**Document Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Ver.** | **Date** | **Changed by** | **Modifications** |
| 1 | Aug 10 2021 | Hoang Dang Khoa (RBVH/ETM2) | Initial |
| 2 | Sep 27 2021 | Hoang Dang Khoa (RBVH/ETM2) | First training document |
| 2 | Oct 4 2021 | Hoang Dang Khoa (RBVH/ETM2) | Update JWT authentication sequence diagram |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[Exercise 1: Bootstrap First Spring Boot Application 2](#_Toc80721263)

[Exercise 2: Creating our first REST application 3](#_Toc80721264)

[Exercise 3: Create a Thymleaf application with Spring Boot 3](#_Toc80721265)

[Exercise 4: Use application properties to connect DB 4](#_Toc80721266)

[Exercise 5: Adding Spring Security 5](#_Toc80721267)

[Exercise 6: Authentication Using JWT with Spring Security 6](#_Toc80721268)

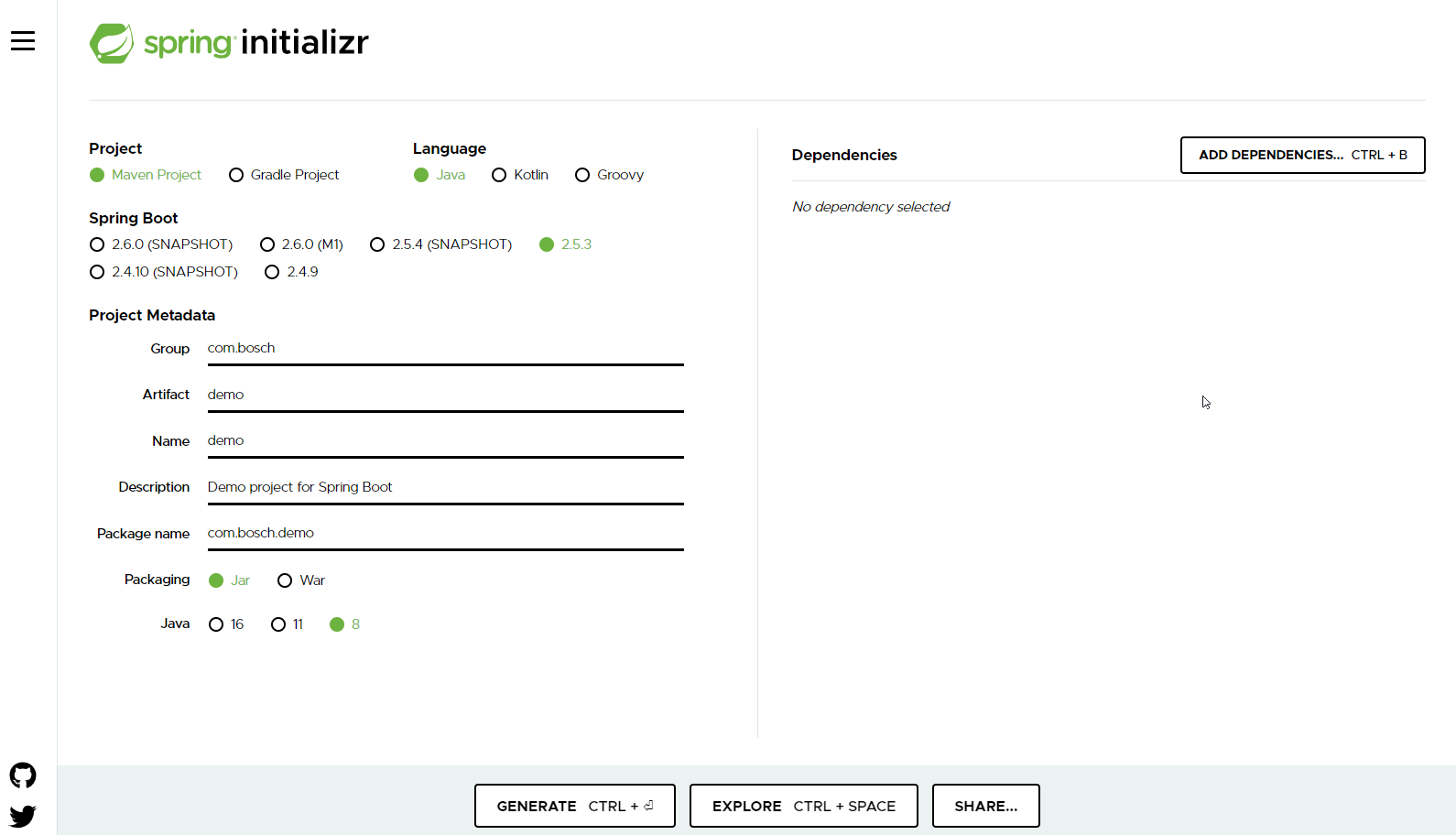
[Exercise 7: Build Simple CRUD REST with Spring Data JPA 13](#_Toc80721269)

[Exercise 8: Use @OneToMany & @ManyToOne Mapping 18](#_Toc80721270)

[Exercise 9: Custom Queries 19](#_Toc80721271)

### Bootstrap First Spring Boot Application

1. Go to <https://start.spring.io/>
2. Filling information:



And click Generate to download source code.

1. After download success, open IntelliJ IDE, create new project and import downloaded source code to your project. If this is the first time you import Spring Boot project, please sync maven to get all libraries.
2. Open Application.java and modify this file as below:

@SpringBootApplication  
public class Application implements ApplicationRunner {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(Application.class, args);  
 }  
  
 @Override  
 public void run(ApplicationArguments arg0) {  
 System.*out*.println("Hello World from Application Runner");  
 }  
}

1. Finally run Spring Boot and check whether this log message is appear in the console log.

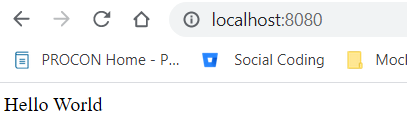
“Hello World from Application Runner”

### Creating our first REST application

1. Add following dependency to POM file:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>

1. Write REST end point
2. Run and testing with Browser by URL: <http://localhost:8080/>



### Create a Thymleaf application with Spring Boot

1. Add following dependency to POM file:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
</dependency>

1. Add Controller

@Controller  
public class WebController {  
 @GetMapping(value = "/index")  
 public String index() {  
 return "index";  
 }  
}

1. Add Web Resources

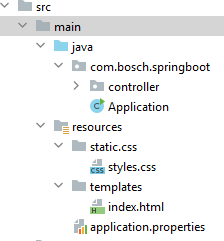
index.html

<!DOCTYPE html>  
<html>  
<head>  
 <meta charset = "ISO-8859-1" />  
 <link href = "css/styles.css" rel = "stylesheet"/>  
 <title>Spring Boot Application</title>  
</head>  
<body>  
<h4>Welcome to Thymeleaf Spring Boot web application</h4>  
</body>  
</html>

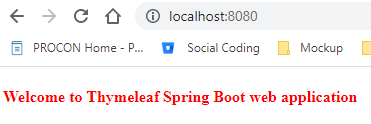
style.css

h4 {  
 color: red;  
}

Following this structure:



To get final result:



### Use application properties to connect DB

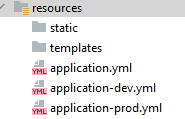
In this exercise, we will apply profile to create two database connection options. One “dev” profile will connect to H2 DB and one “prod” profile will connect to Mysql DB.

1. Add following dependencies to your POM file:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

*<!-- MySQL -->*<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
</dependency>  
  
*<!-- in-memory database -->*<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
</dependency>

1. Add three application properties files in resources folder:



1. Add below configuration to application.yml file:

spring:  
 profiles:  
 active: "dev"  
 jpa:  
 hibernate.ddl-auto: update  
 generate-ddl: false  
 show-sql: true

1. Add below configuration to application-dev.yml file, update DB connection as your local MySql setup:

spring:  
 datasource:  
 url: jdbc:h2:mem:db;DB\_CLOSE\_DELAY=-1  
 username: sa  
 password: sa

1. Add below configuration to application-prod.yml file, update DB connection as your local MySql setup:

spring:  
 datasource:  
 driverClassName: com.mysql.cj.jdbc.Driver  
 url: jdbc:mysql://localhost:3306/test  
 username: root  
 password: 1234

1. Finally, run your application. By default, dev profile will be started, and you can see below logs:

The following profiles are active: dev

HHH000400: Using dialect: org.hibernate.dialect.H2Dialect

But, if you change the default profile to “prod” in application.yml, then you could see below logs:

HHH000400: Using dialect: org.hibernate.dialect.MySQL8Dialect

### Adding Spring Security

In this exercise, we will try to add Spring Security and test simple security feature

1. Add following dependencies to your POM file

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>

1. After adding dependency, start your application, you could see a similar log that show the password:

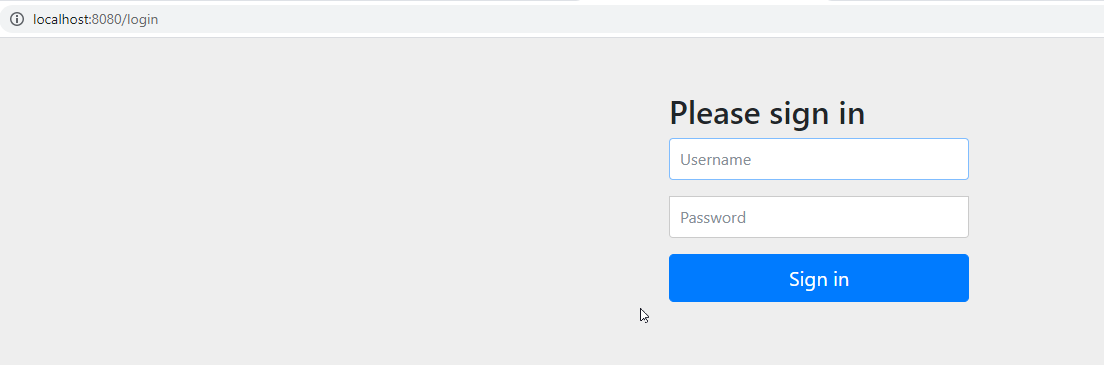
Using generated security password: 1fc15145-dfee-4bec-a009-e32ca21c77ce

This is default behavior because the Spring Security framework requires authentication out of the box for all URLs.

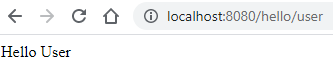
1. Now let’s play around this feature by adding an simple rest End-point:

@RestController  
@RequestMapping("hello")  
public class HelloController {  
  
 @GetMapping("user")  
 public String helloUser() {  
 return "Hello User";  
 }  
  
 @GetMapping("admin")  
 public String helloAdmin() {  
 return "Hello Admin";  
 }  
}

1. Then restart application and try to access localhost:8080/hello/user , you will be navigate to a login page



1. Then try to login the password in console and “user” as username, then you will be navigated to the previous url /hello/user.



You could logout and try again by go to this url: http://localhost:8080/logout

### Authentication Using JWT with Spring Security

In this exercise, continuously with exercise 5, we will implement basic authentication using JWT with Spring Security.

Sequence diagram of generating JWT:



Sequence diagram of validating JWT:



Authentication manager:



1. Add follow dependency:

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt</artifactId>  
 <version>0.9.1</version>  
</dependency>

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

@Component  
public class JwtTokenUtil implements Serializable {  
  
 private static final long *serialVersionUID* = -2550185165626007488L;  
  
 public static final long *JWT\_TOKEN\_VALIDITY* = 5 \* 60 \* 60;  
  
 private String secret = "zdtlD3JK56m6wTTgsNFhqzjqP";  
  
 *//retrieve username from jwt token* public String getUsernameFromToken(String token) {  
 return getClaimFromToken(token, Claims::getSubject);  
 }  
  
 *//retrieve expiration date from jwt token* public Date getExpirationDateFromToken(String token) {  
 return getClaimFromToken(token, Claims::getExpiration);  
 }  
  
 public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {  
 final Claims claims = getAllClaimsFromToken(token);  
 return claimsResolver.apply(claims);  
 }  
 *//for retrieveing any information from token we will need the secret key* private Claims getAllClaimsFromToken(String token) {  
 return Jwts.*parser*().setSigningKey(secret).parseClaimsJws(token).getBody();  
 }  
  
 *//check if the token has expired* private Boolean isTokenExpired(String token) {  
 final Date expiration = getExpirationDateFromToken(token);  
 return expiration.before(new Date());  
 }  
  
 *//generate token for user* public String generateToken(UserDetails userDetails) {  
 Map<String, Object> claims = new HashMap<>();  
 return doGenerateToken(claims, userDetails.getUsername());  
 }  
  
 *//while creating the token -  
 //1. Define claims of the token, like Issuer, Expiration, Subject, and the ID  
 //2. Sign the JWT using the HS512 algorithm and secret key.  
 //3. According to JWS Compact Serialization(https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41#section-3.1)  
 // compaction of the JWT to a URL-safe string* private String doGenerateToken(Map<String, Object> claims, String subject) {  
  
 return Jwts.*builder*().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.*currentTimeMillis*()))  
 .setExpiration(new Date(System.*currentTimeMillis*() + *JWT\_TOKEN\_VALIDITY* \* 1000))  
 .signWith(SignatureAlgorithm.*HS512*, secret).compact();  
 }  
  
 *//validate token* public Boolean validateToken(String token, UserDetails userDetails) {  
 final String username = getUsernameFromToken(token);  
 return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));  
 }  
}

1. Add JWTUserDetailsService class, this class 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.](https://www.javainuse.com/onlineBcrypt)

@Service  
public class JwtUserDetailsService implements UserDetailsService {  
  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 if ("foolishuser".equals(username)) {  
 return new User("foolishuser", "$2a$10$7pXT3Q6xnONfSdBNlxvqk.j4lc6dGi.TuSxRez7ddia6PyRKBn3TK",  
 new ArrayList<>());  
 } else {  
 throw new UsernameNotFoundException("User not found with username: " + username);  
 }  
 }  
}

1. Add JwtAuthenticationController class, 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.

@RestController  
@CrossOrigin  
public class JwtAuthenticationController {  
  
 @Autowired  
 private AuthenticationManager authenticationManager;  
  
 @Autowired  
 private JwtTokenUtil jwtTokenUtil;  
  
 @Autowired  
 private JwtUserDetailsService userDetailsService;  
  
 @PostMapping(value = "/authenticate")  
 public ResponseEntity<?> createAuthenticationToken(@RequestBody JwtRequest authenticationRequest) throws Exception {  
  
 authenticate(authenticationRequest.getUsername(), authenticationRequest.getPassword());  
  
 final UserDetails userDetails = userDetailsService  
 .loadUserByUsername(authenticationRequest.getUsername());  
  
 final String token = jwtTokenUtil.generateToken(userDetails);  
  
 return ResponseEntity.*ok*(new JwtResponse(token));  
 }  
  
 private void authenticate(String username, String password) throws Exception {  
 try {  
 authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));  
 } catch (DisabledException e) {  
 throw new Exception("USER\_DISABLED", e);  
 } catch (BadCredentialsException e) {  
 throw new Exception("INVALID\_CREDENTIALS", e);  
 }  
 }  
}

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

public class JwtRequest implements Serializable {  
  
 private static final long *serialVersionUID* = 5926468583005150707L;  
  
 private String username;  
 private String password;  
  
 *//need default constructor for JSON Parsing* public JwtRequest()  
 {  
  
 }  
  
 public JwtRequest(String username, String password) {  
 this.setUsername(username);  
 this.setPassword(password);  
 }  
  
 public String getUsername() {  
 return this.username;  
 }  
  
 public void setUsername(String username) {  
 this.username = username;  
 }  
  
 public String getPassword() {  
 return this.password;  
 }  
  
 public void setPassword(String password) {  
 this.password = password;  
 }  
}

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

public class JwtResponse implements Serializable {  
  
 private static final long *serialVersionUID* = -8091879091924046844L;  
 private final String jwttoken;  
  
 public JwtResponse(String jwttoken) {  
 this.jwttoken = jwttoken;  
 }  
  
 public String getToken() {  
 return this.jwttoken;  
 }  
}

1. Add JwtRequestFilter, this class 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.

@Component  
public class JwtRequestFilter extends OncePerRequestFilter {  
  
 @Autowired  
 private JwtUserDetailsService jwtUserDetailsService;  
  
 @Autowired  
 private JwtTokenUtil jwtTokenUtil;  
  
 @Override  
 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)  
 throws ServletException, IOException {  
  
 final String requestTokenHeader = request.getHeader("Authorization");  
  
 String username = null;  
 String jwtToken = null;  
 *// JWT Token is in the form "Bearer token". Remove Bearer word and get  
 // only the Token* if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {  
 jwtToken = requestTokenHeader.substring(7);  
 try {  
 username = jwtTokenUtil.getUsernameFromToken(jwtToken);  
 } catch (IllegalArgumentException e) {  
 logger.error("Unable to get JWT Token");  
 } catch (ExpiredJwtException e) {  
 logger.error("JWT Token has expired");  
 }  
 } else {  
 logger.warn("JWT Token does not begin with Bearer String");  
 }  
  
 *// Once we get the token validate it.* if (username != null && SecurityContextHolder.*getContext*().getAuthentication() == null) {  
  
 UserDetails userDetails = this.jwtUserDetailsService.loadUserByUsername(username);  
  
 *// if token is valid configure Spring Security to manually set  
 // authentication* if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {  
  
 UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(  
 userDetails, null, userDetails.getAuthorities());  
 usernamePasswordAuthenticationToken  
 .setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
 *// After setting the Authentication in the context, we specify  
 // that the current user is authenticated. So it passes the  
 // Spring Security Configurations successfully.* SecurityContextHolder.*getContext*().setAuthentication(usernamePasswordAuthenticationToken);  
 }  
 }  
 chain.doFilter(request, response);  
 }  
}

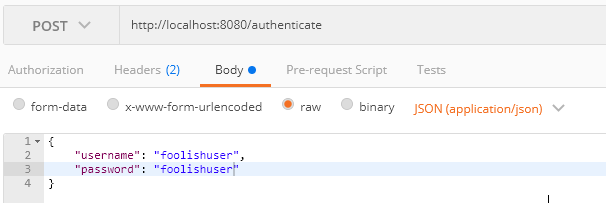
1. Add JwtAuthenticationEntryPoint, this class will extend Spring's AuthenticationEntryPoint class and override its method commence. It rejects every unauthenticated request and send error code 401

@Component  
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint, Serializable {  
  
 private static final long *serialVersionUID* = -7858869558953243875L;  
  
 @Override  
 public void commence(HttpServletRequest request, HttpServletResponse response,  
 AuthenticationException authException) throws IOException {  
  
 response.sendError(HttpServletResponse.*SC\_UNAUTHORIZED*, "Unauthorized");  
 }  
}

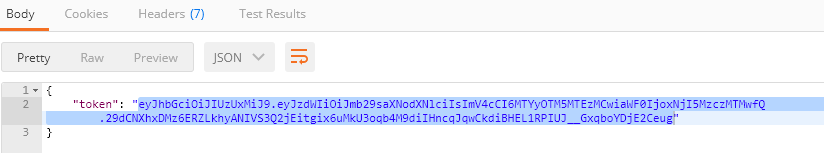
1. Add WebSecurityConfig, this class extends the WebSecurityConfigurerAdapter is a convenience class that allows customization to both WebSecurity and HttpSecurity.

@Configuration  
@EnableWebSecurity  
@EnableGlobalMethodSecurity(prePostEnabled = true)  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
  
 @Autowired  
 private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;  
  
 @Autowired  
 private UserDetailsService jwtUserDetailsService;  
  
 @Autowired  
 private JwtRequestFilter jwtRequestFilter;  
  
 @Autowired  
 public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
 *// configure AuthenticationManager so that it knows from where to load  
 // user for matching credentials  
 // Use BCryptPasswordEncoder* auth.userDetailsService(jwtUserDetailsService).passwordEncoder(passwordEncoder());  
 }  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
  
 @Bean  
 @Override  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 }  
  
 @Override  
 public void configure(WebSecurity web) throws Exception {  
 web.ignoring().antMatchers("/authenticate");  
 }  
  
 @Override  
 protected void configure(HttpSecurity httpSecurity) throws Exception {  
 *// We don't need CSRF for this example* httpSecurity.csrf().disable()  
  
 .authorizeRequests()  
 *// dont authenticate this particular request* .antMatchers("/authenticate").permitAll()  
 *// all other requests need to be authenticated* .anyRequest().authenticated()  
 .and()  
 *// make sure we use stateless session; session won't be used to  
 // store user's state.* .exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint)  
 .and().sessionManagement()  
 .sessionCreationPolicy(SessionCreationPolicy.*STATELESS*);  
  
 *// Add a filter to validate the tokens with every request* httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);  
 }  
}

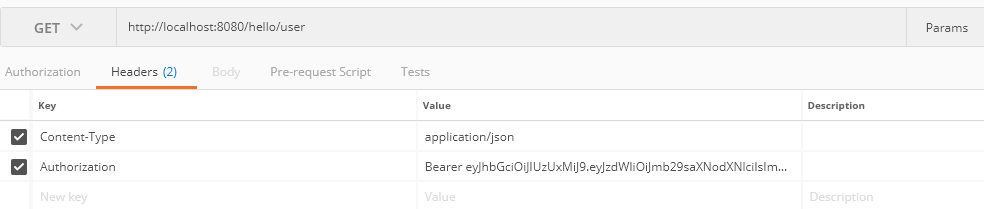
1. Start the Spring Boot Application and generate JWT token by creating a post request (username: foolishuser, password: foolishuser):



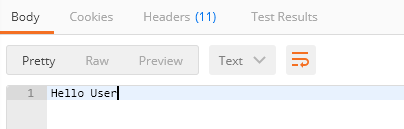
Then, application will return a token:



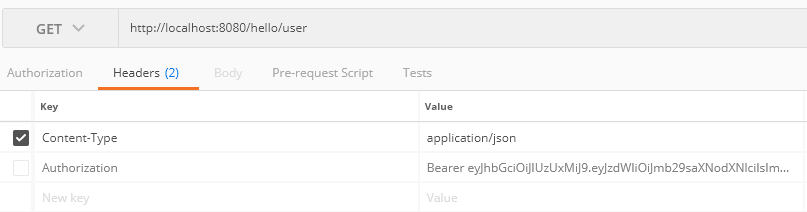
1. Validate token by accessing this url localhost:8080/hello/user and attach generated token like following picture:



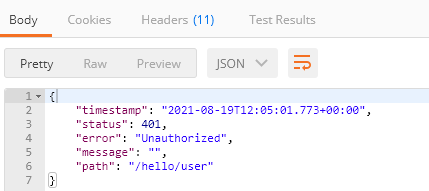
And the result is:



If you disable Authorization option:



Then we will receive unauthorized exception:



### Build Simple CRUD REST with Spring Data JPA

1. Add following dependency to POM file:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

1. Insert todos table to your DB

CREATE TABLE `todos` (

`id` INT NOT NULL AUTO\_INCREMENT,

`title` VARCHAR(45) NULL,

`description` VARCHAR(45) NULL,

`creationTime` DATETIME NOT NULL DEFAULT NOW(),

`modificationTime` DATETIME NULL,

`version` INT NULL,

PRIMARY KEY (`id`));

1. Add entity todo and map with your table in DB.

@Entity  
@Table(name = "todos")  
public class Todo implements Serializable {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 @Column(name = "creation\_time", nullable = false)  
 private ZonedDateTime creationTime;  
  
 @Column(name = "description", length = 500)  
 private String description;  
  
 @Column(name = "modification\_time")  
 private ZonedDateTime modificationTime;  
  
 @Column(name = "title", nullable = false, length = 100)  
 private String title;  
  
 @Version  
 private long version;

*// Getters and Setters*

1. Create repository layer:

public interface TodoRepository extends Repository<Todo, Long> {  
  
 void deleteById(Long id);  
  
 List<Todo> findAll();  
  
 Optional<Todo> findById(Long id);  
  
 Todo save(Todo persisted);  
}

1. Create Service layer with interface and implementation class:

TodoService interface:

public interface TodoService {  
 void delete(Long id);  
  
 List<Todo> findAll();  
  
 Optional<Todo> findOne(Long id);  
  
 Todo save(Todo todo);  
}

TodoService implementation:

@Service  
public class TodoServiceImpl implements TodoService{  
  
 @Autowired  
 private TodoRepository todoRepository;

@Override

public void delete(Long id) {  
 todoRepository.deleteById(id);  
 }

@Override  
 public List<Todo> findAll() {  
 return todoRepository.findAll();  
 }

@Override  
 public Optional<Todo> findOne(Long id) {  
 return todoRepository.findById(id);  
 }

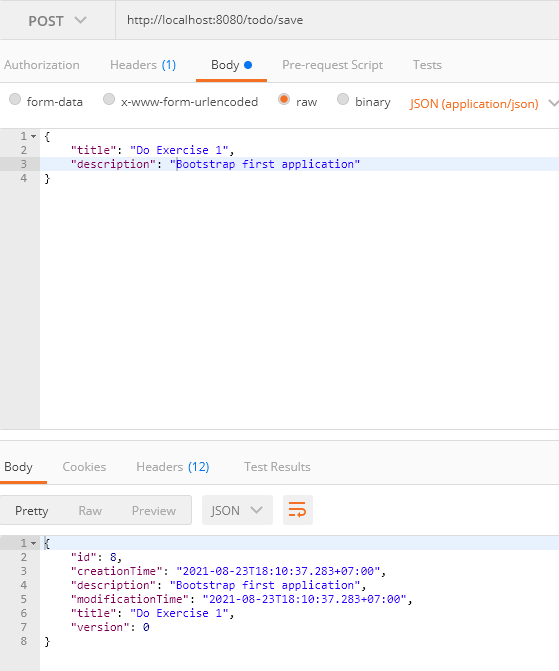
@Override  
 public Todo save(Todo todo) {  
 todo.setCreationTime(ZonedDateTime.*now*());  
 todo.setModificationTime(ZonedDateTime.*now*());  
 return todoRepository.save(todo);  
 }  
}

1. Create REST controller for todo entity:

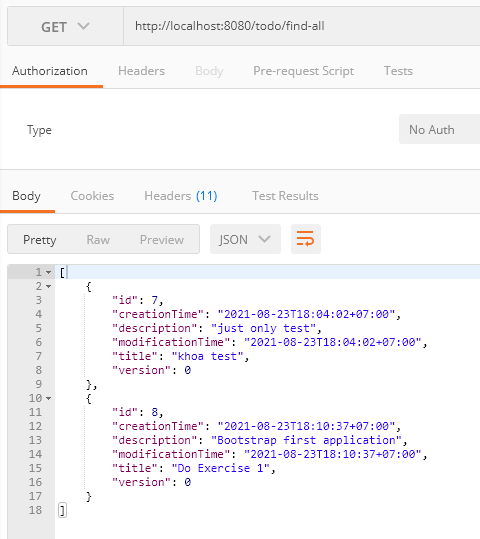
@RestController  
@RequestMapping("todo")  
public class TodoResource {  
  
 @Autowired  
 private TodoService todoService;  
  
 @GetMapping("/find-all")  
 public ResponseEntity<List<Todo>> findAll() {  
 List<Todo> result = this.todoService.findAll();  
 return ResponseEntity.*ok*().body(result);  
 }  
  
 @GetMapping("/find-one/{id}")  
 public ResponseEntity<Todo> getTodo(@PathVariable Long id) {  
 Optional<Todo> result = todoService.findOne(id);  
 return result.map(response -> ResponseEntity.*ok*().body(response))  
 .orElse(new ResponseEntity<>(HttpStatus.*NOT\_FOUND*));  
 }  
  
 @PostMapping("/save")  
 public ResponseEntity<Todo> save(@RequestBody Todo todo) throws URISyntaxException {  
 Todo result = this.todoService.save(todo);  
 return ResponseEntity.*created*(new URI("/todo/add/" + result.getId()))  
 .body(result);  
 }  
  
 @PutMapping("/update")  
 public ResponseEntity<Todo> update(@RequestBody Todo todo) throws URISyntaxException {  
 Todo result = this.todoService.save(todo);  
 return ResponseEntity.*created*(new URI("/todo/add/" + result.getId()))  
 .body(result);  
 }  
  
 @DeleteMapping("/delete/{id}")  
 public ResponseEntity<Void> deleteTodo(@PathVariable Long id) {  
 todoService.delete(id);  
 return ResponseEntity.*ok*().build();  
 }  
}

1. So we have add repository, service and resource layers to our application. Then let’s start the application and test each of APIs

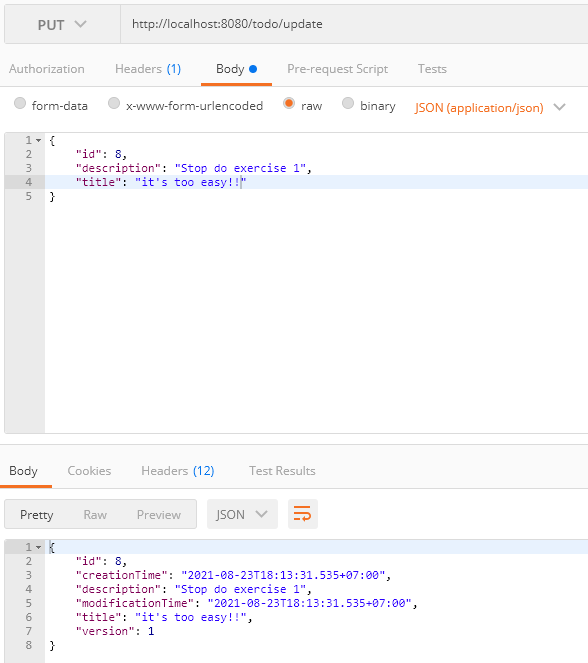
* Create a todo item:



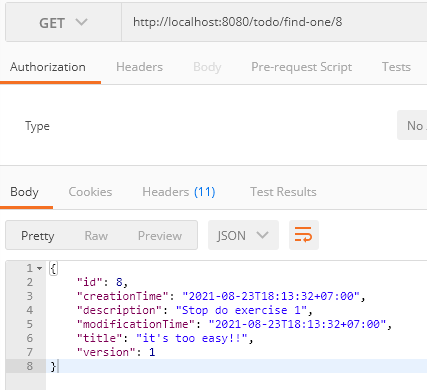
* Find all todo items:



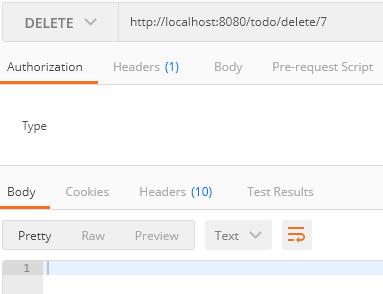
* Update your todo item:



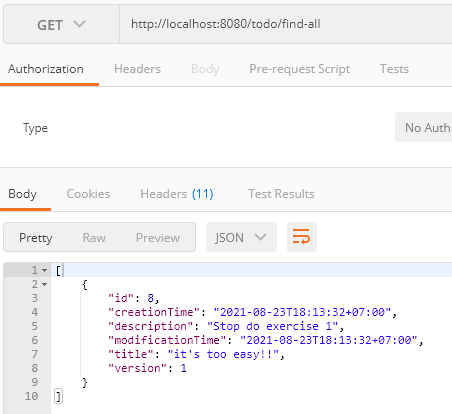
* Find one todo item:



* Delete your todo and find all again:



Now item 7 was removed if you run find all again



### Use @OneToMany & @ManyToOne Mapping

In this exercise, we will practice using @ManyToOne by adding new entity is Plan. A plan has many to do items so a plan is one-to-many relationship with todo and in the other side, a todo is many-to-one relationship with plan.

1. Add plan table and update plan\_id column as well as foreign key to todo table in database:

CREATE TABLE `plan` (

`id` INT NOT NULL AUTO\_INCREMENT,

`name` VARCHAR(100) NULL,

`description` VARCHAR(500) NULL,

`plan\_start` DATETIME NULL,

`plan\_end` DATETIME NULL,

`creation\_time` DATETIME NOT NULL DEFAULT NOW(),

`modification\_time` DATETIME NULL,

PRIMARY KEY (`id`));

ALTER TABLE `todos`

ADD COLUMN `plan\_id` INT,

ADD FOREIGN KEY fk\_planid\_todo(plan\_id) REFERENCES plan(id);

1. Add plan entity mapping with plan table:

@Entity  
@Table(name = "plan")  
public class Plan implements Serializable {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 @Column(name = "name", nullable = false, length = 100)  
 private String planName;  
  
 @Column(name = "description", length = 500)  
 private String description;  
  
 @Column(name = "plan\_start", nullable = false)  
 private ZonedDateTime planStart;  
  
 @Column(name = "plan\_end", nullable = false)  
 private ZonedDateTime planEnd;  
  
 @Column(name = "creation\_time", nullable = false)  
 private ZonedDateTime creationTime;  
  
 @Column(name = "modification\_time")  
 private ZonedDateTime modificationTime;  
  
 @OneToMany(mappedBy = "plan")  
 @JsonIgnore  
 private Set<Todo> todos = new HashSet<>();  
  
 *// Getters and Setters*

1. Try to add CRUD feature same as todos at repository, service and rest layer (do by yourself)

### Custom Queries

In this exercise, we will implement some custom queries to add these below features and make application could be a realistic usable application:

Add, edit, remove todo item from plan

List all items in a plan

List all plans together with their own items

Search flexible plans and items.