

JAVA ACADEMY - XIDERAL

MONTERREY, N.L

WEEK 5

FINAL PROJECT

Made by:

Luis Miguel Sánchez Flores

Table of Contents

INTRODUCTION.....	2
MVC Application.....	3
Model.....	3
Views (With Thymeleaf).....	4
Controller.....	9
Adding Spring Security.....	13
DEMONSTRATION.....	15
Unit testing with JUnit and Mockito.....	22
REST Application.....	29
Entities.....	29
JPA Repositories.....	33
Service.....	34
REST Controller.....	39
DEMONSTRATION.....	45

INTRODUCTION

This work serves as a culmination of the concepts and tools learned during the academy, putting into practice the foundations of the Java language, the usefulness of unit testing, among other practices for effective software development.

For this project, tools / frameworks such as Spring and Lombok were used to help streamline the development process of the application backend as a modern and effective solution that meets the established requirements.

The final project consists of two Spring projects that make up a task application that embodies the topics learned throughout the academy that allows users to perform CRUD operations on a series of tasks and keep their list organized.

The following pages of this document provide detailed information about the projects carried out, explaining the components developed, the design decisions made and the lessons learned during the development, highlighting the specific technologies and techniques employed.

MVC Application

This project applies the MVC architecture to provide the user with a website they can interact with to manage the to-do list, being able to create, update, delete tasks, with the purpose of organizing their responsibilities in a quick and easy way, while the application's backend takes care of the requests and store the data as needed.

For this MVC application, a number of tools were used to develop the program, including the **Spring Web** service to handle the user's requests, update the model's attributes and load the appropriate view page. **Spring Data JPA** was used to manage access and data manipulación towards the connected SQL database, persisting the task data with the use of repository and service layers. **Thymeleaf** is used as the application template engine to render dynamic web pages based on the user interaction with the application, displaying relevant content and giving access to other resources of the application.

Finally, unit tests were employed with the help of the **JUnit** and **Mockito** frameworks to ensure that each component of the application works as intended and verify its business logic.

Model

The Model component of the architecture represents the data and business logic of the Task application, serving as the bridge between the view (UI) and the controller (request handler). It is responsible to manage and manipulate the data, regardless of how it is presented to the user, or how it interacts with it.

For this project, the Model corresponds to a simple Task entity that is mapped to the “task” table of the SQL database connected with the use of Spring Data JPA (from now on JPA) annotations, and serves part of the data access layer to said database. For example, the **@Entity** annotation indicates to Spring that the class should be handled as a JPA entity and maps it to the corresponding table of the schema with the **@Table** annotation.

In addition to the JPA annotations, annotations that are part of the **Lombok** library are applied as well. **Lombok** is used to implement common, boilerplate code with just a simple annotation, making the developed code look much cleaner. The **@Data** annotation is used to generate the getters and setters for each of the Task fields, and the **@AllArgsConstructor** and **@NoArgsConstructor** generates a constructor that require arguments for each of the fields, and a empty constructor, respectively:

```
7 @Entity // Specify that the class is an entity
8 @Table(name = "task") // Sets the database table to which associate
9
10 // LOMBOK ANNOTATIONS
11 @Data // Generate getters, setters, toString and hashCode method, among other things...
12 @AllArgsConstructor // Sets up a constructor that requires an argument for each of the class fields
13 @NoArgsConstructor // Sets up an empty constructor
14 public class Task {
15
16     // Primary key attribute
17     @Id
18     @GeneratedValue(strategy = GenerationType.IDENTITY)
19     private int ID;
20
21     private String title;
22
23     // Specify how the enum value should be persisted (in this case, treat it as a numerical value)
24     @Enumerated(EnumType.ORDINAL)
25     private Status status;
26
27     public enum Status {
28         TODO, DONE
29     }
30 }
```

Since the Task class was specified as an entity, Spring will automatically take care of mapping the attributes with the table's column, simply based on the name of each attribute.

In the case of the **status** attribute, which is a Status enum type, JPA offers the **@Enumerated** annotation to establish how the enumerated value should be persisted into the database, either as a numerical value (ORDINAL) or a string value (STRING).

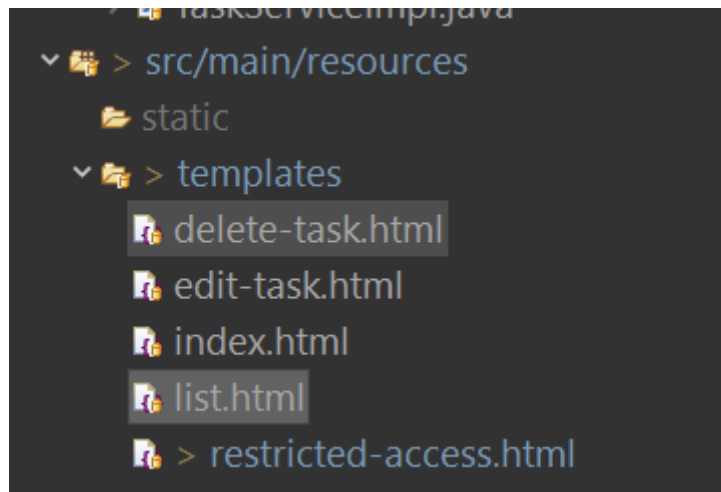
The **@Id** annotation is required by JPA to set the field which will represent the primary key of the entity / primary table. And in this case, since the primary key is assigned automatically with the AUTO_INCREMENT feature, it is made known to the JPA provider (Hibernate, for this project) by means of the **@GeneratedValue** annotation, setting up the strategy with an IDENTITY value.

Views (With Thymeleaf)

The View component of the MVC architecture is responsible for rendering static and / or dynamic web pages that presents the data towards the user, acting as a bridge between said user and the application to capture the interaction and inputs of the client to then send it off onto the Controller and display the relevant information as a response. It takes care of the presentation logic and it's independent of the underlying data (Model) and how it is processed (Controller).

In addition to being able to generate views using HTML, CSS and JavaScript, template engines can also be used as an alternative that allows the generation of dynamic web pages that update based on the user interactions. One of the most popular for Java, and the one used for this MVC project, is the **Thymeleaf** library, since it has a seamless integration with Spring, and provides an elegant and modern solution to create such templates.

Spring is auto-configured to look for Thymeleaf template files in the **resources/templates** folder of the application. Spring also checks that the file is of HTML extension and is case sensitive, so it is recommended to follow a naming convention in order to access these pages correctly:



Thymeleaf files can consist of a normal HTML structure, with the addition of the Thymeleaf XML namespace that enables the use of the library specific attributes and tags (known as *dialects*) within the HTML file that is later processed by the server.

```
1 <!DOCTYPE html>                                <!-- Enable Thymeleaf's attributes and tags -->
2 <html xmlns="http://www.w3.org/1999/xhtml"        xmlns:th="http://www.thymeleaf.org">
3 <head>
```

Such attributes shape the way in which content should be rendered onto the web page, and are prefixed with “th:” to indicate as a Thymeleaf attribute. Some common attributes include:

- **th:text** : Replaces the text of the HTML tag with the model’s data.
- **th:value** : Sets the value for a input field of the form
- **th:href** : Establishes a dynamic link based on the provided expression
- **th:if** : Conditional that renders the content if the evaluated expression is true
- **th:each** : Allows for the iteration of a list, and displays the content of each element.

- **th:object** : Specify the object to which the submitted form data will be bound.

Thymeleaf also provides special expressions to manipulate and display dynamic content onto the web page, with different types of expressions available such as:

- **Variable expressions:** Allows to access variables and/or model attributes passed from the controller
- **Link expressions:** Used to generate dynamic links to other resources of the application based on its context-path.
- **Selection expressions:** Used in conjunction with th:object to specify a property of said object.
- **Message expressions:** Apply locale text (such as from the properties file) as HTML content.

The **list.html** file is a good example that employs Thymeleaf attributes and expressions to showcase the current list of tasks for the user:

- A div tag employs the **th:each** attribute to begin iterating the list of tasks passed from the controller of the application, and for each of the tasks included:
 - **th:text** is applied to display the task title and status.
 - **th:href** is used in two hyperlinks to create a dynamic link that directs to the edit and delete handler methods of the controller based on the context-path.
- A simple form section is included to quickly generate a new task for the list. It is here that a **th:object** attribute and **selección expression** are

added to specify the template object and its properties in which the input values will be embedded in. An **th:action** provides the form's action URL where it will be submitted to.

- A simple POST form that uses a **link expression** to direct the user to the logout page if the button is clicked.

```
1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3   <head>
4     <title>Task List</title>
5   </head>
6   <body>
7
8     <!-- Simple form button to logout the current user: -->
9     <form th:action="@{/logout}" method="POST">
10       <input type="submit" value="Log out">
11     </form>
12
13     <h2>List of Tasks :</h2>
14
15     <!-- For each of the tasks included in the current list ---->
16     <div th:each="curTask : ${curList}">
17       <!-- Display the task's title and status : -->
18       <b th:text="${curTask.title}">Title</b>
19       <p th:text="${curTask.status}">Status:</p>
20
21       <!-- Create hyperlinks that directs the user to either the editing or delete page of the specific task : -->
22       <a th:href="@{/list/task/{id}(id=${curTask.ID})}">Edit Task</a>
23       <a th:href="@{/list/task/delete/{id}(id=${curTask.ID})}">Delete Task</a>
24     </div>
25
26     <h4> Add a new Task : </h4>
27
28     <!-- Simple form section that creates a new task for the list: -->
29     <form method="POST" th:object="${task}" th:action="@{/list}">
30       <input th:field="*{title}" type="text" placeholder="Enter task title"/>
31
32       <input type="submit" value="Add Task">
33     </form>
34
35   </body>
36 </html>
```

Other template files like **edit-task.html** and **delete-task.html** contains form that leverages Thymeleaf attributes and expressions to pre-populate the form elements with the details of the selected task, showcasing its ID, title and status to the users to then edit or remove the task, respectively:

edit-task.html

```
1 <!-- View for editing an existing task's attributes -->
2
3 <!DOCTYPE html>
4 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
5   <head>
6     <title>Edit Task</title>
7   </head>
8   <body>
9     <h2>Edit Task</h2>
10
11     <!-- Pre-populate the form with the existing task details : -->
12
13     <form role="form" method="POST" th:action="@{/list/task/{id}(id=${editTask.ID})}" th:object="${editTask}">
14
15       <!-- Hidden form that contains the task ID -->
16       <input type="hidden" th:field="*{ID}">
17       <label for="title">Title:</label><br>
18       <input id="title" type="text" th:field="*{title}" th:value="${editTask.title}" placeholder="Enter new title: "><br><br>
19       <label for="stat">Task Status:</label><br>
20       <select id="stat" type="checkbox" th:field="*{status}">
21         <option th:each="state : ${T(academyMty.lmsf.final_project.model.Task.Status).values()}"
22           th:value="${state}"
23           th:text="${#strings.toUpperCase(state)}"></option>
24
25       </select><br><br>
26       <input type="submit" value="Save Changes">
27     </form>
28
29
30
31   </body>
32 </html>
```

delete-task.html

```
1 <!-- View for the removal of a task from the list -->
2
3 <!DOCTYPE html>
4 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
5   <head>
6     <title>Delete Task</title>
7   </head>
8   <body>
9     <h2>Delete Task ?</h2>
10     <form role="form" th:method="delete" th:action="@{/list/task/delete/{id}(id=${delTask.ID})}" th:object="${delTask}">
11       <input type="hidden" name="_method" value="delete">
12       <h3 th:text="'Are you sure you want to delete the task with ID # ' + ${delTask.ID}">Delete task</h3><br>
13       <b>Title:</b><br>
14       <p th:text="${delTask.title}"></p><br>
15       <b>Status:</b><br>
16       <p th:text="${#strings.toUpperCase(delTask.status)}"></p><br>
17
18       <button type="submit">Delete Task</button>
19     </form>
20
21
22
23   </body>
24 </html>
```

Controller

The Controller component of the MVC architecture is considered as the central hub that handles the user input in the application, interacts with the Model component to process the data and sends out the response towards the View component, so that the application reacts with the appropriate response based on the user's interactions.

The Controller is defined by the **TaskController** class that is annotated with **@Controller** provided by Spring to handle the user's requests and interact with the Model-View components. Along with it, a **@RequestMapping** sets the base URL in which the user accesses it to begin interacting with the application controller.

The controller is injected with the service layer (**taskService**) through the **@Autowired** annotation that allows it to interact with the database.

```
18
19 @Controller
20 @RequestMapping("/list") // Base URL
21 public class TaskController {
22
23     // Inject the Service layer to interact with the database:
24     @Autowired
25     private TaskService taskService;
26
27
```

For the **addTasksToModel()** function, the **@ModelAttribute** is applied to bind objects onto the Model, and since it gets called before the corresponding request handler method, it's a good way to pre-populate the model with the current list of tasks to show in the list view, as well as a new Task object that is placed onto the simple form section to create a new one.

```
// Retrieve the list of existing tasks beforehand:
@ModelAttribute
public void addTasksToModel(Model model) {

    List<Task> allTasks = taskService.findAllTasks();

    // Add the existing task to the model:
    model.addAttribute("curList", allTasks);

    // Add a new Task object to the model, as part of
    // the simple creation form included in the page:
    model.addAttribute("task", new Task());
}
```

The request handler methods are defined with a special, shortcut Request Mapping annotation based on the type of HTTP request it should manage. For example the **@GetMapping** annotation tells that the function must be invoked when a GET request at the specific endpoint is sent:

- **getList()** : Basic GET function in charge of returning the view to display the list of tasks. It is here that the Thymeleaf view retrieves the list of tasks inserted as a Model attribute.
- **editTask()** : Retrieves a specific task by its ID (taken as a Path Variable) and adds it to the model for the “edit-task” view to modify it.
- **deleteTask()**: Retrieves a view to display the selected task to delete from the database, based on its ID

```
// return the view of list:
@GetMapping
public String getList() {
    return "list";
}

//Get the view for edit a specific task, based on it's id:
@GetMapping("/task/{id}")
public String editTask(@PathVariable("id") int id, Model model) {

    // Get the existing task by its id:
    Task taskToEdit = taskService.getTaskById(id);

    // Add the task to the model:
    model.addAttribute("editTask", taskToEdit);

    return "edit-task";
}

// Retrieve the view for deleting an existing task:
@GetMapping("/task/delete/{id}")
public String deleteTask(@PathVariable("id") int id, Model model) {

    // Get the task to be deleted:
    Task taskToDelete = taskService.getTaskById(id);

    // Add it to the model:
    model.addAttribute("delTask", taskToDelete);

    return "delete-task";
}
```

Similarly, POST and DELETE methods were developed to support other HTTP requests the user can make to either save, update or delete the task from the list. Each method redirects to the list view to update it:

```
// Save changes for an existing task:
@PostMapping("/task/{id}")
public String saveTaskChanges(@PathVariable("id") int id, @ModelAttribute("editTask") Task editTask) {

    taskService.updateTask(editTask);

    return "redirect:/list";
}

// Request to confirm the deletion of the task:
@DeleteMapping("/task/delete/{id}")
public String confirmTaskDelete(@PathVariable("id") int id) {

    taskService.removeTaskById(id);

    return "redirect:/list";
}

// Add a new Task object to the existing list:
@PostMapping
public String addNewTask(@ModelAttribute Task task) {

    task.setID(0);
    task.setStatus(Task.Status.TODO);

    taskService.addTask(task);

    return "redirect:/list";
}
```

Adding Spring Security

To limit the task list view to only authorized users, the **Spring Security** service was used, ensuring that said users are the only ones able to access and modify the tasks from the list.

Spring Security is a comprehensive authentication and authorization framework that provides the necessary tools to secure the application, managing access to certain resources based on the user's roles and permissions. Spring Security simplifies the implementation for robust security measures for Java applications.

For this project, the **SecurityConfig** class was developed as a configuration class that covers the settings on the authorized users to the application, and the security rules applied on the project's resources.

Since the list of users are stored in the SQL database, a **UserDetailsManager** function is set up as a Spring Bean that allows the framework to automatically

search and configure the data source (JDBC relational database, in this case) to be used for locating the authorized users of the Task application:

```
// Set up the data source where the list of users are stored in:
@Bean
public UserDetailsManager userDetailsManager(DataSource dataSource) {
    // Work with a JDBC-based user details manager:
    return new JdbcUserDetailsManager(dataSource);
}
```

The application resources are filtered with a **SecurityFilterChain** that executes before any business logic, so it intercepts every request sent to the controller to check if any authentication is required or perform any other security-related tasks beforehand.

In this case, the **list GET request** is restricted by means of the **authorizeHttpRequests** function, set within a Authorization Manager that checks if the user has the role of “**USER**” to be able to access the “**/list**” endpoint or any of its child directories. Meanwhile, the **index page** should be allowed to be viewed by any user, regardless of their role:

```
// Security-related settings to configure to relevant project resources:
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
        // Set up authorization for HTTP requests
        .authorizeHttpRequests(config ->

            config
                .requestMatchers("/list", "/list/**").hasRole("USER") // The list of
                .requestMatchers("/").permitAll() // Allow anyone to access the index
                .anyRequest().authenticated() // Any other requests should be authenticated
        )
    ;
}
```

Also in the filter chain, the login page (offered automatically by Spring Security) is configured to be accessed by any user, and the denied page view

is set up that is displayed in cases when the current user does not have the required permissions to access the resource. Finally, a logout page is established and enable its access for any user as well:

```
// Add the login page and enable its access to everyone:
formLogin(login ->

    login
    .permitAll()).

// Add the denied page
exceptionHandling(config ->

    config
    .accessDeniedPage("/restricted-access")

    ).

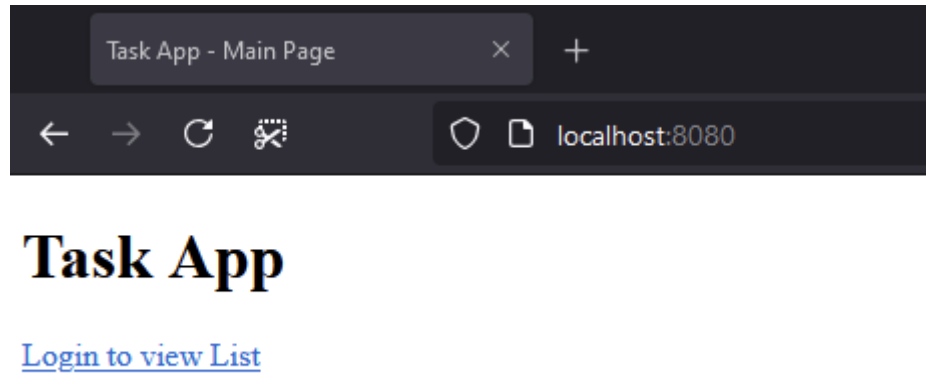
// Finally, add the logout feature to the application:
logout(logout ->

    logout
    .permitAll()
)

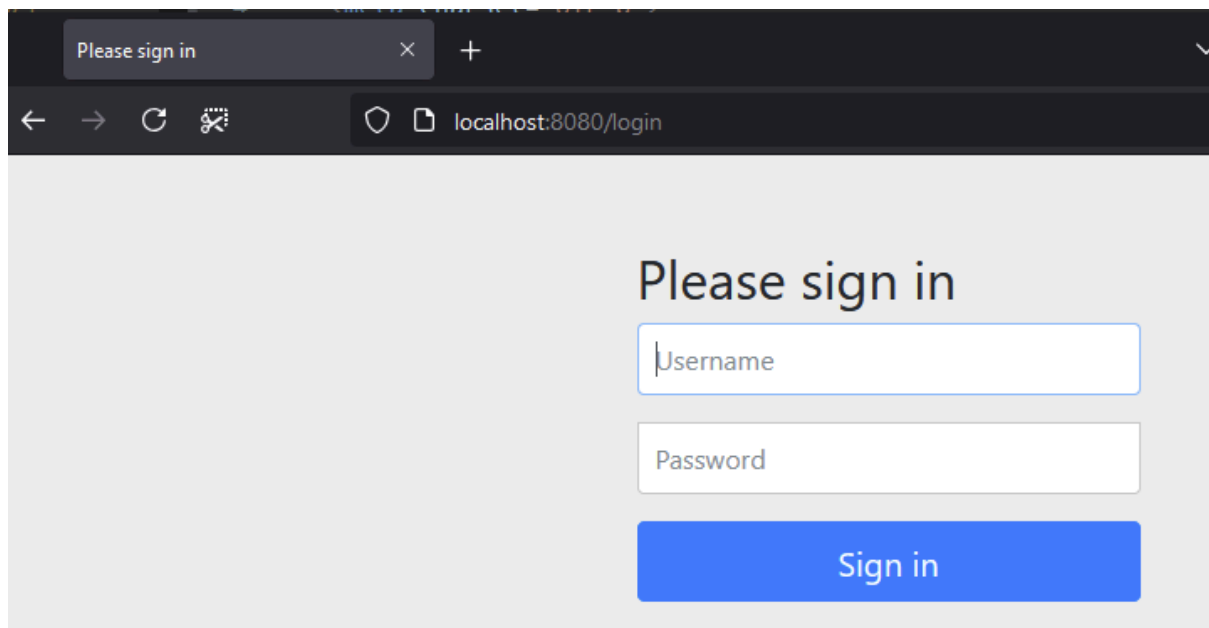
.build();
```

DEMONSTRATION

INDEX PAGE:



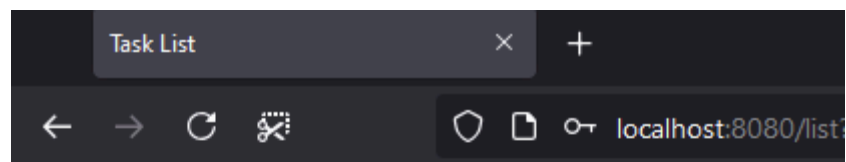
LOGIN PROMPT WHEN ACCESSING THE /LIST URL



Example User to access the list:

Result Grid			
	username	password	enabled
▶	luis	{noop}xiderallmsf	1
✱	NULL	NULL	NULL

LIST VIEW (With initial tasks):



List of Tasks :

Test 1

TODO

[Edit Task](#) [Delete Task](#)

Test 2

TODO

[Edit Task](#) [Delete Task](#)

Test 3

DONE

[Edit Task](#) [Delete Task](#)

Add a new Task :

ADDING A NEW TASK:

Add a new Task :

List of Tasks :

Test 1

TODO

[Edit Task](#) [Delete Task](#)

Test 2

TODO

[Edit Task](#) [Delete Task](#)

Move around stuff

TODO

[Edit Task](#) [Delete Task](#)

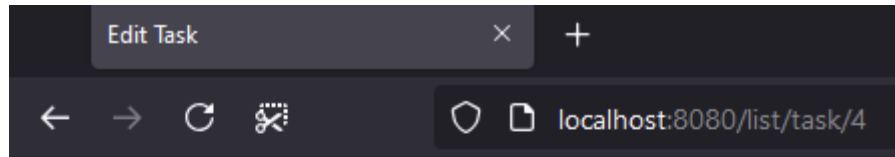
Test 3

DONE

[Edit Task](#) [Delete Task](#)

Test 4

UPDATING EXISTING TASK:



Edit Task

Title:

Task Status:

TODO ▾

Save Changes

Edit Task

Title:

Task Status:

DONE ▾

Save Changes

List of Tasks :

Test 1

TODO

[Edit Task](#) [Delete Task](#)

Test 2

TODO

[Edit Task](#) [Delete Task](#)

Test 3

DONE

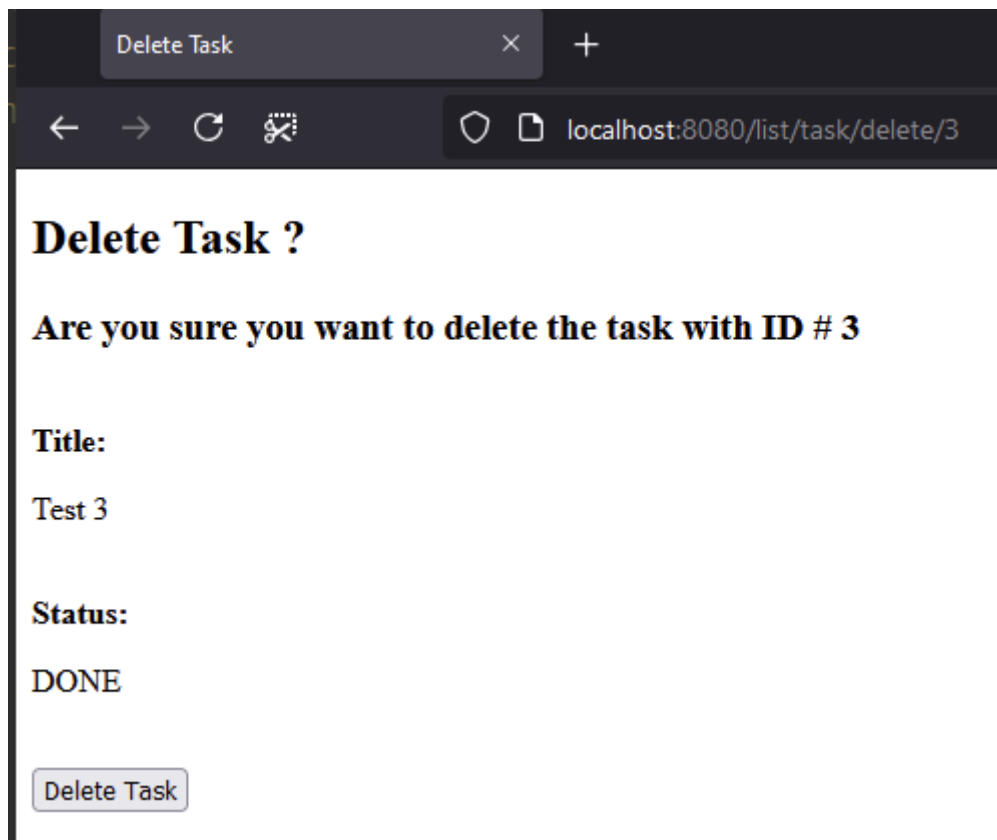
[Edit Task](#) [Delete Task](#)

Moved stuff already :)

DONE

[Edit Task](#) [Delete Task](#)

DELETING TASK:



List of Tasks :

Test 1

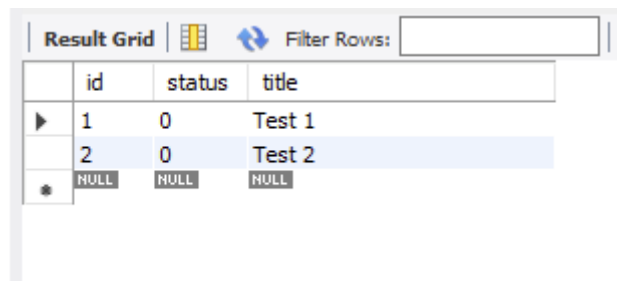
TODO

[Edit Task](#) [Delete Task](#)

Test 2

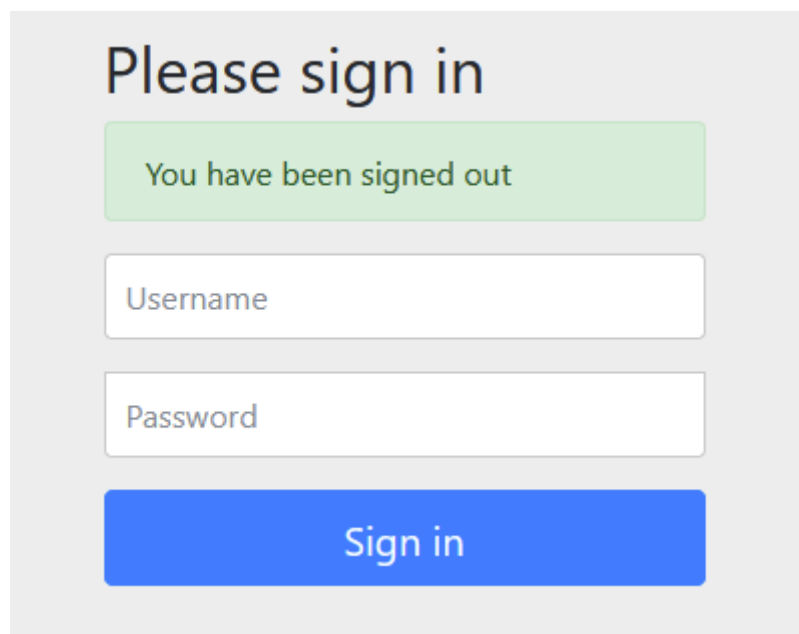
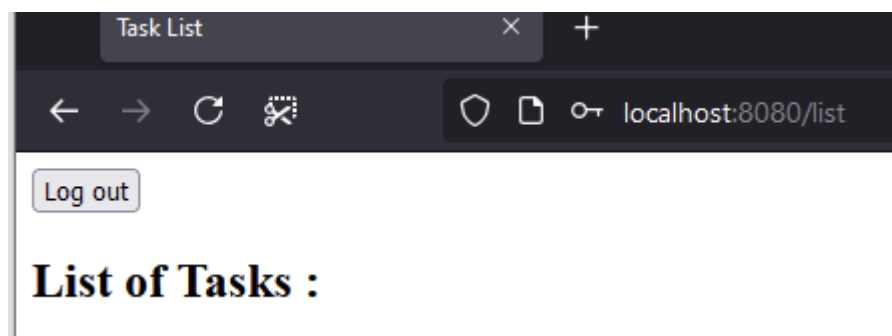
TODO

[Edit Task](#) [Delete Task](#)



	id	status	title
▶	1	0	Test 1
	2	0	Test 2
*	NULL	NULL	NULL

LOGOUT FEATURE



Please sign in

You have been signed out

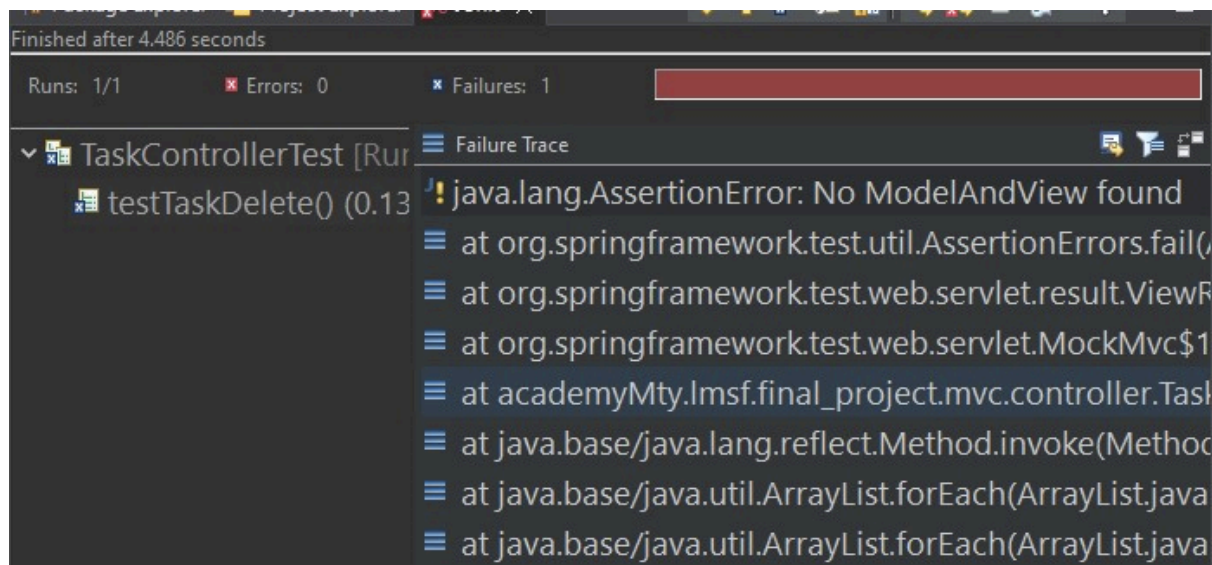
Username

Password

Sign in

Unit testing with JUnit and Mockito

Unit testing plays a fundamental role in making sure that the various components of the application works as intended and verifying the business logic required, by isolating the units of code and checking if they meet the criteria or not. Unit testing helps in identifying bugs and issues at early stages of development and can also serve as documentation with examples of how the components should behave, improving the maintainability of the program.



For Java/Spring projects, **JUnit** and **Mockito** are both popular testing frameworks worth considering thanks to its intuitive and user-friendly syntax, and offers a variety of functions to develop effective unit tests that cover most, if not all, of the program logic.

A **TaskControllerTest**, **TaskRepositoryTest** and **TaskServiceTest** were set up to test out the various aspects for each of the component's behavior, as seen in the following images.

In the **TaskServiceTest** class, the **TaskRepository** is treated as a Mock object with the **@Mock** annotation, allowing us to simulate the operations and responses of the JPA repository without the need to initialize it and know said operations beforehand, avoid complex configuration and making the tests simpler and easier to maintain:

```
class TaskServiceTest {

    // Treat TaskRepository as a Mock Object
    @Mock
    TaskRepository taskRepository;

    // Mark TaskService as an attribute in which Mock objects will be injected to
    @InjectMocks
    TaskServiceImpl taskService;

    List<Task> exampleList;

    // Before every test in the class:
    @BeforeEach
    void setUp() throws Exception {
        // Initialize the Mock objects
        MockitoAnnotations.openMocks(this);

        // Create a new list and Pre-populate it with a few tasks:
        exampleList = new ArrayList<>(List.of(

            new Task(1, "Test 1", Task.Status.TODO),
            new Task(2, "Test 2", Task.Status.DONE),
            new Task(3, "Test 3", Task.Status.TODO)

        ));
    }
}
```

```
// Test out the retrieval of a individual task by its ID;
@Test
void testGetTaskById() {
    final int ID = 1;

    Task taskToRetrieve = exampleList.get(ID-1);

    // Simulate the return of the task by its ID:
    when(taskRepository.findById(ID)).thenReturn(Optional.of(taskToRetrieve));

    Task task = taskService.getTaskById(ID);

    assertEquals(taskToRetrieve, task); // Expected: TRUE
}
```

```
// Test out the DELETE operation of an task by its ID:
@Test
void testRemoveTaskById() {

    final int ID = 2;

    // Retrieve the task from the list:
    Task taskToDelete = exampleList.get(ID-1);

    // Simulate the removal of the task from the list when deleteById() function is invoked:
    doAnswer(new Answer<Integer>() {

        @Override
        public Integer answer(InvocationOnMock invocation) throws Throwable {
            // get ID
            int id = invocation.getArgument(0);

            // Check every Task object in the list
            // and if any match the ID passed, remove it from the list:
            for(Task t : exampleList) {
                if (t.getID() == id) {
                    exampleList.remove(t);
                    break;
                }
            }
            return null;
        }
    }).when(taskRepository).deleteById(anyInt());

    // Call the service function:
    taskService.removeTaskById(ID);

    // Check if the list still contains the deleted task after the operation:
    assertFalse(exampleList.contains(taskToDelete));

    // Did it reduce its size by 2?
    assertEquals(2, exampleList.size());
}
```

```
// Test out the UPDATE operation
@Test
void testUpdateTask() {

    // Create Task object with existing ID
    final int ID = 2;
    Task oldTask = exampleList.get(ID-1);

    // New task object with same ID, but different attributes:
    Task changeTask = new Task(ID, "Change task", Task.Status.TODO);

    // Mock findById that should return the oldTask if the corresponding ID is passed:
    when(taskRepository.findById(changeTask.getID())).thenReturn(Optional.of(oldTask));

    // Mock the update to save the new changes of the existing task:
    when(taskRepository.save(changeTask)).thenReturn(changeTask);

    Task savedTask = taskService.updateTask(changeTask);

    assertNotNull(savedTask.getID());
    assertEquals("Change task", savedTask.getTitle());

    // Verify if the findById() method was called, checking beforehand that the task exists
    verify(taskRepository).findById(ID);
}
```

For the **TaskRepositoryTest** class, a **@DataJpaTest** is applied at class level to only enable the auto-configuration that is relevant to Data JPA tests. Instead of using the SQL database, an in-memory database (H2 engine) is set up as a test database in which the unit tests will be carried out.

```
20 @DataJpaTest
21 @AutoConfigureTestDatabase(connection = EmbeddedDatabaseConnection.H2)
22 class TaskRepositoryTest {
23
24     @Autowired
25     TaskRepository taskRepository;
26
27     @Autowired
28     TestEntityManager testEntityManager;
29
30     // Example task object
31     Task t1 = new Task(0, "T1", Task.Status.TODO);
32
33     // Example list of tasks
34     List<Task> exampleTasks = List.of(
35         t1,
36         new Task(0, "T2", Task.Status.DONE),
37         new Task(0, "T3", Task.Status.TODO)
38     );
39
40     // Before each test:
41     @BeforeEach
42     void setUp() {
43         // Reset the database by deleting all entities
44         taskRepository.deleteAll();
45
46         // Save all the tasks examples from the list
47         taskRepository.saveAll(exampleTasks);
48     }
49 }
```

```
// Check if the list of tasks retrieved are order by their Status:
@Test
void findTasksOrderByStatus() {

    List<Task> tasks = taskRepository.findAll();

    List<Task> orderedTasks = taskRepository.findAllOrderByStatus();

    // Are they different in terms of order?
    assertNotEquals(tasks, orderedTasks);

    // Sort the original task
    tasks.sort((t1, t2) -> t1.getStatus().compareTo(t2.getStatus()));

    assertEquals(tasks, orderedTasks);
}

// Test out the correct retrieval of an individual task:
@Test
void findTaskFound() {

    // Retrieve the ID of a particular task using the TestEntityManager:
    final int ID = testEntityManager.getId(t1, Integer.class);

    // Get the task by the retrieved ID
    Optional<Task> retrieveTask = taskRepository.findById(ID);

    // Check if the title is the expected "T1":
    assertEquals("T1", retrieveTask.get().getTitle());
}
```

In the case of the TaskControllerTest class, since it is in charge of testing the Controller component of the MVC, it contains the **@WebMvcTest** annotation that tells Spring to only apply the relevant configuration needed to test out the component. Furthermore, a **MockMvc** is injected, which allows us to make test requests to the controller, and define the expected responses, verifying the status, content and view that should be returned at the time of the request:

```
34 @WebMvcTest(TaskController.class)
35 @AutoConfigureMockMvc(addFilters = false)
36 class TaskControllerTest {
37
38     @Autowired
39     private MockMvc mockMvc;
40
41     @MockBean
42     private TaskServiceImpl taskService;
43
44     Task exampleTask;
45
46     @BeforeEach
47     void setUp() throws Exception {
48
49         exampleTask = new Task(1, "Test 1", Task.Status.TODO);
50     }
51
```

```
// Test out the list view
@Test
void testListView() throws Exception {

    List<Task> exampleTaskList = List.of(

        exampleTask,
        new Task(2, "Test 2", Task.Status.DONE),
        new Task(3, "Test 3", Task.Status.TODO)
    );

    // Mock findAllTasks() by returning the create exampleTaskList:
    when(taskService.findAllTasks()).thenReturn(exampleTaskList);

    mockMvc.perform(get("/list")) // <- perform the GET request at "/list"
        .andExpect(model().attribute("curList", exampleTaskList)) // The model must have the list as attribute
        .andExpect(view().name("list")) // Return the "list" view
        .andExpect(status().isOk()); // The status is OK

}
```

REST Application

The subsequent Spring application created as part of the final project is a RESTful version of the task application, setting up the REST API and endpoints to the application in order to adapt a common design which allows for a seamless integration with various types of clients, such as web apps, mobile apps and third-party services that can interact with the back-end functionality of the application in a flexible manner.

Similar to the MVC version of the Task application, the RESTful version leverages the Spring Web service to generate the REST endpoints and effectively handle the HTTP requests sent by the client in order to perform the necessary CRUD operations of the tasks.

Spring Data JPA was also used to streamline the database interactions and enable task data persistence, ensuring data integrity when performing operations towards the SQL database.

Entities

Entities represent the data that can be persisted onto the database with the use of a plain old Java object (POJO) with the class representing the table, the attributes as the columns of said table, and each generated instance of the class corresponds with a row. For this project, two entities are managed by the application: **User** and **Task**.

The **User** entity, as the name implies, represents the accounts that users can administer to manage a to-do list of their own. In order to define the mapping between the Java object with the relational table, JPA offers a set of annotations that specifies how the JPA provider (Hibernate, in this case) should manage the persistence of the User data.

For example, the **@Entity** annotation at class level is a core property that tells Spring to mark the User class as a database entity and become persistable during the execution of the app. Along with this, the **@Table** annotation is set to help Spring set the table to which it should be related with (in this case, the “users” table is assigned):

```
13 @Entity
14 @Table(name="users") // <- Table the entity should map to
15 // LOMBOK ANNOTATIONS
16 @Data
17 @AllArgsConstructor
18 @NoArgsConstructor
19 public class User {
```

To map the table’s columns with the object’s attributes, there’s the **@Id** and **@Column** anotación, with the former specifying that the assigned attribute should represent the **primary key** of the object, and the latter is optional, but

allows to customize the mapping, such as assigning the attribute with the corresponding column through a “**name**” value.

Along with the ID annotation, the GeneratedValue annotation specifies how the primary key is assigned. In this case, the “**strategy**” value is assigned as **IDENTITY** from the **GenerationType** enumeration to tell Spring that the column is designated automatically with the auto_increment feature to generate a unique number.

In addition to JPA annotations, the project also takes advantage of the Lombok library and its annotations to avoid writing repetitive code that Spring may require to instantiate the entity. In this case, the **@Data** annotation is used to generate the getters and setters of the attributes, while the **@AllArgsConstructor** and **@NoArgsConstructor** generates the empty and all attributes constructor, respectively:

```
12
13 @Entity
14 @Table(name="users") // <- Table the entity should map to
15 // LOMBOK ANNOTATIONS
16 @Data
17 @AllArgsConstructor
18 @NoArgsConstructor
19 public class User {
20
21     // UserID that represents the primary key of the table
22     @Id
23     @GeneratedValue(strategy = GenerationType.IDENTITY)
24     private long ID;
25
26     @Column(nullable = false, unique = true)
27     private String name;
28
29     @Column(nullable = false)
30     private String password;
31
```

Next, the **Task** entity was developed to represent the task object of the application. Unlike the User entity, Task uses a **composite key** as its primary key of the `tasks` table (both the **tId** and **uId** attributes of the class), in which it combines the task ID with the user ID that uniquely identifies each record:

```
0 @Entity
1 @Table(name = "tasks")
2 @IdClass(TaskId.class) // <- Associate the Task entity with the TaskId composite key class
3 @Data
4 @AllArgsConstructor
5 @NoArgsConstructor(access = AccessLevel.PRIVATE, force=true)
6 public class Task implements Comparable<Task> {
7
8     @Id
9     @Column(name = "task_id")
10    private int tId;
11
12    @JsonIgnore // Exclude the attribute from the JSON response of the app
13    @Id
14    @Column(name = "user_id")
15    private long uId;
16
17    // No need for the @Column annotation |
18    private String title;
19 }
```

To properly setup the composite key, an `@IDClass` annotation is used in which a **composite primary key class** is configured (TaskId), in which the following rules must be followed in order to be treated as a composite key:

- Class must be public
- Implements the Serializable interface
- Must have a constructor with no arguments
- Must define its equals() and hashCode() methods
- The associated entity must declare the same fields of the IDClass used as its ID:

With the help of Lombok, the composite key class can look as simple as the following image:

```
9
10 @Data
11 @AllArgsConstructor
12 @NoArgsConstructor
13 @EqualsAndHashCode
14 public class TaskId implements Serializable {
15
16     private int tId;
17     private long uId;
18
19
20
21 }
22
```

Lastly, a bidirectional one-to-many relationship is established between the two entities to reflect the dynamics of the real world (In this case, one user can contain one or various tasks at the same time, as in real life). To achieve this, the **@OneToMany** and **@ManyToOne** annotations are applied to specify this bidirectional relationship. In the User class, the user attribute of the Task entity is set as the owner of the relationship, and specifies the cascade operations to be applied onto the “child” entity:

```
@JsonIgnore
@JsonManagedReference
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true)
@OrderBy("task_id ASC") // Order the list of tasks by their IDs
private List<Task> tasks = new ArrayList<>();
```

On the other hand, the Task entity has a **JoinColumn** to establish the mapping of the foreign key column (“user_id”). To avoid creating a redundant column, the **insertable** and **updatable** attributes are set with a false value:

```
// Set up the relationship with the User entity|
@JsonBackReference
@ManyToOne
@JoinColumn(name = "user_id", referencedColumnName = "ID", insertable = false, updatable = false)
private User user;
```

JPA Repositories

One of the core features provided by the Spring Data JPA is a repository interface known as the **JpaRepository** that comes with built-in CRUD operations and allows to implement data access layers with ease while avoiding boilerplate code for simple database interactions.

For instance, in order to set up a repository to access the data of the User entities from the database, we can simply create an interface such as UserRepository and extend it to the JpaRepository interface to have Spring automatically inject basic CRUD operations that can be made onto the user table. It's worth noting that when using the JpaRepository, we must established the entity type and identifier (primary key) type to properly setup the CRUD operations desired:

```
4 import org.springframework.data.jpa.repository.JpaRepository;
7
8 // USER REPOSITORY
9
10 public interface UserRepository extends JpaRepository<User, Long> {
11
12
13
14 }
```

For manipulating Task entities within the SQL database, the **TaskRepository** interface is defined, extending the JpaRepository with the Task and TaskId classes established as the entity type and identifier type, respectively.

Apart from the common CRUD operations, the TaskRepository defines custom queries functions to retrieve the desired data for the app's response towards the user.

For example, instead of retrieving all of the tasks from all users of the database, we want to only retrieve the list of task based on a userID, which is defined with the **@Query** annotation on the **findByUser** function, that retrieves the desired tasks with the JPQL query established. Similarly, the **findTaskByUser** function leverages the JPQL queries to get a specific task from the user with both the task ID and user ID:

```
4 import java.util.List;
2
3 // TASK REPOSITORY
4
5 public interface TaskRepository extends JpaRepository<Task, TaskId> {
6
7     // Retrieve the list of tasks for a particular user:
8     @Query("SELECT t FROM Task t WHERE t.userId = :userId")
9     List<Task> findByUser(long userId);
10
11     // Retrieve a specific task, from a specific user:
12     @Query("SELECT t FROM Task t WHERE t.taskId = :taskId AND t.userId = :userId")
13     Optional<Task> findTaskByUser(int taskId, long userId);
14
15     // Count the number of tasks for a particular user:
16     @Query("SELECT COUNT(*) FROM Task t WHERE t.userId = :userId")
17     long countTasks(long userId);
18
19 }
```

Service

A service layer for both the Task and User entities were established that acts out as the intermediary between the controllers and repositories of the Task application, as it encapsulates the business logic and rules required, as well as it administers the interaction among the components of the program.

To improve flexibility and maintainability of the code, both an interface and an implementation class are created to define a contract about the available methods without exposing the implementation details.

The **UserService** interface defines the contract for managing the users accounts of the Task application through basic CRUD operation:

```
7 // USER SERVICE CONTRACT
8
9 public interface UserService {
10
11     // CREATE new user
12     User createUser(User user);
13
14     // UPDATE existing user
15     User updateUser(User user);
16
17     // UPDATE password of an existing user:
18     void changePassword(User user);
19
20     // READ user's details based on the ID provided:
21     User getUserById(long id);
22
23     // READ all the users in the database:
24     List<User> getAllUsers();
25
26     // READ the number of users in the database:
27     long countUser();
28
29     // DELETE the user from the database by its ID:
30     void deleteUser(long id);
31
32
33 }
```

Whereas the **UserServiceImpl** provides the concrete implementation of the contract, establishing how the user should be created, read, updated and deleted. This is where the **UserRepository** is injected through Spring's **@Autowired** annotation to leverage the repository's methods to perform the operations on the database:

```
30 import java.util.List;
12
13 // CONCRETE IMPLEMENTATION OF THE USER SERVICE
14
15 @Service
16 public class UserServiceImpl implements UserService {
17
18     // Inject the UserRepository onto the service:
19     @Autowired
20     private UserRepository userRepository;
21
22
23     // Method to create a new user in the database:
24     @Override
25     @Transactional
26     public User createUser(User user) {
27         User newUser = userRepository.save(user);
28
29         return newUser;
30     }
31
```



```
// Method to update an existing user:
@Transactional
@Override
public User updateUser(User user) {
    return userRepository.save(user);
}

// Retrieve a user from the database through its ID:
@Override
public User getUserById(long id) {
    return userRepository.findById(id).orElseThrow(
        () -> new EntityNotFoundException("No user with ID #" + id + " was found..."));
}

// Retrieve all users of the database:
@Override
public List<User> getAllUsers() {
    return userRepository.findAll(); // <- Example of JpaRepository built-in method
}

// Count the # of users in the database:
@Override
public long countUser() {
    return userRepository.count();
}

// Delete the user from the database with its ID:
@Transactional
@Override
public void deleteUser(long id) {
    userRepository.deleteById(id);
}
```

Likewise, a **TaskService** interface and implementation class was created to act as the service layer for the Task entity, but using the custom queries defined in the TaskRepository to take advantage of the composite key (TaskId) to perform the CRUD operations to only the tasks that are specific to one user by means of its ID:

TaskService interface:

```
3 import java.util.List;
7
8 //TASK SERVICE CONTRACT
9
10 public interface TaskService {
11
12     // CREATE a new task for a specific user:
13     Task addTaskToUser(Task task);
14
15     // READ a specific task, from a specific user:
16     Task getTaskByUser(long userId, int taskId);
17
18     // READ the list of tasks from a particular user:
19     List<Task> getTasksOfUser(long userId);
20
21     // READ the number of tasks available for a user:
22     long countTasks(long userId);
23
24     // UPDATE a task details for a specific user:
25     Task updateTaskOfUser(Task task);
26
27     // DELETE a specific task based on its composite key (Task ID and User ID):
28     void deleteTask(TaskId id);
29
30 }
```

TaskServiceImpl class:

```
5 // TASK SERVICE IMPLEMENTATION
6
7 @Service
8 public class TaskServiceImpl implements TaskService{
9
10     // Automatically inject the TaskRepository onto the service to perform the operations:
11     @Autowired
12     private TaskRepository taskRepository;
13
14     @Transactional
15     @Override
16     public Task addTaskToUser(Task task) {
17         return taskRepository.save(task); // Use built-in JpaRepository method to save the task object:
18     }
19
20     @Override
21     public Task getTaskByUser(long userId, int taskId) {
22
23         // Find the specific task of a specific user by means of their ID
24         // If the task was not found, throw an exception that the entity was not found:
25         return taskRepository.findTaskByUser(taskId, userId).orElseThrow(
26
27             () -> new EntityNotFoundException("Task of ID #" + taskId + " for User #" + userId + " does not exist...")
28
29         );
30     }
31 }
```

```
@Override
public long countTasks(long userId) {
    return taskRepository.countTasks(userId);
}

@Override
public List<Task> getTasksOfUser(long userId) {
    return taskRepository.findByUser(userId);
}

@Transactional
public Task updateTaskOfUser(Task task) {

    // Check if the task exists for the specific user:
    Task oldTask = getTaskByUser(task.getUserId(), task.getId());

    // If that's the case, then update the title and status of the existing task:
    oldTask.setTitle(task.getTitle());
    oldTask.setStatus(task.getStatus());

    // Save the existing task:
    return taskRepository.save(oldTask);
}

@Transactional
@Override
public void deleteTask(TaskId id) {
    taskRepository.deleteById(id);
}
```

REST Controller

With all the other components of the application set up, work began to develop the REST controllers who are responsible for communicating with the client and the backend of the application, handling the user's HTTP requests and delegating them to the service layer to process the data, so as to deliver the appropriate response to the client in standard JSON format. This is where the endpoints are defined and accessed by the client to perform the necessary operations towards the corresponding Task or User entity.

To define a class as a REST controller to Spring, the **@RestController** annotation is used, along with a **@RequestMapping** that maps out the entry point of the application.

In the case of the **UserRestController**, the path of `"/api/users"` is set as the entry point or base URL for the controller:

```
21 // USER REST CONTROLLER
22
23 @RestController // <- Tells Spring that the class should
24 @RequestMapping("/api/users") // Entry point
25 public class UserRestController {
26
```

The controller methods, together with the GET, POST, PUT and DELETE mapping annotations, set up the various endpoints to handle the different types of HTTP requests the user can send to perform the desired operation.

The following method is a simple example of a GET handler function that retrieves the list of users from the database with the service layer, in the entry point of the RESTful application:

```
25 public class UserRestController {
26
27     @Autowired
28     private UserService userService;
29
30     // Retrieve the list of users of the database:
31     @GetMapping
32     public List<User> getAllUsers() {
33         return userService.getAllUsers();
34     }
35
```

Other methods include parameters to handle the input data sent by the client with the use of the **@PathVariable** and **@RequestBody** annotations.

Path variable extracts the value passed in the URL path. For example, the **getUser()** method extracts the value passed as the "{user_id}" parameter included in the GET endpoint to retrieve a specific user by its ID:

```
// Get a specific user by its ID:
@GetMapping("/{user_id}")
public User getUser(@PathVariable("user_id") long id) {
    return userService.getUserById(id);
}
```

The RequestBody takes care of mapping the HTTP request body sent by the client into the domain object required.

In the **createUser()** method, for example, when the user sends out the POST request to the "/create" endpoint, Spring checks out the data passed as the HTTP request body and deserializes it into a User object to properly create a new user into the database:

```
// Create a new user into the database, using the HTTP request body data for the instance attributes:
@PostMapping("/create")
public String createUser(@RequestBody User newUser) {

    // Set the ID as 0, and let the repository handle the automatic assignment of the new user ID:
    newUser.setID(0);

    userService.createUser(newUser);

    // Show the following message, if successful:
    return "Created new user : \n"+ newUser;
}
```

For the **updateUser()** and **changePassword()** methods, it combines the Path Variable and Request Body features to correctly handle the update requests of the client to modify an existing user's data:

```
// Handler method to update an existing user:
@PutMapping("/{user_id}")
public String updateUser(@PathVariable("user_id") long id, @RequestBody User user) {

    // Verify that the user exists in the database by its ID:
    userService.getUserById(id);

    user.setID(id);

    // Update the user with the new data from the request's body:
    userService.updateUser(user);

    return "Updated user with ID # " + id;
}
```

```
// PATCH handler method to change an existing user's password:
@PatchMapping("/{user_id}/password")
public String changePassword(@PathVariable("user_id") long id, @RequestBody User user) {

    user.setID(id);

    // If the password is not empty, carry on:
    if(user.getPassword() != null || user.getPassword().isBlank()) {

        // Delegate the password change to the service layer:
        userService.changePassword(user);

        // Return the following message, if successful:
        return "Changed password for user with ID #" + user.getID();
    }

    // If the password is empty, throw a Runtime error:
    throw new RuntimeException("Password cannot be empty. Please try again.");
}
```

Finally, the UserRestController class ends with a DELETE handler method that takes care of removing an existing user from the database:

```
// DELETE handler method to remove a user from the database by its ID:
@DeleteMapping("/{user_id}")
public String deleteUser(@PathVariable("user_id") long id) {

    // Check if the user exists. If not, don't do anything:
    try {
        userService.getUserById(id);
    } catch (EntityNotFoundException e) {return "";}

    userService.deleteUser(id);

    // Return the following message, if successful:
    return "User with ID # " + id + " was deleted";
}
```

Another controller called the **TaskRestController** was made to handle the HTTP requests to perform the CRUD operations of the tasks onto the database. But because the Task entity uses a composite key (TaskId) as its primary key, both the TaskService and UserService are applied to implement the necessary logic to perform the CRUD operations of the tasks for a specific user.

```
23 //TASK REST CONTROLLER
24
25 @RestController
26 @RequestMapping(path = "/api/user/{user_id}") // In order to access the entry point, a user id must be passed:
27 public class TaskRestController {
28
29     @Autowired
30     private TaskService taskService;
31
32     @Autowired
33     private UserService userService;
34
35     // Get the list of tasks for a specific user:
36     @GetMapping("/tasks") // Use the user_id path variable passed in the entry point:
37     public List<Task> getAllTasks(@PathVariable("user_id") long userId) {
38
39         return taskService.getTasksOfUser(userId);
40     }
41
42     // Get the number of tasks of a specific user:
43     @GetMapping("/tasks/size")
44     public String getNumTasks(@PathVariable("user_id") long userId) {
45         long size = taskService.countTasks(userId);
46
47         return "Number of tasks for User #" + userId + " : " + size;
48     }
49
50     // Get a specific task, for a specific user:
51     @GetMapping("/task/{task_id}")
52     public Task getTask(@PathVariable("user_id") long userId, @PathVariable("task_id") int taskId) {
53
```

```
54         // Pass both the user ID and task ID of the path variables to retrieve the individual task:
55         return taskService.getTaskByUser(userId, taskId);
56     }
57
58     // Create a new task for a specific user:
59     @PostMapping("/task")
60     public Task createTask(@PathVariable("user_id") long userId, @RequestBody Task task) {
61
62         // Retrieve the existing user from the database:
63         User user = userService.getUserById(userId);
64
65         // Get the list of tasks of the user:
66         List<Task> tasks = user.getTasks();
67
68         // Assign an ID for the new task
69         // If the list is empty, assign it 1, if not, assign it as the subsequent ID available:
70         int taskId = tasks.isEmpty() ? 1 : tasks.get(tasks.size()-1).getTId() + 1;
71
72         // Set up the IDs for the new task
73         task.setTId(taskId);
74         task.setUIId(userId);
75
76         // Add it to the database:
77         return taskService.addTaskToUser(task);
78
79     }
80
81
```

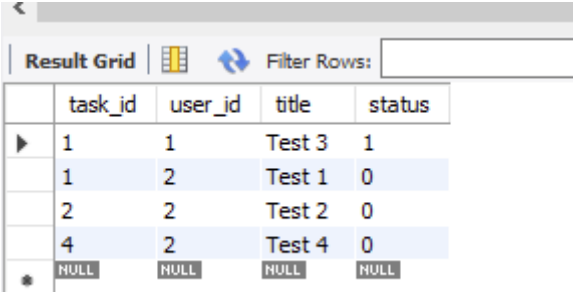


```
82 // Change an existing task's details of a specific user, based on the task ID and user ID:
83 @PutMapping("/task")
84 public Task updateTask(@PathVariable("user_id") long userId, @RequestBody Task task ) {
85
86     // Set up the user ID, if it was not passed in the request's body:
87     task.setUIId(userId);
88
89     return taskService.updateTaskOfUser(task);
90 }
91
92 // Delete a task from the list of a specific user:
93 @DeleteMapping("/task/{task_id}")
94 public String deleteTask(@PathVariable("user_id") long userId, @PathVariable("task_id") int taskId) {
95
96     // Create the composite key, based on the task ID and user ID:
97     TaskId tId = new TaskId(taskId, userId);
98
99     taskService.deleteTask(tId);
100
101     return "Deleted Task #" + taskId + " from User with ID #" + userId;
102 }
```

DEMONSTRATION

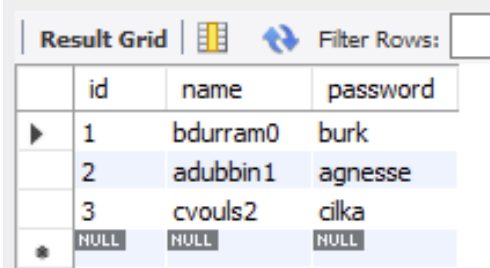
After starting the RESTful project, we can make use of an API Client such as Postman or Bruno to start sending requests and receiving responses from the application, such as the following:

INITIAL TASKS AND USERS IN THE DATABASE:



A screenshot of a database result grid interface. It features a 'Result Grid' tab, a 'Filter Rows' input field, and a table with four columns: 'task_id', 'user_id', 'title', and 'status'. The table contains five rows of data, with the last row being a 'NULL' row. The first four rows are: (1, 1, 'Test 3', 1), (1, 2, 'Test 1', 0), (2, 2, 'Test 2', 0), and (4, 2, 'Test 4', 0).

task_id	user_id	title	status
1	1	Test 3	1
1	2	Test 1	0
2	2	Test 2	0
4	2	Test 4	0
NULL	NULL	NULL	NULL



A screenshot of a database result grid interface. It features a 'Result Grid' tab, a 'Filter Rows' input field, and a table with three columns: 'id', 'name', and 'password'. The table contains four rows of data, with the last row being a 'NULL' row. The first three rows are: (1, 'bdurram0', 'burk'), (2, 'adubbin1', 'agnesne'), and (3, 'cvouls2', 'cilka').

id	name	password
1	bdurram0	burk
2	adubbin1	agnesne
3	cvouls2	cilka
NULL	NULL	NULL

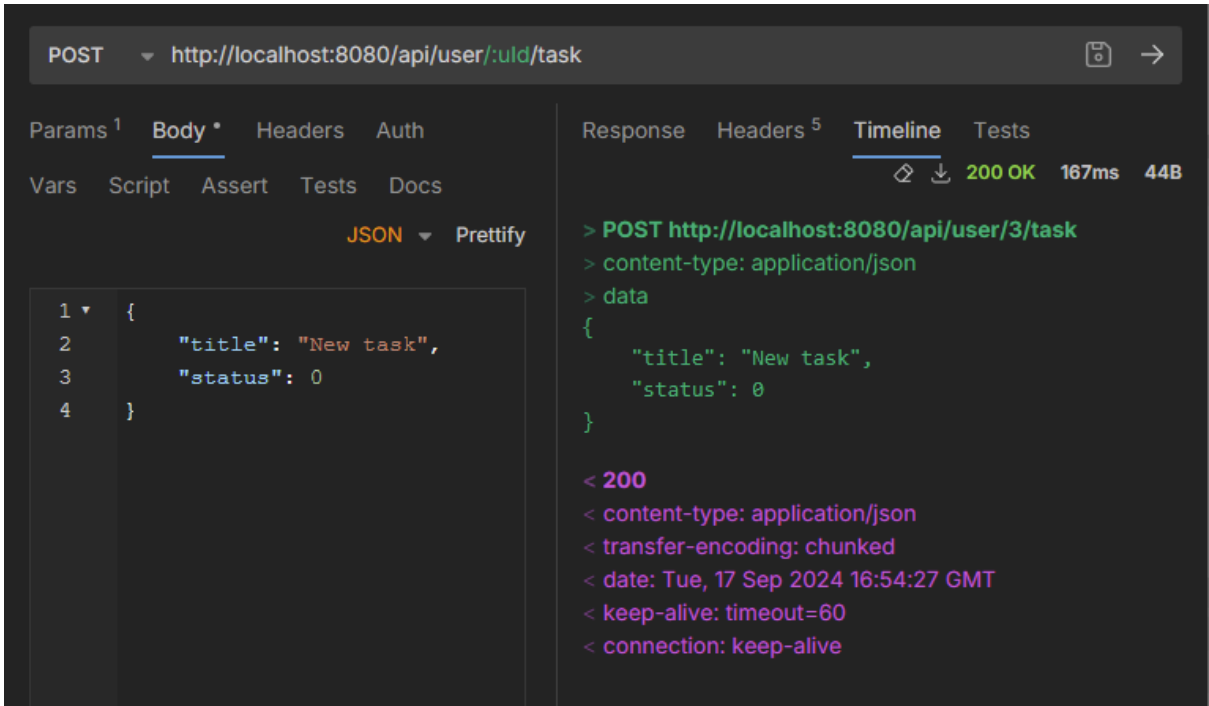
READ THE TASK LIST OF USER #2 (adubbin1)

The screenshot shows a REST client interface with a GET request to `http://localhost:8080/api/user/:uld/tasks`. The path parameter `uld` is set to `2`. The response is a JSON array of 3 tasks, each with a title, status, and tid.

Name	Value
uld	2

```
[
  {
    "title": "Test 1",
    "status": "TODO",
    "tid": 1
  },
  {
    "title": "Test 2",
    "status": "TODO",
    "tid": 2
  },
  {
    "title": "Test 4",
    "status": "TODO",
    "tid": 4
  }
]
```

POST NEW TASK FOR USER #3 (cvouls2)



Result Grid					Filter Rows:	
	task_id	user_id	title	status		
▶	1	1	Test 3	1		
	1	2	Test 1	0		
	1	3	New task	0		
	2	2	Test 2	0		
	4	2	Test 4	0		
✱	NULL	NULL	NULL	NULL		

EDIT TASK DETAILS FOR USER #3 WITH PUT REQUEST

PUT

http://localhost:8080/api/user/:uld/task

Params¹

Body^{*}

Headers

Auth

Vars

Script

Assert

Tests

Docs

Query

Name	Value
------	-------

+ Add Param

Path [?]

Name	Value
uld	3

Response

Headers⁵

Timeline

Tests

200 OK

37ms

48B

> PUT http://localhost:8080/api/user/3/task

> content-type: application/json

> data

{

"tid": 1,

"title": "Changed task",

"status": 1

}

< 200

< content-type: application/json

< transfer-encoding: chunked

< date: Tue, 17 Sep 2024 16:56:04 GMT

< keep-alive: timeout=60

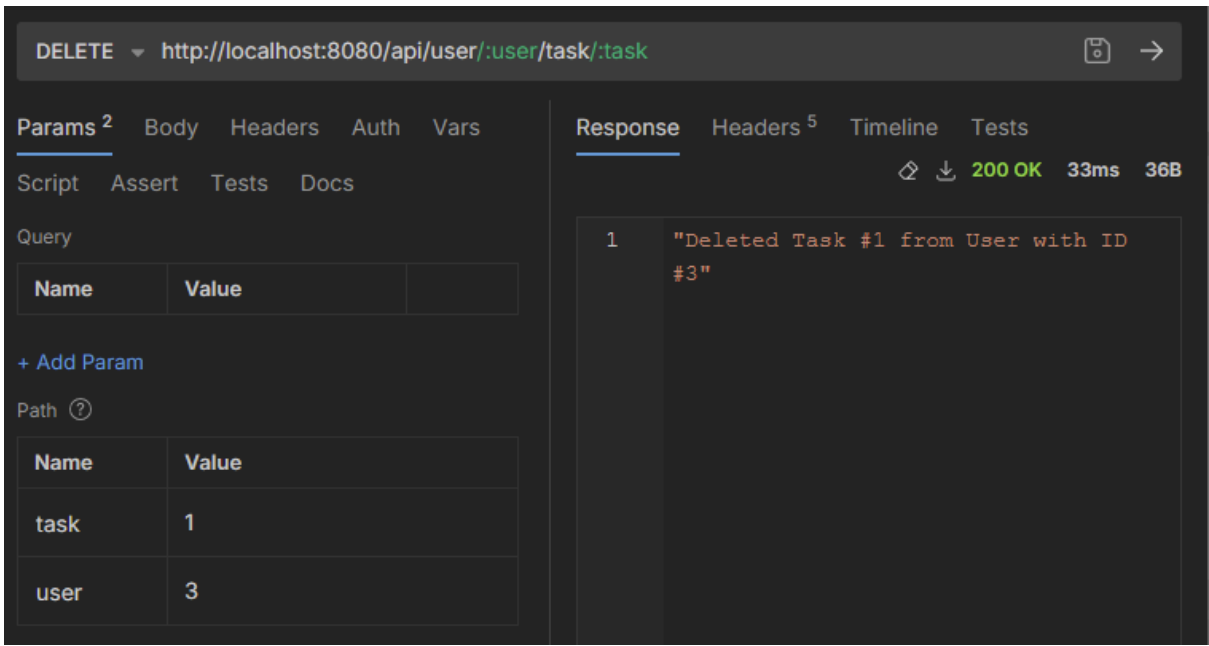
< connection: keep-alive

Result Grid

Filter Rows:

	task_id	user_id	title	status
▶	1	1	Test 3	1
	1	2	Test 1	0
	1	3	Changed task	1

DELETE TASK #1 FROM USER #3



Result Grid				
	task_id	user_id	title	status
▶	1	1	Test 3	1
	1	2	Test 1	0
	2	2	Test 2	0
	4	2	Test 4	0
✱	NULL	NULL	NULL	NULL

COUNT NUMBER OF TASKS FOR USER #2

GET `http://localhost:8080/api/user/:uld/tasks/size`

Params ¹ Body Headers Auth Vars

Script Assert Tests Docs

Query

Name	Value
uld	2

+ Add Param

Path [?]

Name	Value
uld	2

Response Headers ⁵ Timeline Tests

200 OK 15ms 31B

```
1 "Number of tasks for User #2 : 3"
```

GET THE LIST OF USERS IN THE DATABASE

GET `http://localhost:8080/api/users`

Params Body Headers Auth Vars

Script Assert Tests Docs

Query

Name	Value
------	-------

+ Add Param

Path [?]

Name	Value
------	-------

Response Headers ⁵ Timeline Tests

200 OK 34ms 139B

```
1 [
2   {
3     "name": "bdurram0",
4     "password": "burk",
5     "id": 1
6   },
7   {
8     "name": "adubbin1",
9     "password": "agnesne",
10    "id": 2
11  },
12  {
13    "name": "cvouls2",
14    "password": "cilka",
15    "id": 3
16  }
17 ]
```

GET INDIVIDUAL USER OF THE DATABASE:

The screenshot shows a REST client interface with a GET request to `http://localhost:8080/api/users/:userId`. The path parameter `userId` is set to `1`. The response is a JSON object with the following structure:

```
{
  "name": "bdurram0",
  "password": "burk",
  "id": 1
}
```

The response status is **200 OK**, with a response time of **11ms** and a body size of **44B**.

CREATE NEW USER INTO THE DATABASE:

The screenshot shows a REST client interface with a POST request to `http://localhost:8080/api/users/create`. The request body is a JSON object:

```
{
  "name": "luis",
  "password": "secretpassowrd"
}
```

The response is a string message: `"Created new user : \nUser(ID=4, name=luis, password=secretpassowrd, tasks=[]) "`. The response status is **200 OK**, with a response time of **53ms** and a body size of **76B**.

Result Grid			
Filter Rows:			
	id	name	password
▶	1	bdurram0	burk
	2	adubbin1	agnesse
	3	cvouls2	cilka
	4	luis	secretpassowrd
✱	NULL	NULL	NULL

CHANGE PASSWORD FOR THE NEW USER #4

PATCH http://localhost:8080/api/users/:userId/password

Params¹Body *HeadersAuth

VarsScriptAssertTestsDocs

JSONPrettify

1 {

2 "password": "testpassword

3 }

ResponseHeaders⁵TimelineTests

200 OK20ms36B

> PATCH http://localhost:8080/api/users/4/password

> content-type: application/json

> data

{

"password": "testpassword"

}

< 200

< content-type: application/json

< content-length: 36

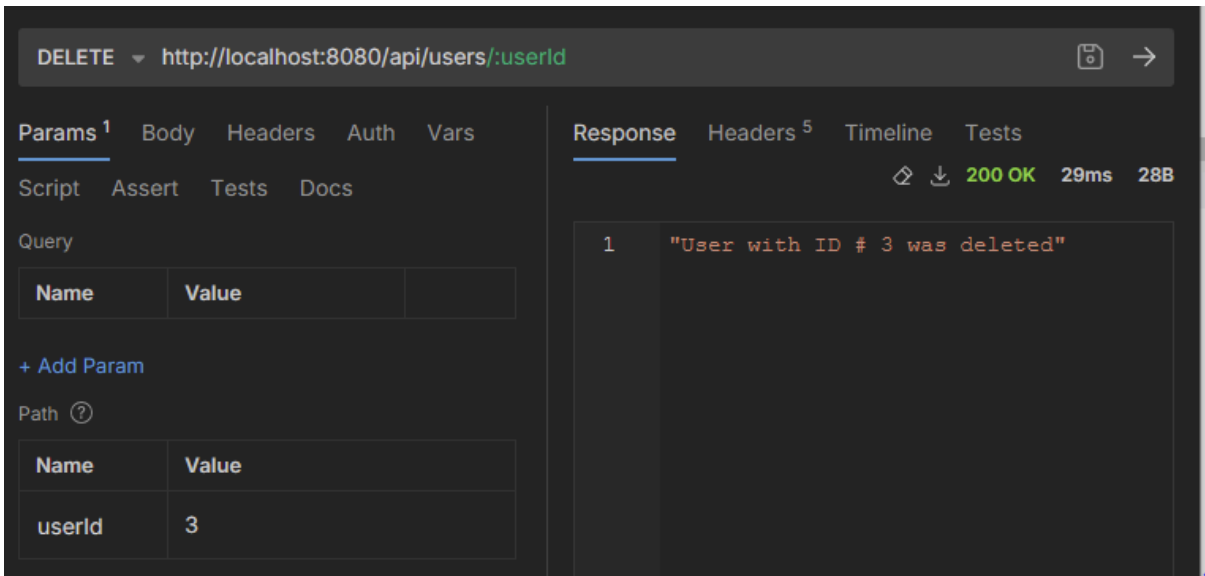
< date: Tue, 17 Sep 2024 17:04:18 GMT

< keep-alive: timeout=60

< connection: keep-alive

Result Grid			
	id	name	password
▶	1	bdurram0	burk
	2	adubbin1	agnesse
	3	cvouls2	cilka
	4	luis	testpassword
	NULL	NULL	NULL

DELETE USER #3 FROM THE DATABASE



Result Grid			
	id	name	password
▶	1	bdurram0	burk
	2	adubbin1	agnesse
	4	luis	testpassword
✱	NULL	NULL	NULL