KF7014 – Workshop week 04

CRUD operations implementation

Introduction

In this workshop, you will implement CRUD (Create, Read, Update and Delete) REST API around a single resource.

Scenario

You will develop a RESTful service for `Employees` management. The service should support adding, viewing, modifying, and removing employees; a standardised usage of HTTP verbs known as Create, Read, Update, Delete (CRUD).

REST requests are made over HTTP. They use the same HTTP verbs that web browsers use to retrieve webpages and send data to servers. The verbs are:

- `GET`: Retrieve data from the web API.
- `POST`: Create a new item of data on the web API.
- `PUT`: Update an item of data on the web API.
- `DELETE`: Delete an item of data on the web API.

RESTful APIs are defined through:

- A base URI.
- HTTP methods, such as `GET`, `POST`, `PUT`, or `DELETE`.
- A media type for the data, such as JavaScript Object Notation (JSON) or XML.

In more details, you will implement the following:

	Description	Request body	Response body
`GET /employees`	Get all employees	None	Array of employees
`GET /employees/{id}`	Get an employee by ID	None	employee
`POST /employees`	Add a new employee	employee	employee
`PUT /employees/{id}`	Update an existing employee	employee	None
`DELETE /employees/{id}`	Delete an employee	None	None

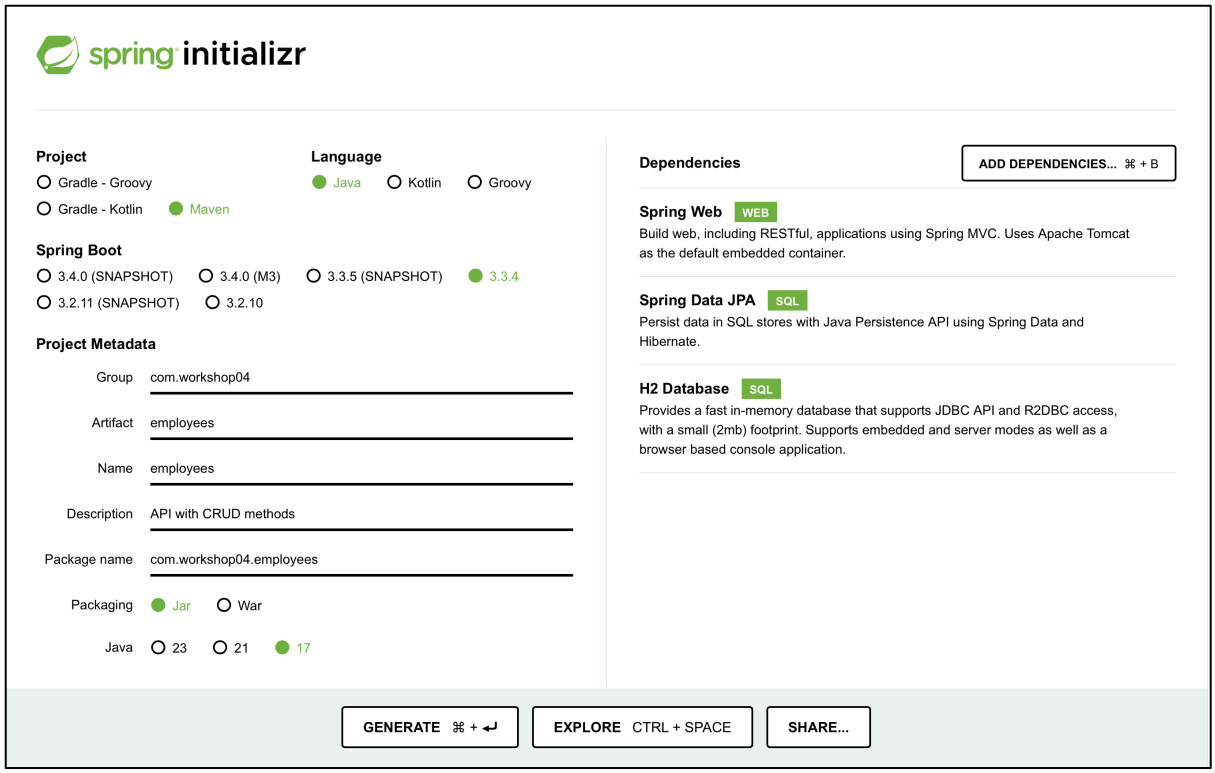
Software requirements

- Spring Boot.
- Maven or Gradle for dependency management.

Exercise 1: Creating the API project

Step 1: Create the project

- Create a new project with Spring Initializr https://start.spring.io
- Configure the project with the following:
    - Project: Maven
    - Language: Java
    - Dependencies: `Spring Web`, `Spring Data JPA`, `H2 Database (In-memory)`



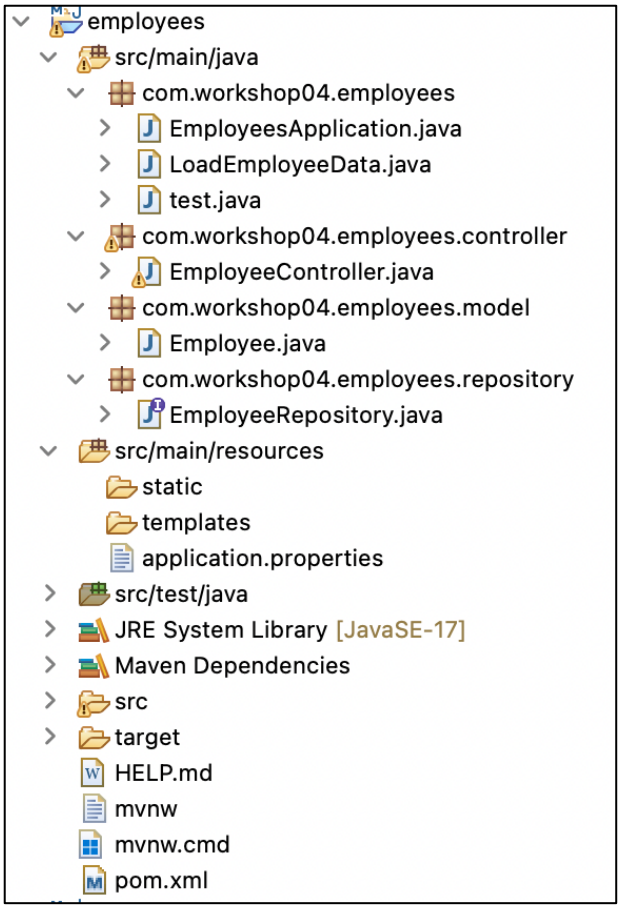
- Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.
- Import to Eclipse as Existing Maven Project.

Step 2: Setup the project structure

To build the `employees` project, you will need to create three classes:

- **`Employee.java` (Model)** – which will represent the actual resource (`employee`) the API will be dealing with. This class will be very simple containing only three properties: `id`, `name` and `role`.
- **`EmployeeRepository.java` (Repository)** – which will act as a temporary data-storage layer for the API. Later, you will add code to access the database and store data. In this workshop, we will use in-memory data.
- **`EmployeeController.java` (Controller)** – which will act as the main entry point for the API. Here you will use annotations to map class methods to endpoints.
- **`EmployeesApplication.java` (Main Application)** – which is the main point to run the application.

Create the necessary packages and classes, so that the project structure should be:



Step 3: Create the Model (`Employee.java`)

In this class, you should:

- Add the attributes `id`, `name` and `role`, then create the constructor, and add the getters and setters methods.
- Add the necessary `annotations` to make this object ready for storage in a JPA-based data store.
- Add a custom constructor to create a new instance while not yet having an `id`.

The code should be:

Employee.java

```java
package com.workshop04.employees.model;

import java.util.Objects;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

@Entity
public class Employee {
    private @Id @GeneratedValue Long id;
    private String name;
    private String role;

    // default constructor
    public Employee() {}

    // constructor
    public Employee(String name, String role) {
        this.name = name;
        this.role = role;
    }

    // getters and setters
    public Long getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public String getRole() {
        return this.role;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setRole(String role) {
        this.role = role;
    }

    @Override
    public boolean equals(Object o) {

        if (this == o)
            return true;
        if (!(o instanceof Employee))
            return false;
        Employee employee = (Employee) o;
        return Objects.equals(this.id, employee.id)
                && Objects.equals(this.name, employee.name)
                && Objects.equals(this.role, employee.role);
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.id, this.name, this.role);
    }

    @Override
    public String toString() {
        return "Employee{"
                + "id=" + this.id
                + ", name='" + this.name + '\''
                + ", role='" + this.role + '\'' + '}'
            ;
    }
}
```

Step 4: Create the Repository (`EmployeeRepository.java`)

In this class, you will:

- Build the data repository around an **ArrayList**, which is where you will store the `employees`.
- Import the necessary Spring libraries and the `Employee` model class.
- Add the necessary annotation to tell **SpringBoot** that this is the class in charge of data storage.

The code should be:

EmployeeRepository.java

```
1  package com.workshop04.employees.repository;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4
5  import com.workshop04.employees.model.Employee;
6
7  public interface EmployeeRepository extends JpaRepository<Employee, Long> {
8
9  }
```

Step 5: Create the Controller (`EmployeeController.java`)

The **controller** class will be the entry point for the application. This means the server will be started and the methods will be automatically mapped to the endpoints.

- Import necessary packages including the model `Employee` and repository `EmployeeRepository` classes, as well as the following springframework libraries:
- Add in this class is the `@RestController` annotation to tell the framework that this controller will be used by the server.
- Add the `@RequestMapping` annotation to specify the content type of the responses from all methods. The information of each method will be specified at class level, since the response of each method will be with JSON.
- Create an instance of the repository class and use it in the constructor.

EmployeeController.java

```java
package com.workshop04.employees.controller;

import java.util.List;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.workshop04.employees.model.Employee;
import com.workshop04.employees.repository.EmployeeRepository;


@RestController
//@Secured(AuthoritiesConstants.USER)
public class EmployeeController {

    private final EmployeeRepository repository;

    // constructor
    public EmployeeController(EmployeeRepository repository) {
        this.repository = repository;
    }
```

Remember: The `Controller` class, without the annotations, is quite simple, where every method uses the `Repository` to perform their respective action, and they return a `ResponseEntity` as a result. As part of the return values, you can also use of the `HttpStatus` enum, which shows the standard **HTTP Status** codes and headers to convey the outcome and metadata of requests and responses.

CRUD methods: Now you should create the CRUD methods to perform each action. The `@GetMapping`, `@PostMapping`, `@PutMapping` and `@DeleteMapping` annotations clearly map the method they annotate to the `HTTP` Method they reference.

As part of the annotation, you should also specify the URI, so you have control over what the endpoints look like. Keep in mind that as part of `REST`, the URIs should reference resources.

Another interesting detail about route mapping with SpringBoot, is that URL parameters, such as the `id` of the `employee` are easily mapped to method parameters. So, if you send a `DELETE` request to `/employee/1`, the value `1` will be received on the `id` parameter of the `deleteEmployee` method.

Exercise 2: Reading data

In this exercise, you will create different `GET` methods for reading data in the Controller class.

Step 1: Get all `Employees`

- Implement the `GET` method `getAllEmployees()` that get all the employees.
- You should use the `@GetMapping` annotation.
- This method should return the arraylist of `employees`.

```
// get all employees
@GetMapping("/employees")
public List<Employee> getAllEmployees() {

    return repository.findAll();
}
```

Step 3: Get `Employee` item by `Id`

- Implement another useful method `getEmployeeById()`to get a `Employee` item by `Id`.
- You should use the annotation `@GetMapping("/{id}")`.
- If the item is found in the `Repository`, the `employee` is returned and the return `HTTP` status is `OK`.
- If not found, the return status would be `NOT_FOUND`.

```
@GetMapping("/employees/{id}")
public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {

    return repository.findById(id)
            .map(employee -> new ResponseEntity<>(employee, HttpStatus.OK))
            .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

The returned `ResponseEntity` instance used in the preceding action is mapped to the corresponding HTTP status code in the following table:

HTTP status	HTTP status code	Description
`Ok`	200	An item that matches the provided `id` parameter exists in the data.
`NotFound`	404	An item that matches the provided `id` parameter does not exist in the data.

Step 4: Load sample data

Now, we need to preload sample data for testing purposes. The following class gets loaded automatically by Spring:

LoadEmployeeData.java

```java
package com.workshop04.employees;

/* Import necessary packages */
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * Import other packages from the project
 */
import com.workshop04.employees.model.Employee;
import com.workshop04.employees.repository.EmployeeRepository;

/**
 * LoadEmployeeData is for loading sample data in the in-memory database
 * when running the application.
 */
@Configuration
public class LoadEmployeeData {

    private static final Logger log = LoggerFactory.getLogger(LoadEmployeeData.class);

    @Bean
    CommandLineRunner initDatabase(EmployeeRepository repository) {

        return args -> {
            log.info("Preloading " + repository.save(new Employee("Maria Salama", "Lecturer")));
            log.info("Preloading " + repository.save(new Employee("John Rooksby", "Assitant Professor")));
        };
    }
}

```

Step 5: Configure and run the server

Given how SpringBoot does much everything, all you need to take care of is configuring the port for the web server. Open the **`application.properties`** file and add the following line:

```
server.port=8082
```

Next, click on the **Run** button in the IDE, and you should get an output like this:

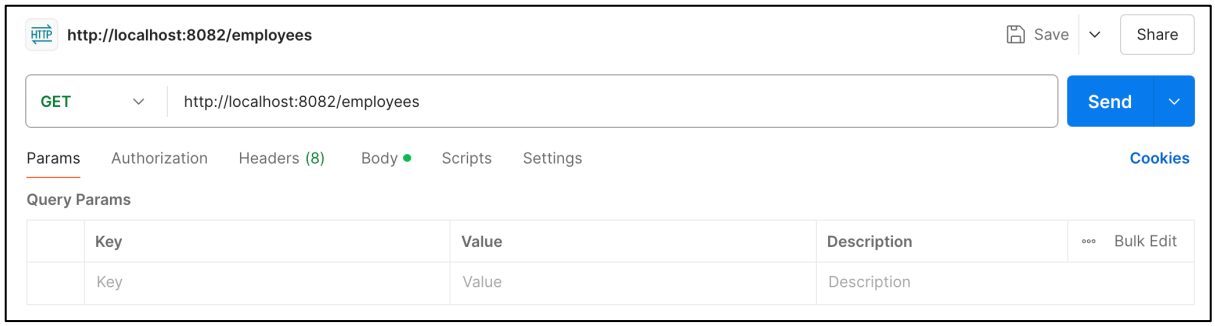
Problems Javadoc Declaration Console

<terminated> EmployeesApplication [Java Application] /Users/m.salama/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_22.0.2.v20240802-1626/jre/bin/java (21 Oct 2024,

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.3.4)

2024-10-21T19:01:24.249+01:00  INFO 21119 --- [employees] [           main] c.w.employees.EmployeesApplication       : Starting EmployeesApplication using Java 22.0.2 with
2024-10-21T19:01:24.265+01:00  INFO 21119 --- [employees] [           main] c.w.employees.EmployeesApplication       : No active profile set, falling back to 1 default prof
2024-10-21T19:01:26.221+01:00  INFO 21119 --- [employees] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT
2024-10-21T19:01:26.329+01:00  INFO 21119 --- [employees] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 98 ms. Fo
2024-10-21T19:01:29.697+01:00  INFO 21119 --- [employees] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8082 (http)
2024-10-21T19:01:29.740+01:00  INFO 21119 --- [employees] [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-10-21T19:01:29.740+01:00  INFO 21119 --- [employees] [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.30]
2024-10-21T19:01:29.845+01:00  INFO 21119 --- [employees] [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2024-10-21T19:01:29.850+01:00  INFO 21119 --- [employees] [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed
2024-10-21T19:01:30.195+01:00  INFO 21119 --- [employees] [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2024-10-21T19:01:30.383+01:00  INFO 21119 --- [employees] [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection conn0: url=jdbc:h2:me
2024-10-21T19:01:30.384+01:00  INFO 21119 --- [employees] [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2024-10-21T19:01:30.472+01:00  INFO 21119 --- [employees] [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: defa
2024-10-21T19:01:30.527+01:00  INFO 21119 --- [employees] [           main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 6.5.3.Final
2024-10-21T19:01:30.561+01:00  INFO 21119 --- [employees] [           main] o.h.c.internal.RegionFactoryInitiator    : HHH000026: Second-level cache disabled
2024-10-21T19:01:30.896+01:00  INFO 21119 --- [employees] [           main] o.s.o.j.p.SpringPersistenceUnitInfo      : No LoadTimeWeaver setup: ignoring JPA class transform
2024-10-21T19:01:32.108+01:00  INFO 21119 --- [employees] [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000489: No JTA platform available (set 'hibernate.
2024-10-21T19:01:32.190+01:00  INFO 21119 --- [employees] [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence
2024-10-21T19:01:32.829+01:00  WARN 21119 --- [employees] [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Theref
2024-10-21T19:01:33.713+01:00  INFO 21119 --- [employees] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8082 (http) with context path
2024-10-21T19:01:33.735+01:00  INFO 21119 --- [employees] [           main] c.w.employees.EmployeesApplication       : Started EmployeesApplication in 10.791 seconds (proce
2024-10-21T19:01:33.822+01:00  INFO 21119 --- [employees] [           main] c.workshop04.employees.LoadEmployeeData  : Preloading Employee{id=1, name='Maria Salama', role='
2024-10-21T19:01:33.824+01:00  INFO 21119 --- [employees] [           main] c.workshop04.employees.LoadEmployeeData  : Preloading Employee{id=2, name='John Rooksby', role='
```

Step 6: Testing `GET` methods

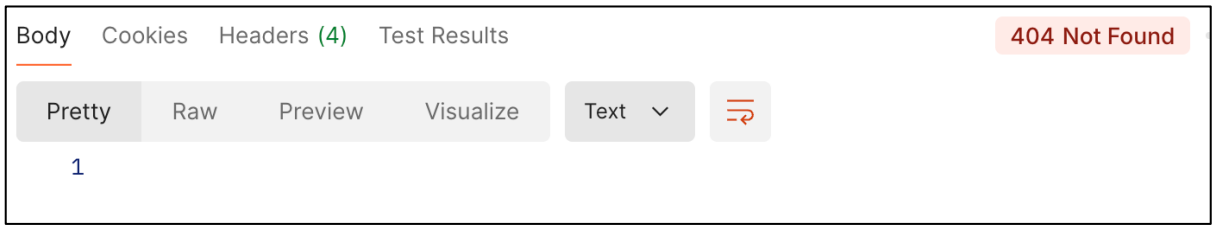
- You can test in Postman or using curl command.
- In **Postman**, set the **HTTP method** to `GET`, set the request URI to `https://localhost/8082/employees`



The call to `GET /employees` produces a response body with all the employees pre-loaded and the `HTTP` status code `200 OK`, similar to the following:

- Test the `getEmployeeByID()` method using the request URI to `https://localhost/8082/employees/1` . The output will be:

- Now, try finding an employee that does Test the `getEmployeeByID()` method using the request URI to `https://localhost/8082/employees/3` . The output will be:



Exercise 3: Creating new item

In this exercise, you will create the `POST` method for creating new item and inserting into the database.

Step 1: Create `createNewEmployee()` method

- Implement the `POST` method `createNewEmployee()` that allows inserting a new item in the database.
- You should use the `@PostMapping` annotation.
- This method parameter is the `newEmployee` data.
- If the request is successful, the method should return the new created `employee`, and the return `HTTP` status is `CREATED`.

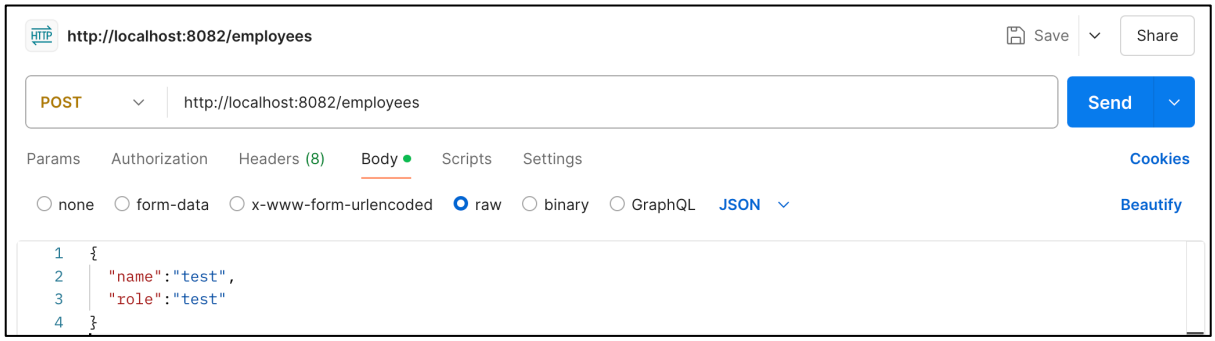
```
// Add a new item
@PostMapping("/employees")
public ResponseEntity<Employee> createNewEmployee(@RequestBody Employee newEmployee) {

    Employee employee = repository.save(newEmployee);
    return new ResponseEntity<>(employee, HttpStatus.CREATED);
}
```

The returned `ResponseEntity` instance used in the preceding action is mapped to the corresponding HTTP status code in the following table:

HTTP status	HTTP status code	Description
`Created`	201	The new employee is added to the in-memory database.
`BadRequest`	400	The request body object is invalid.

Step 2: Test `PUT` method

- In **Postman**, set the **HTTP method** to `POST`, set the request URI to `https://localhost/8082/employees/3`
- In the **Body** tab, select **raw** and enter the `key:value` pair of all parameters, as follows:



The call to `POST /employees` produces a response body with new `employee` and the `HTTP` status code `201 Created`, similar to the following:

- Now, try creating a new employee with wrong parameters. The output will be:

Exercise 4: Updating an existing item

In this exercise, you will create the `PUT` method for creating new item and inserting into the database.

Step 1: Create `updateEmployee()` method

- Implement the `PUT` method `updateEmployee()` that allows updating an existing item in the database.
- You should use the `@PutMapping` annotation.
- This method parameter is the `id` of the employee to be updated and `updatedEmployee` data.
- If the request is successful, the method should return the updated `employee`, and the return `HTTP` status is `NoContent`.

```
// Update existing employee
@PutMapping("/employees/{id}")
public ResponseEntity<Void> updateEmployee(@PathVariable Long id, @RequestBody Employee updatedEmployee) {

    return repository.findById(id)
            .map(employee -> {
                employee.setName(updatedEmployee.getName());
                employee.setRole(updatedEmployee.getRole());
                repository.save(updatedEmployee);
                return new ResponseEntity<Void>(HttpStatus.NO_CONTENT);
            })
            .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

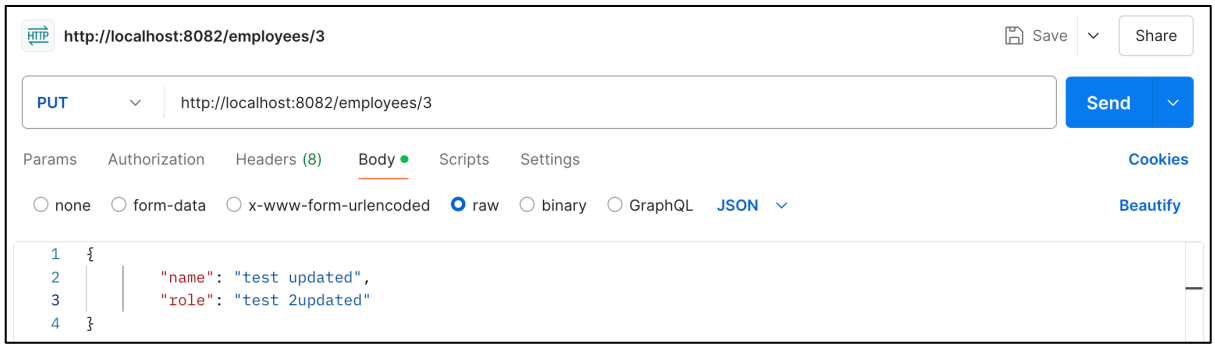
The returned `ResponseEntity` instance used in the preceding action is mapped to the corresponding HTTP status code in the following table:

HTTP status	HTTP status code	Description
`NoContent`	204	The employee is updated in the in-memory database.

`BadRequest`	400	The object could not be found with the `id`. OR The request body object is invalid.

Step 2: Test `POST` method

- In **Postman**, set the **HTTP method** to `PUT`, set the request URI to `https://localhost/8082/employees/3`
- In the **Body** tab, select **raw** and enter the `key:value` pair of all parameters, as follows:



The call to `PUT /employees` produces a response body with no content and the `HTTP` status code `204`, similar to the following:

Exercise 5: Delete an item

In this exercise, you will create the `DELETE` method for deleting an item from the database.

Step 1: Create `deleteEmployee()` method

- Implement the `PUT` method `deleteEmployee()` that allows deleting an existing item from the database.
- You should use the `@DeleteMapping` annotation.
- This method parameter is the `id` of the employee to be deleted.
- If the request is successful, the method should return the return `HTTP` status is `NoContent`.

```
// Delete an item
@DeleteMapping("/employees/{id}")
public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {
    return repository.findById(id)
            .map(employee -> {
                repository.delete(employee);
                return new ResponseEntity<Void>(HttpStatus.NO_CONTENT);
            })
            .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

The returned `ResponseEntity` instance used in the preceding action is mapped to the corresponding HTTP status code in the following table:

HTTP status	HTTP status code	Description
`NoContent`	204	The employee is deleted from the database.
`BadRequest`	400	The object could not be found with the `id`. OR The request body object is invalid.

Step 2: Test `POST` method

- In **Postman**, set the **HTTP method** to `DELETE`, set the request URI to `https://localhost/8082/employees/3`

This produces a response body with no content and the `HTTP` status code `204`, similar to the following:

Now you have created your API with all CRUD methods…

Directed learning: More `GET` methods

Try implementing custom messages for each HTTP status code.