it was made by the legitimate user.

Configure multiple users in In-Memory object

In the Spring Framework, a **Spring Bean** is an object that is managed by the Spring **Inversion of Control (IoC) container.**

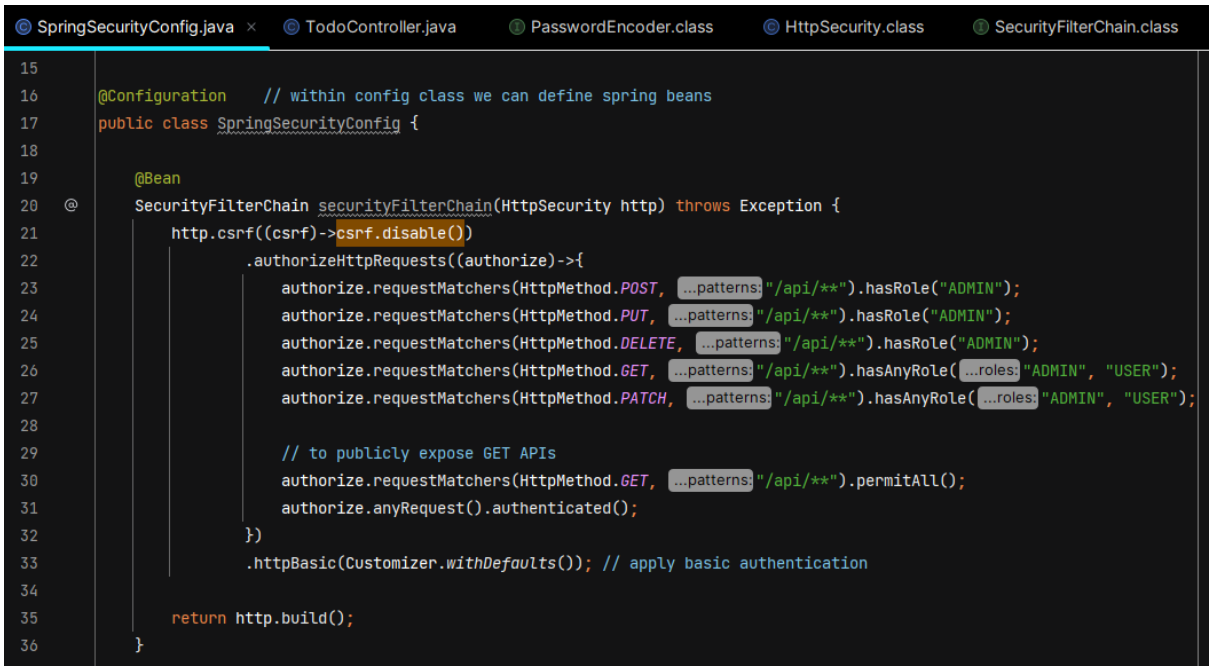
Spring security expects password in encoded format so we create PasswordEncoder bean.

SpringSecurityConfig.java × PasswordEncoder.class HttpSecurity.class Securit

```
29      @Bean
30  @   public static PasswordEncoder passwordEncoder(){
31          return new BCryptPasswordEncoder();
32      }
33
34      @Bean
35      public UserDetailsService userDetailsService(){
36          UserDetails ramesh = User.builder()
37                  .username("ramesh")
38                  .password(passwordEncoder().encode( rawPassword: "password"))
39                  .roles("USER")
40                  .build();
41
42          UserDetails admin = User.builder()
43                  .username("admin")
44                  .password(passwordEncoder().encode( rawPassword: "admin"))
45                  .roles("ADMIN")
46                  .build();
47
48          return new InMemoryUserDetailsManager(ramesh, admin);
49      }
50  }
51
```

Role-Based Authorization

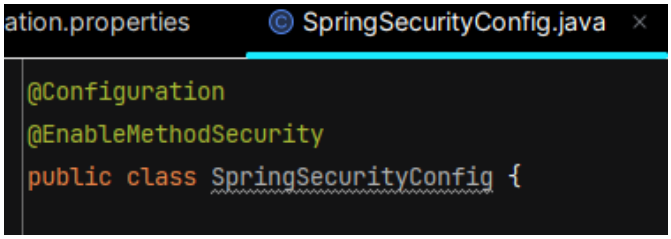
Only ADMIN can access add, update and delete todo REST API



```java
@Configuration    // within config class we can define spring beans
public class SpringSecurityConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf((csrf)->csrf.disable())
                .authorizeHttpRequests((authorize)->{
                    authorize.requestMatchers(HttpMethod.POST, ...patterns:"/api/**").hasRole("ADMIN");
                    authorize.requestMatchers(HttpMethod.PUT, ...patterns:"/api/**").hasRole("ADMIN");
                    authorize.requestMatchers(HttpMethod.DELETE, ...patterns:"/api/**").hasRole("ADMIN");
                    authorize.requestMatchers(HttpMethod.GET, ...patterns:"/api/**").hasAnyRole(...roles:"ADMIN", "USER");
                    authorize.requestMatchers(HttpMethod.PATCH, ...patterns:"/api/**").hasAnyRole(...roles:"ADMIN", "USER");

                    // to publicly expose GET APIs
                    authorize.requestMatchers(HttpMethod.GET, ...patterns:"/api/**").permitAll();
                    authorize.anyRequest().authenticated();
                })
                .httpBasic(Customizer.withDefaults()); // apply basic authentication

        return http.build();
    }
```

Method Level Security

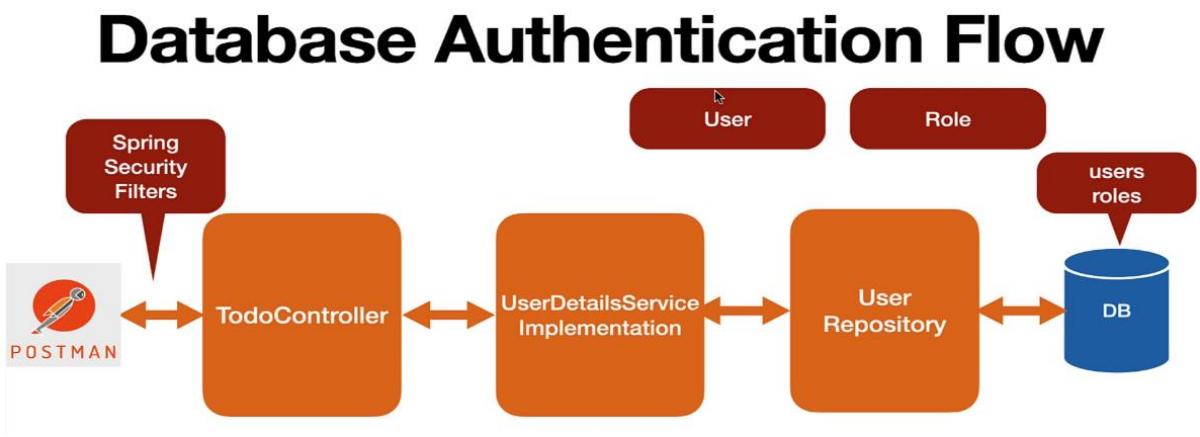
We need to use @EnableMethodSecurity and @PreAuthorize



```java
@Configuration
@EnableMethodSecurity
public class SpringSecurityConfig {
```

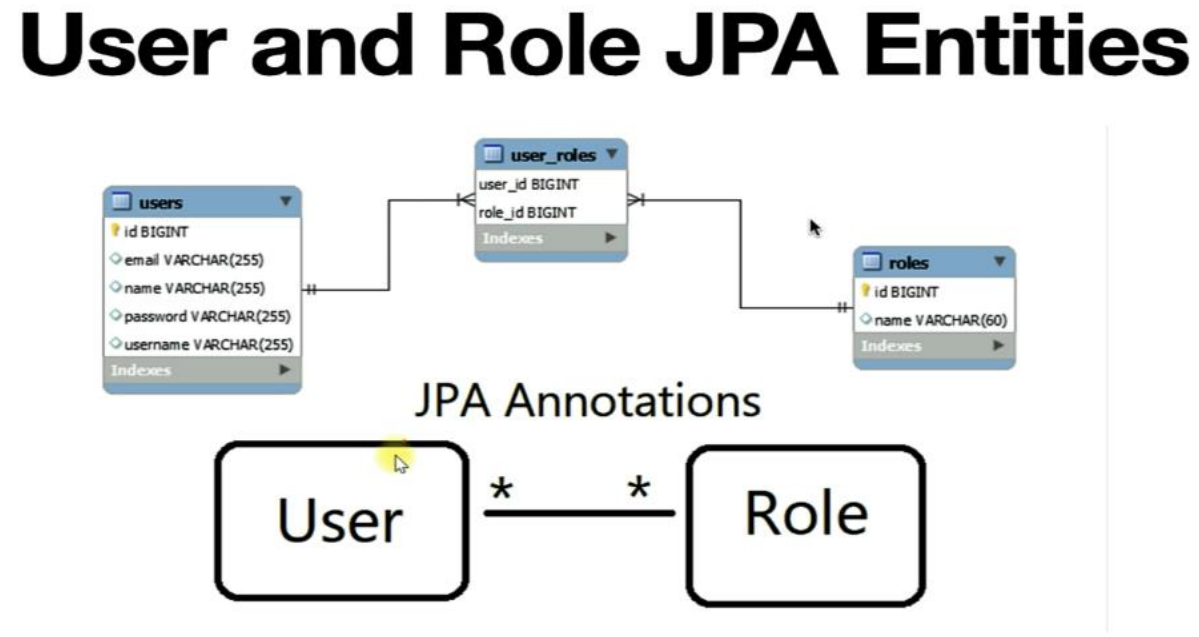
```java
// Build Add Todo REST API
@PreAuthorize("hasRole('ADMIN')")
@PostMapping
public ResponseEntity<TodoDto> addTodo(@RequestBody TodoDto todoDto){
    TodoDto savedTodo = todoService.addTodo(todoDto);
    return new ResponseEntity<>(savedTodo, HttpStatus.CREATED);
};

@PreAuthorize("hasAnyRole('ADMIN', 'USER')")
@GetMapping("{id}")
public ResponseEntity<TodoDto> getTodo(@PathVariable("id") Long todoId){
    TodoDto todoDto = todoService.getTodo(todoId);
    return new ResponseEntity<>(todoDto, HttpStatus.OK);
}
```

Database Authentication Flow



Request from Postman will first come to spring security filters, which extract the username and password from header and load the corresponding user object from the DB and validate username & password.

We'll create User & Role jpa entity and hibernate will create users & roles tables in DB. We have many to many mapping between user and role as 1 user can have multiple roles and 1 roles can be assigned to multiple user.



Whenever we create many to many mapping between 2 jpa entities then 3rd table is created called join table. It maintains details from users and roles table.

Users table contain has a primary key ID, this primary key becomes a foreign key in 3rd table. Roles table has primary key ID, this primary key ID becomes a foreign key in a third table.

FetchType.EAGER means whenever we load User entity it will also load its role.

In SQL, a **cascade** is a mechanism used to ensure that changes made to one table automatically propagate to related tables, maintaining referential integrity. **CascadeType.ALL** propagates all operations (persist, merge, remove, refresh, detach) from the parent entity to the associated child entities.

Whenever we save User it will also save Roles as User is parent and Role is its child.

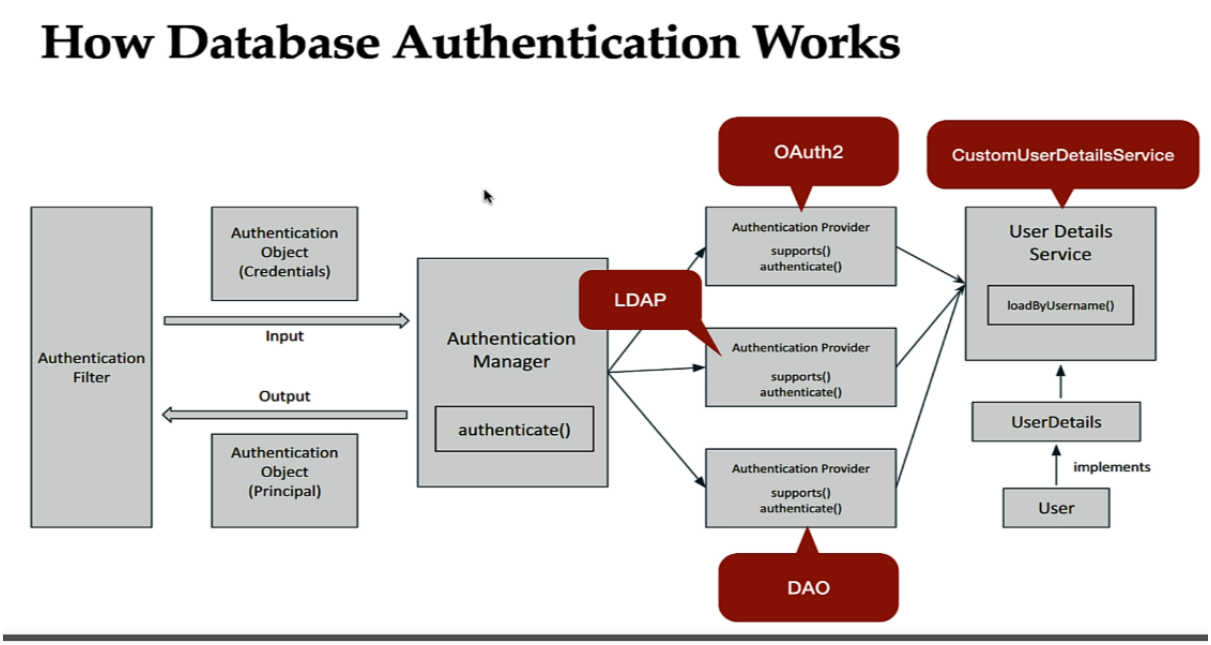
```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name="users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @Column(nullable = false, unique = true)
    private String username;
    @Column(nullable = false, unique = true)
    private String email;
    @Column(nullable = false)
    private String password;

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable(name="users_roles",
            joinColumns = @JoinColumn(name="user_id", referencedColumnName = "id"),
            inverseJoinColumns = @JoinColumn(name="role_id", referencedColumnName = "id")
    )
    private Set<Role> roles;
}
```

```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name="roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```

We'll create some custom query method in UserRepository. findBy is a keyword which retreive entity by field name. existsBy is a keyword which check whether user object is present in database with given fieldname. Spring data jpa will internally create query with these method name.

```java
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByUsername(String username);
    Boolean existsByEmail(String email);
    Optional<User> findByUsernameOrEmail(String username, String email);
}
```

```java
public interface RoleRepository extends JpaRepository<Role, Long> {
}
```



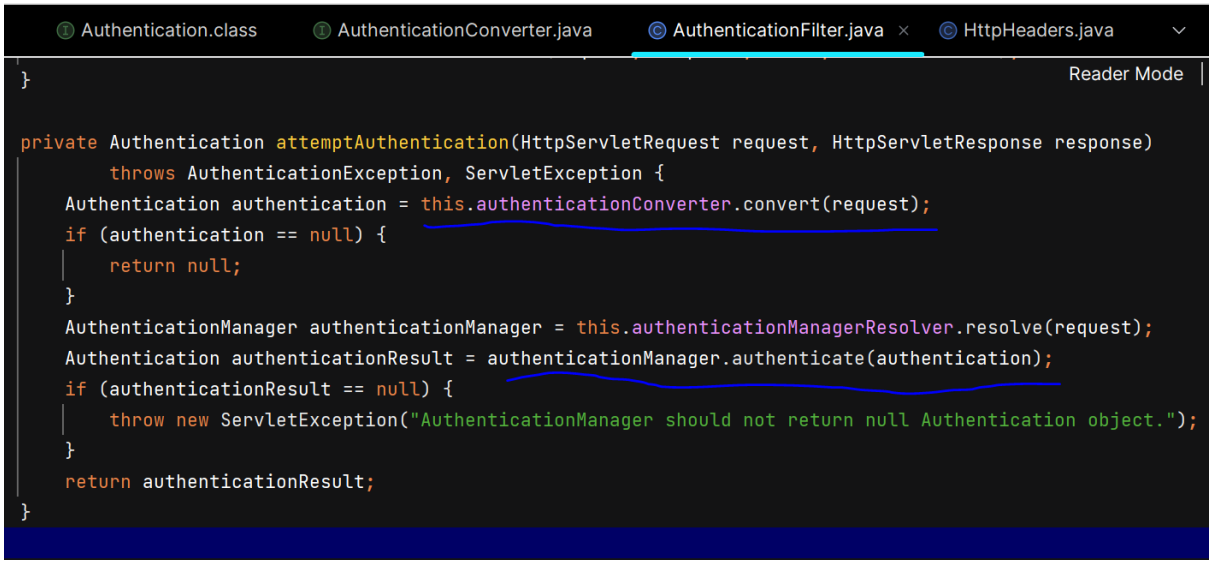
Whenever user submit a request, the username and password are stored in header and the request first comes to AuthenticationFilter, and then AuthenticationFilter will create an instance of Authentication object. This object basically contains the username and password, and then AuthenticationFilter will pass this authentication object to AuthenticationManager. This component manages different authentication providers.

AuthenticationManager's authenticate() method internally call this supports() method of all these authentication providers to check with authentication provider provides a support. If supports() returns true, the authenticate() method of the corresponding AuthenticationProvider is invoked.

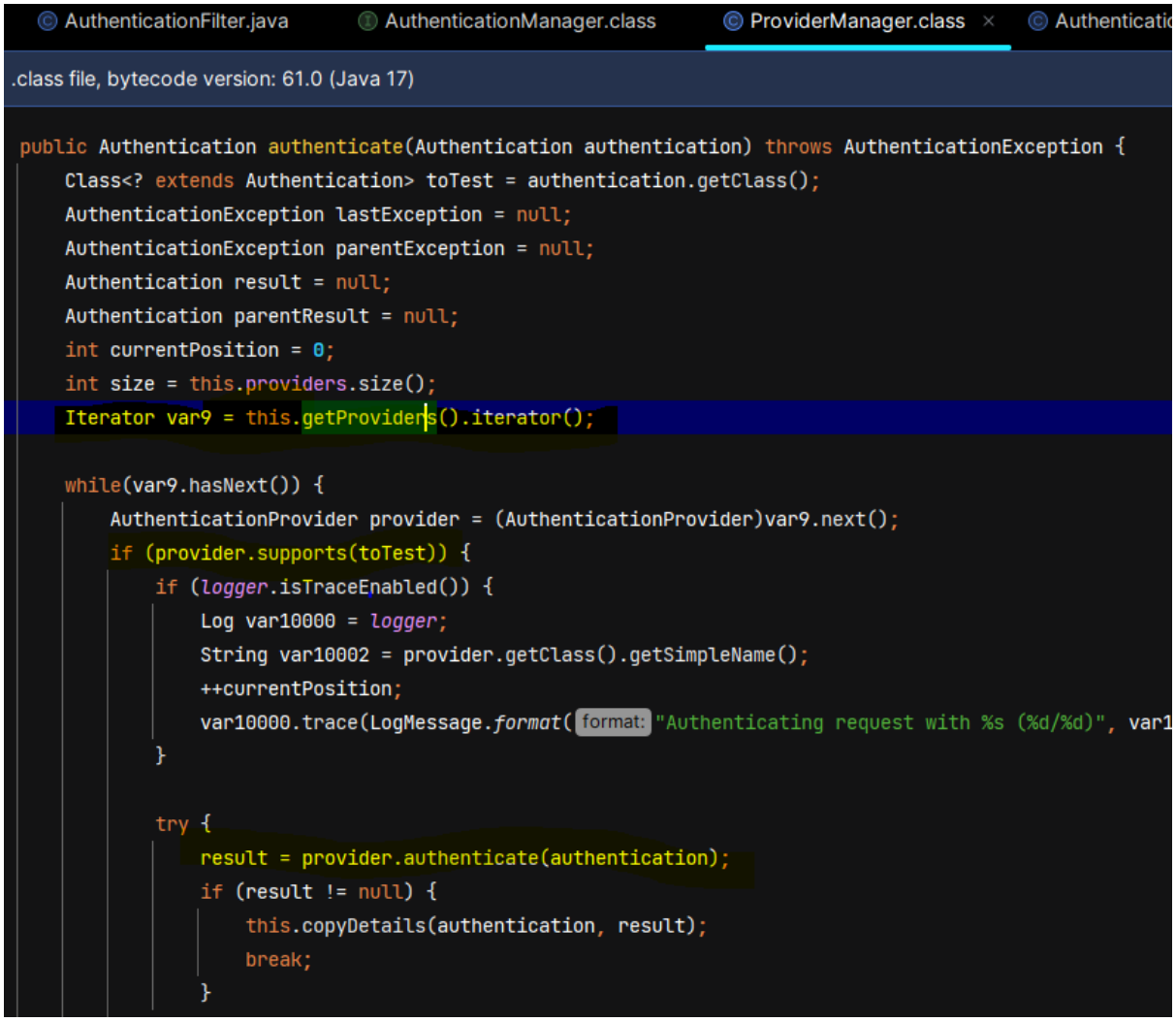
Each authentication provider uses loadByUsername method to load the user object. In case of database authentication we have to create a CustomUserDetailsService class that implements UserDetailsService interface, and then we need to provide impl for loadByUsername method.

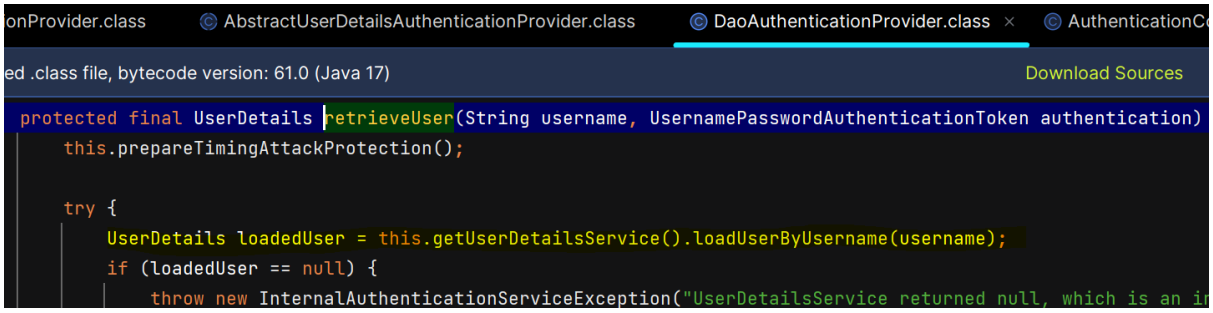
Internal working - Request comes to doFilterInternal() method of AuthenticationFilter class. Inside this we create Authentication object by extracting the "Authorization" header which contains

username:password in base64 format and finally object of UsernamePasswordAuthenticationToken is created. This class internally implements the Authentication interface.



DaoAuthenticationProvider is responsible to authenticate the requests related to database.

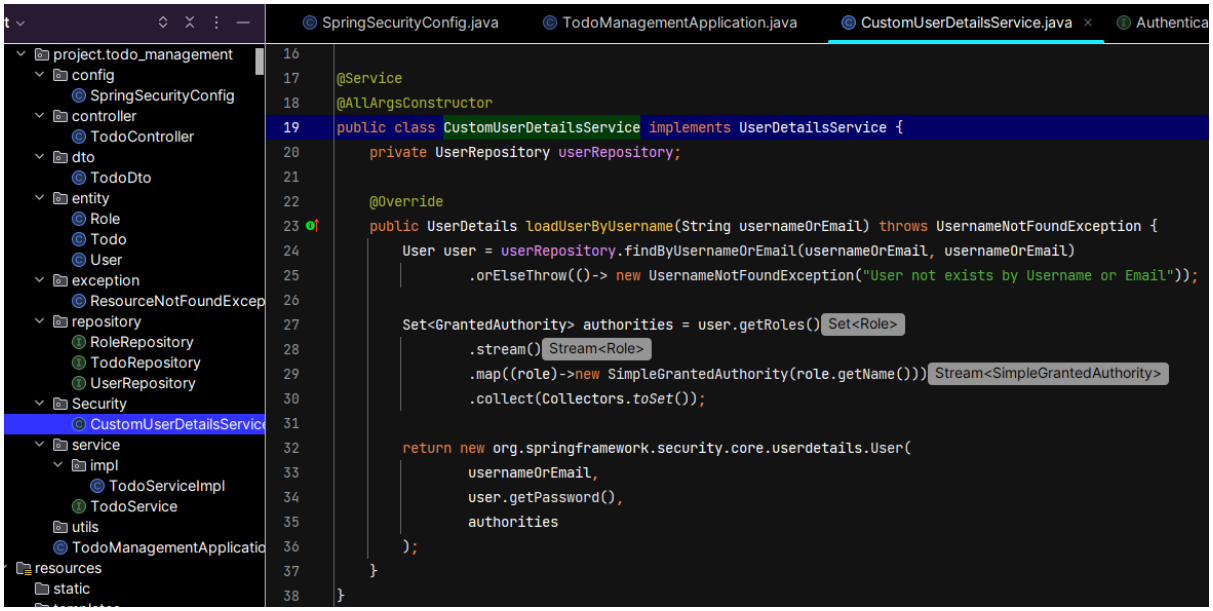




Creating CustomUserDetailsService class

we'll provide implementation for loadByUsername() method to load the user from db and give it to DaoAuthenticationProvider which will provide that user to the authentication manager.

User object has a set of Role, so we convert a set of Role into a set of GrantedAuthority.



Database Authentication

We'll use UserDetailsService interface to inject the dependency so to achieve loose coupling.

spring security internally add ROLE_ prefix for roles

id	name
1	ROLE_ADMIN
2	ROLE_USER
	NULL

id	email	name	password	username
1	arjun@ukg.com	arjun	$2a$13$...	arjun
2	admin@google.c...	admin	$2a$12$...	admin

user_id	role_id
2	1
1	2

Whenever we inject UserDetailsService in a SpringSecurityConfig class, then spring security 6 will automatically use this UserDetailsService and it will call its loadUserByUsername method. We don't have to explicitly provide this UserDetailsService instance to the authentication manager.

```
@Configuration
@EnableMethodSecurity
@AllArgsConstructor
public class SpringSecurityConfig {

    private UserDetailsService userDetailsService;

    @Bean
    public AuthenticationManager authenticationManager(
            AuthenticationConfiguration configuration) throws Exception {
        return configuration.getAuthenticationManager();
    }

    @Bean
    public static PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(); }

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf((csrf) -> csrf.disable())
                .authorizeHttpRequests((authorize) -> {
                    authorize.anyRequest().authenticated();
                })
                .httpBasic(Customizer.withDefaults()); // apply basic authentication

        return http.build();
    }
}
```

We stored password in DB using Bcrypt encoder. Spring Security uses the configured PasswordEncoder (e.g., BCryptPasswordEncoder) to hash the plain text password.

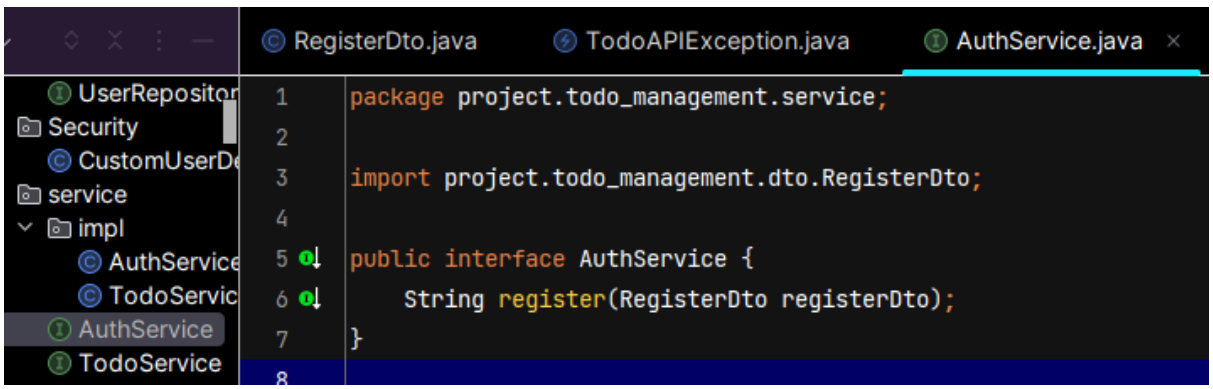
Build Register REST API

Development Steps

1. Create **RegisterDto** Class
2. Create Custom Exception - **TodoAPIException**
3. Create **AuthService** interface and define **register()** method
4. Create **AuthServiceImpl** class that implements **AuthService** interface and it's **register()** method
5. Create **AuthController** and build Register REST API
6. Test Register REST API using Postman Client

```java
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class RegisterDto {
    private String name;
    private String username;
    private String email;
    private String password;
}
```

```java
@Getter
@AllArgsConstructor
public class TodoAPIException extends RuntimeException{
    private HttpStatus status;
    private String message;
}
```



```java
@Service
@AllArgsConstructor
public class AuthServiceImpl implements AuthService {
    private UserRepository userRepository;
    private RoleRepository roleRepository;
    private PasswordEncoder passwordEncoder;

    @Override
    public String register(RegisterDto registerDto) {
        // check username & email is already exists in DB
        if(userRepository.existsByUsername(registerDto.getUsername())){
            throw new TodoAPIException(HttpStatus.BAD_GATEWAY, "Username already exist!");
        }
        if(userRepository.existsByEmail(registerDto.getEmail())){
            throw new TodoAPIException(HttpStatus.BAD_GATEWAY, "Email already exist!");
        }

        User user = new User();
        user.setName(registerDto.getName());
        user.setUsername(registerDto.getUsername());
        user.setEmail(registerDto.getEmail());
        user.setPassword(passwordEncoder.encode(registerDto.getPassword()));

        Set<Role> roles = new HashSet<>();
        Role role = roleRepository.findByName("ROLE_USER");
        roles.add(role);
        user.setRoles(roles);

        userRepository.save(user);

        return "User Registered Successfully !!";
    }
}
```

```java
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByUsername(String username);
    Boolean existsByEmail(String email);
    Boolean existsByUsername(String username);
    Optional<User> findByUsernameOrEmail(String username, String email);
}
```

```java
public interface RoleRepository extends JpaRepository<Role, Long> {
    Role findByName(String name);
}
```

```java
@RestController
@RequestMapping("/api/auth")
@AllArgsConstructor
public class AuthController {
    private AuthService authService;

    // build register REST API
    @PostMapping("/register")
    public ResponseEntity<String> register(@RequestBody RegisterDto registerDto){
        String response = authService.register(registerDto);
        return new ResponseEntity<>(response, HttpStatus.CREATED);
    }
}
```

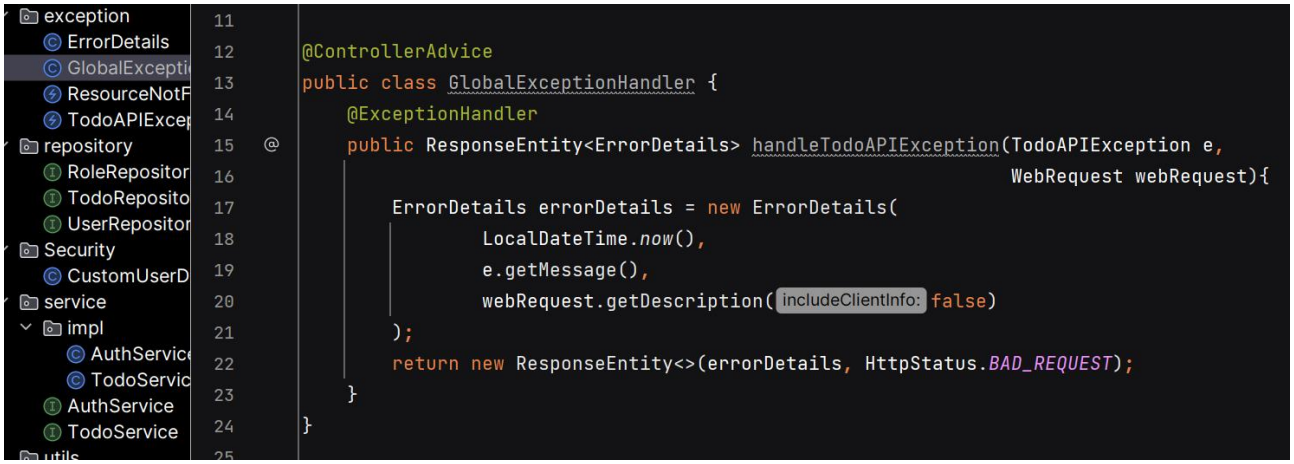
```java
@Configuration
@EnableMethodSecurity
@AllArgsConstructor
public class SpringSecurityConfig {
    private UserDetailsService userDetailsService;

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf((csrf) -> csrf.disable())
                .authorizeHttpRequests((authorize) -> {
                    authorize.requestMatchers( ...patterns: "/api/auth/**").permitAll();
                    authorize.anyRequest().authenticated();
                })
                .httpBasic(Customizer.withDefaults()); // apply basic authentication

        return http.build();
    }
```

Exception Handling in Register REST API

When user already exist in db and we still submit register rest API then we should get proper exception details in response. Within GlobalExceptionHandler class we'll handle all the exception that occur in our application.

We use **@ControllerAdvice** annotation to handle the exceptions globally across all controllers.

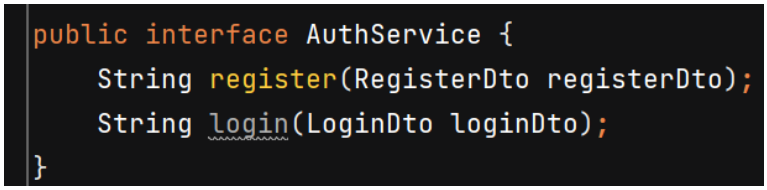
@ExceptionHandler used on method that handle specific exception.



Build Login REST API

Development Steps

1. Create **LoginDto** Class
2. Define **login()** method in **AuthService** interface
3. Implement **login()** method in **AuthServiceImpl** class
4. Build Login REST API
5. Test Login REST API using Postman Client



```java
@Service
@AllArgsConstructor
public class AuthServiceImpl implements AuthService {
    private UserRepository userRepository;
    private RoleRepository roleRepository;
    private PasswordEncoder passwordEncoder;
    private AuthenticationManager authenticationManager;

    @Override
    public String login(LoginDto loginDto){
        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                loginDto.getUsernameOrEmail(),
                loginDto.getPassword()
        ));

        // store authentication object in security context holder
        SecurityContextHolder.getContext().setAuthentication(authentication);
        return "User Logged-in successfully";
    }
```

```java
@RestController
@RequestMapping("/api/auth")
@AllArgsConstructor
public class AuthController {
    private AuthService authService;

    // Build Login REST API
    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestBody  LoginDto loginDto){
        String response = authService.login(loginDto);
        return new ResponseEntity<>(response, HttpStatus.OK);
    }
```

JWT (JSON Web Token)

What is JWT

- 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.
- JWT, or JSON Web Tokens (RFC 7519), is a standard that is mostly used for securing REST APIs.
- JWT is best way to communicate securely between client and server
- **JWT follows stateless authentication mechanism**

When should you use JSON Web Tokens?

- Authorization
- Information Exchange

This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays.

What is the JSON Web Token structure?

- JSON Web Tokens consist of three parts separated by dots (.), which are:
- Header
- Payload
- Signature

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

```
{
   "sub": "1234567890",
   "name": "John Doe",
   "admin": true
}
```

```
HMACSHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload),
secret)
```

xxxxx.yyyyy.zzzzz

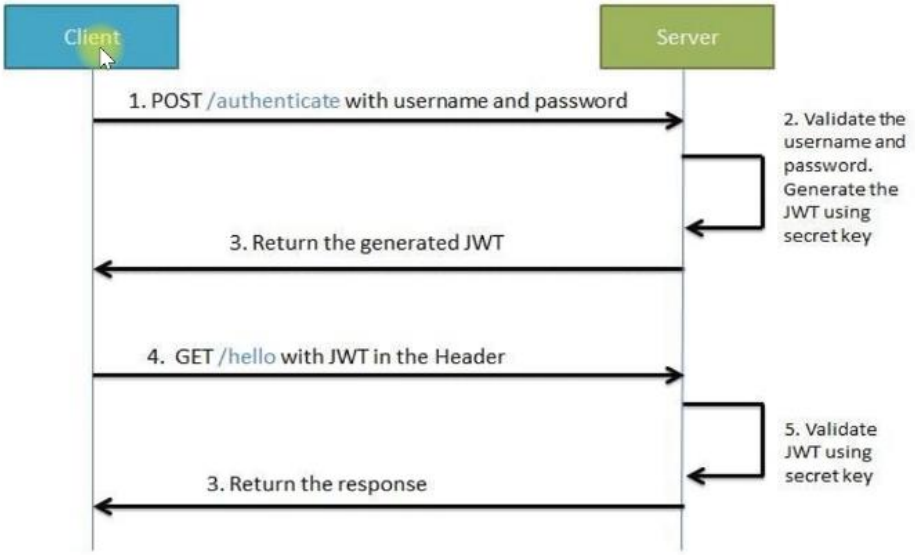
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbkBnbWFpbC5jb20iLCJpYXQiOjE2MTY1NjY5NDksImV4cCI6MTYxNzE3MTc0OX0.RVggbCFH2VGRZw9-pptLI7EKgp2BYxfOw8DXoE22MVTGJUBer600dx4yUZyd-TeFvBPflOKH9Rbi8SOvzYmIAA

Header consists of alogorithm being used and type of token.

Payload contains the claims. It means here the statement about an entity or a user and the additional data.

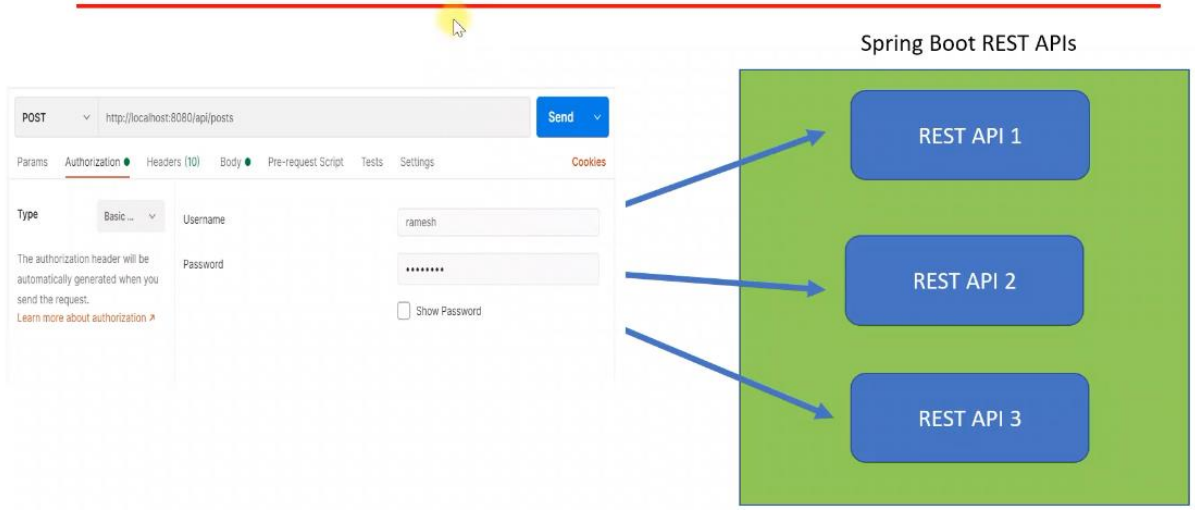
To create a signature, we have to take the encoded header and encoded payload and a secret key

How do JSON Web Tokens work?

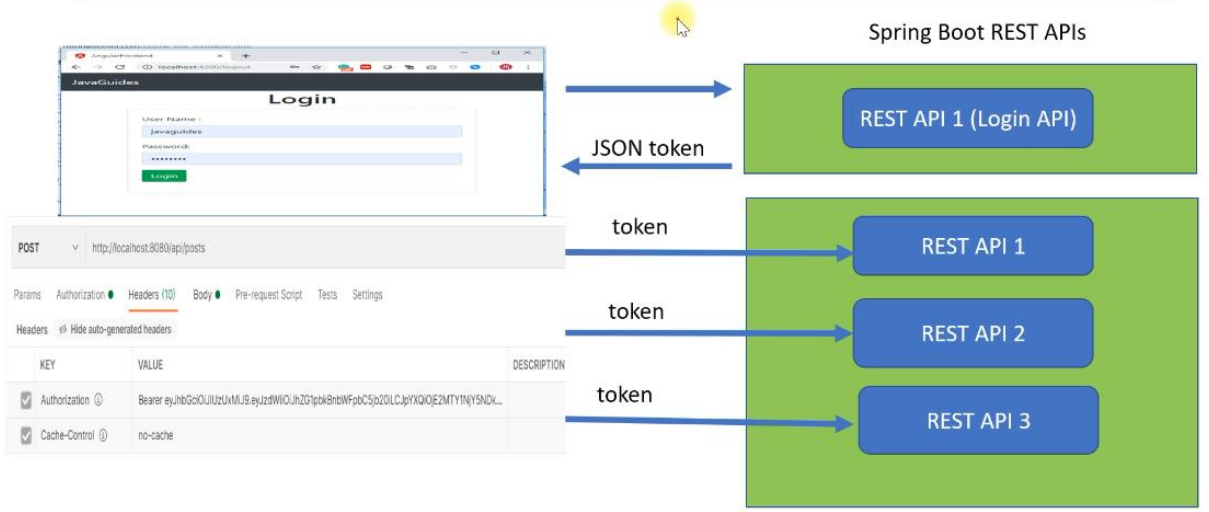


Disadvantage of Basic Auth - we have to pass hard coded username and passoword in header of each and every request.

Spring Security Basic Auth



How JWT work in spring security – client will call the login API and it will pass username and password. Server will process the request and if username & password are valid, then server will basically generate a JWT token and it will send as a part of response of login API to the client. So once client get JWT token from the server, then client should pass this JWT token in a header of each and every subsequent request.



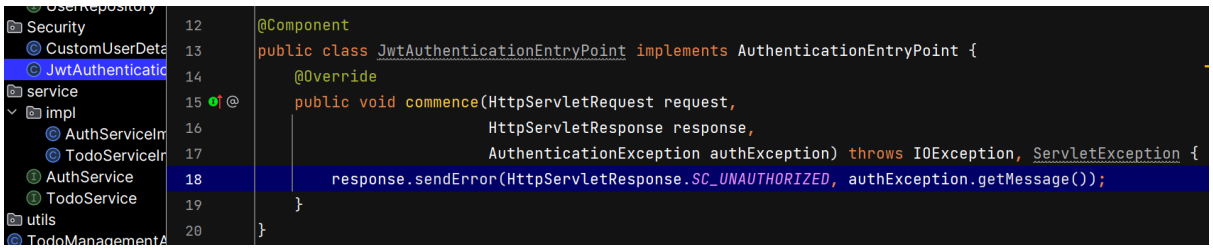
Development Steps

1. Add JWT related maven dependencies
2. Create **JwtAuthenticationEntryPoint** class
3. Add JWT properties in **application.properties** file
4. Create JwtTokenProvider class - Utility class
5. Create JwtAuthenticationFilter
6. Create JwtAuthResponse - DTO class
7. Configure JWT in Spring Security
8. Change Login REST API to return JWT Token

Add 3 maven dependencies – jjwt-api, jackson and impl in pom.xml file.

Commence() method is called whenever an exception is thrown due to an unauthorized user trying to access a resource that require an authentication.

We saved secret key using SHA 256 algorithm.



application.properties

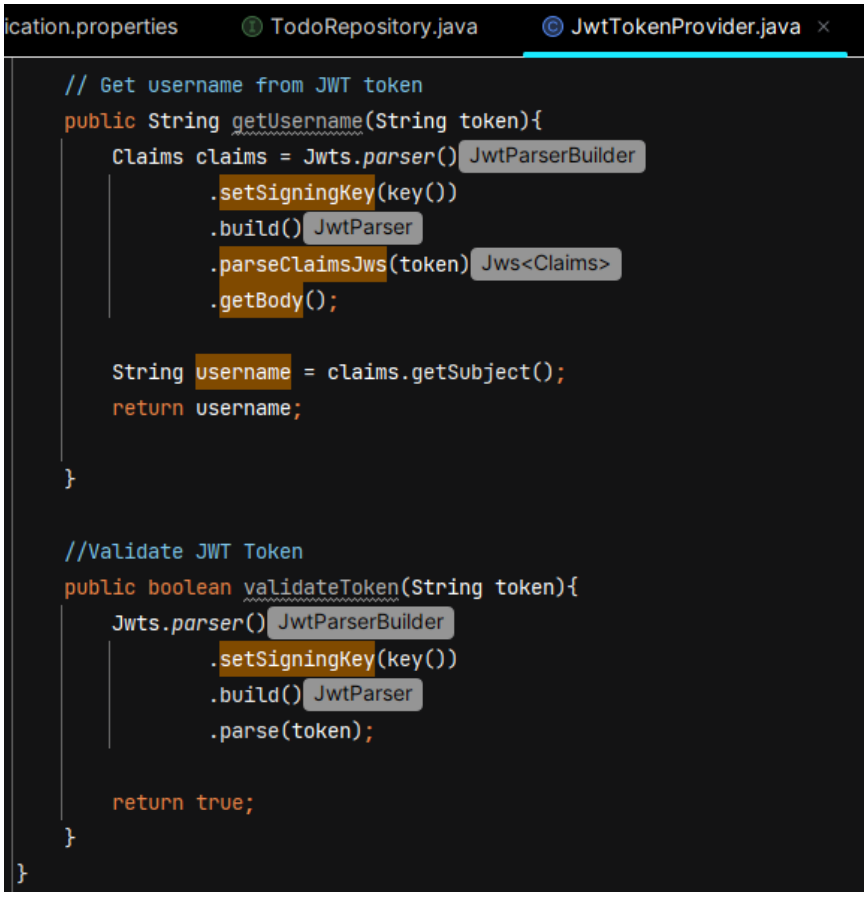
```
app.jwt-secret=fa4f7155e110eac164bda0f36b5d43f01d5d43d6afa190a317c5bec463f85764
app.jwt-expiration-milliseconds=864000000
```

Creating JwtTokenProvider Utility class

@Value annotation is used to inject value into field, method parameter or constructor argument. We can give any literal value or inject properties from property file.

JwtTokenProvider.java | Date.java | application.properties | JwtAuthenticatio

```java
@Component
public class JwtTokenProvider {
    @Value("${app.jwt-secret}")
    private String jwtSecret;

    @Value("${app.jwt-expiration-milliseconds}")
    private long jwtExpirationDate;

    // Generate JWT token
    public String generateToken(Authentication authentication){
        String username = authentication.getName();
        Date currentDate = new Date();
        Date expireDate = new Date(currentDate.getTime() + jwtExpirationDate);

        String token = Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(expireDate)
                .signWith(key())
                .compact();
        return token;
    }

    private Key key(){
        return Keys.hmacShaKeyFor(
                Decoders.BASE64.decode(jwtSecret)
        );
    }
```



Creating JwtAuthenticationFilter

OncePerRequestFilter is a base class provided to ensure that a filter is executed only once per request.

In request header, we'll get a key as Authorization and value as "Bearer <TOKEN_VALUE>"

JwtAuthenticationFilter will execute before executing spring security filters. It will validate the JWT token and provides user details to Spring security for Authentication.

TodoRepository.java | JwtTokenProvider.java | JwtAuthenticationFilter.java | AuthServiceImpl.java

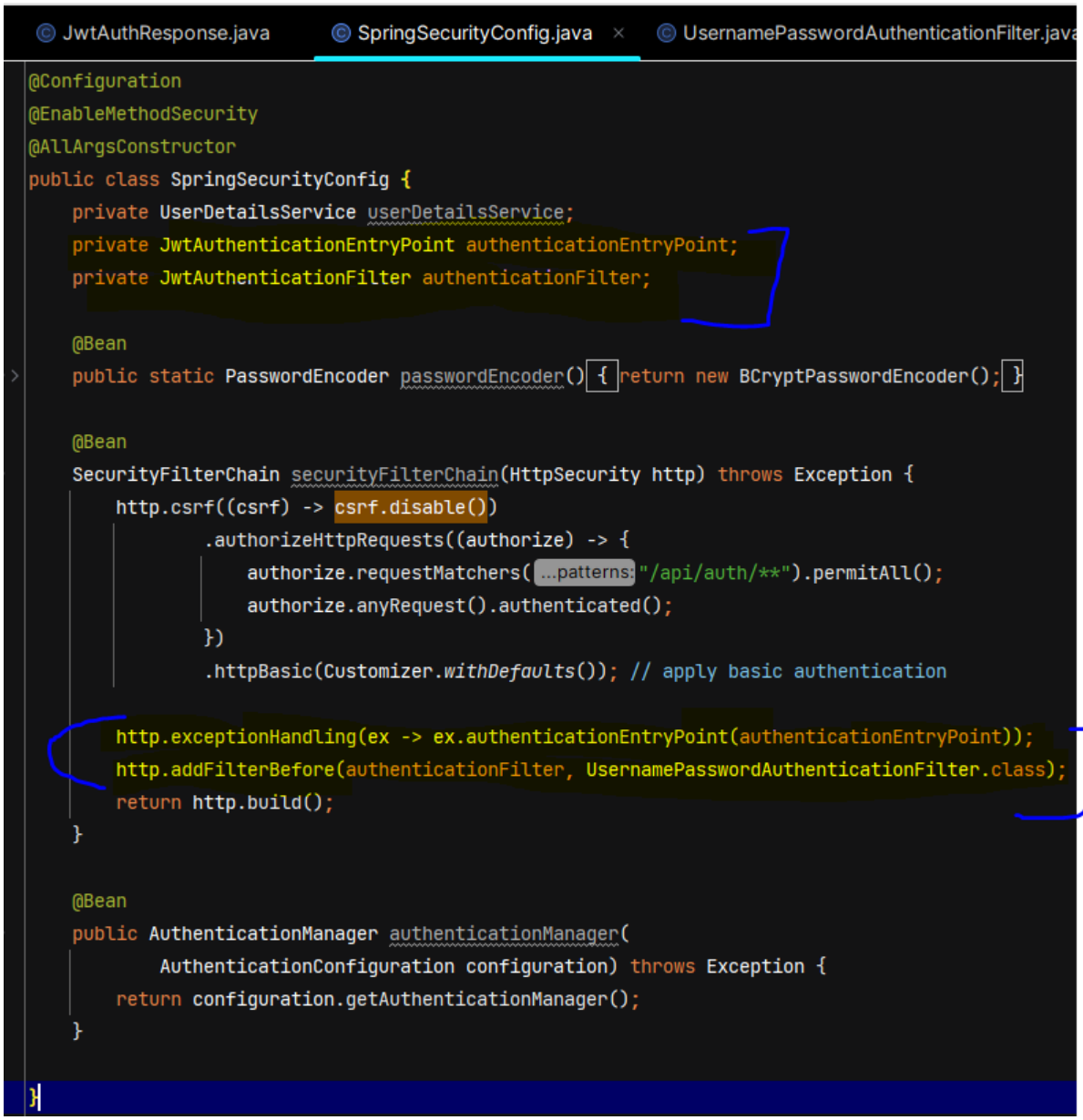
```java
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    private JwtTokenProvider jwtTokenProvider;
    private UserDetailsService userDetailsService;

    public JwtAuthenticationFilter(JwtTokenProvider jwtTokenProvider,
                                   UserDetailsService userDetailsService) {
        this.jwtTokenProvider = jwtTokenProvider;
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {
        // Get JWT token from HTTP request
        String token = getTokenFromRequest(request);

        // Validate Token
        if(StringUtils.hasText(token) && jwtTokenProvider.validateToken(token)){
            // get username from token
            String username = jwtTokenProvider.getUsername(token);
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);

            UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(
                    userDetails,
                    credentials: null,
                    userDetails.getAuthorities()
            );
            authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

            SecurityContextHolder.getContext().setAuthentication(authenticationToken);
        }

        filterChain.doFilter(request, response);
    }
```

TodoRepository.java | JwtTokenProvider.java | JwtAuthenticationFilter.java

```java
    }
    private String getTokenFromRequest(HttpServletRequest request){
        String bearerToken = request.getHeader( s: "Authorization");
        if(StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")){
            return bearerToken.substring(7, bearerToken.length());
        }

        return null;
    }
}
```

Configure JWT in Spring Security

Whenever unauthenticated user tried to access the resource, then spring security throws authentication exception and JwtAuthenticationEntryPoint class will catch that exception and return the error response to the client.



Change Login REST API to return JWT Token

nePasswordAuthenticationFilter.jav | AuthServiceImpl.java | StringUtils.class

```java
@Service
@AllArgsConstructor
public class AuthServiceImpl implements AuthService {
    private UserRepository userRepository;
    private RoleRepository roleRepository;
    private PasswordEncoder passwordEncoder;
    private AuthenticationManager authenticationManager;
    private JwtTokenProvider jwtTokenProvider;

    @Override
    public String login(LoginDto loginDto){
        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                loginDto.getUsernameOrEmail(),
                loginDto.getPassword()
        ));

        // store authentication object in security context holder
        SecurityContextHolder.getContext().setAuthentication(authentication);

        String token = jwtTokenProvider.generateToken(authentication);
        return token;
    }
```

Change login REST API to return Role along with JWT Token

Add role in JwtAuthResponse -

```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class JwtAuthResponse {
    private String accessToken;
    private String tokenType = "Bearer";
    private String role;
}
```

Change return type of login() method of AuthService to JwtAuthResponse from String –

```java
public interface AuthService {
    String register(RegisterDto registerDto);
    JwtAuthResponse login(LoginDto loginDto);
}
```

TokenProvider.java | JwtAuthResponse.java | AuthServiceImpl.java | AuthController.java | AuthService.java

```java
@Service
@AllArgsConstructor
public class AuthServiceImpl implements AuthService {
    private UserRepository userRepository;
    private RoleRepository roleRepository;
    private PasswordEncoder passwordEncoder;
    private AuthenticationManager authenticationManager;
    private JwtTokenProvider jwtTokenProvider;

    @Override
    public JwtAuthResponse login(LoginDto loginDto){
        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                loginDto.getUsernameOrEmail(),
                loginDto.getPassword()
        ));

        // store authentication object in security context holder
        SecurityContextHolder.getContext().setAuthentication(authentication);

        String token = jwtTokenProvider.generateToken(authentication);

       String role = null;
        Optional<User> userOptional = userRepository.findByUsernameOrEmail(loginDto.getUsernameOrEmail(), loginDto.getUsernameOrEmail());
        if(userOptional.isPresent()){
            User loggedInUser = userOptional.get();
            Optional<Role> optionalRole = loggedInUser.getRoles().stream().findFirst();

            if(optionalRole.isPresent()){
                Role userRole = optionalRole.get();
                role = userRole.getName();
            }
        }

        JwtAuthResponse jwtAuthResponse = new JwtAuthResponse();
        jwtAuthResponse.setAccessToken(token);
        jwtAuthResponse.setRole(role);
        return jwtAuthResponse;
    }
```

oject > todo_management > service > impl > AuthServiceImpl > register

TokenProvider.java | JwtAuthResponse.java | AuthServiceImpl.java | AuthController.java

```java
@RestController
@RequestMapping("/api/auth")
@AllArgsConstructor
public class AuthController {
    private AuthService authService;

    // Build Login REST API
    @PostMapping("/login")
    public ResponseEntity<JwtAuthResponse> login(@RequestBody  LoginDto loginDto){
        JwtAuthResponse jwtAuthResponse = authService.login(loginDto);
        return new ResponseEntity<>(jwtAuthResponse, HttpStatus.OK);
    }
```