



Spring Rest API
with IntelliJ IDEA

Content

1. REST API	3
1.1. Introduction	3
1.2. What is a REST API?	3
1.3. REST design principles	3
1.4. How REST APIs work	4
2. Spring	6
2.1. Introduction	6
2.2. Spring Boot	6
3. Creating a REST API with Spring Boot	7
3.1. Creating the project	7
3.2. Connecting Spring Boot to the Database	8
3.3. Setting up the MVC structure	9
3.4. Creating repository classes	9
3.5. Creating the controller	10
3.6. Running and testing the REST API	12
3.7. Completing our CRUD	14
Using IntelliJ IDEA built-in functionality	15
Using Postman	16
3.8. Data Transfer Objects (DTOs)	17
3.9. The service layer	20
3.10. Validating data	21
3.11. Activities	23
4. Consuming REST: client side	23
4.1. JAVA client	23
JSON file format	23
GET Request	24
POST Request	26
DELETE Request	27
4.2. Thymeleaf web application	27
Definition of the main controller	28
Creation of templates with Thymeleaf	29
Activity	31
Sending Data: POST Requests	31
Improving Our Application	32
Activity	33
Parameter Passing	34
Activity	35
Using Filters in Searches	35
Activity	36
5. Spring Boot testing	36
5.1. Activity	40
6. Security with Spring Boot	40
6.1. Introduction	40
6.2. Basic REST API Security	41
6.3. Using HTTPS	41
Certificates	41
Using certificates in Spring Boot	42
Role-based authentication	43
6.4. Securing the client side	45
6.5. Activity	47
7. REST API documentation	47
7.1. OpenAPI	48
7.2. Generating Documentation with IntelliJ IDEA	49
8. Spring application deployment	51
8.1. Generating the JAR file with IntelliJ IDEA	52
8.2. Virtual machine deployment	52
8.3. Activity	53
9. Annex: Unit testing class example	54
10. Bibliography	57

1. REST API

1.1. Introduction

Imagine you were suddenly transported into a foreign city where you don't speak the language. In fact, every person you encounter speaks a different language, and you aren't even sure which one they are. That's the situation faced by many developers and users today as they try to integrate different software and systems.

One of the greatest challenges of modern computing is its complexity. With millions of different software applications, services, and systems currently in use, each one is speaking its own "language." How can they ever hope to have meaningful communications by exchanging information with each other?

So what can you do in the face of this spiraling complexity and lack of universal communication standards? The answer, for many organizations, is using a REST API. But what is a REST API, exactly, and why do you need one?

1.2. What is a REST API?

An API, or *application programming interface*, is a set of rules that define how applications or devices can connect to and communicate with each other. A REST API is an API that conforms to the design principles of the REST, or *representational state transfer* architectural style. For this reason, REST APIs are sometimes referred to RESTful APIs.

First defined in 2000 by computer scientist Dr. Roy Fielding in his doctoral dissertation, REST provides a relatively high level of flexibility and freedom for developers. This flexibility is just one reason why REST APIs have emerged as a common method for connecting components and applications in a [microservices](#).

1.3. REST design principles

At the most basic level, an API is a mechanism that enables an application or service to access a resource within another application or service. The application or service doing the accessing is called the client, and the application or service containing the resource is called the server.

Some APIs, such as SOAP or XML-RPC, impose a strict framework on developers. But REST APIs can be developed using virtually any programming language and support a variety of data formats. The only requirement is that they align to the following six REST design principles - also known as architectural constraints:

1. **Uniform interface.** All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). Resources shouldn't be too large but should contain every piece of information that the client might need.
2. **Client-server decoupling.** In REST API design, client and server applications must be completely independent of each other. The only information the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application shouldn't modify the client application other than passing it to the requested data via HTTP.
3. **Statelessness.** REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it. In other words, REST APIs do not require any server-side sessions. Server applications aren't allowed to store any data related to a client request.
4. **Cacheability.** When possible, resources should be cacheable on the client or server side. Server responses also need to contain information about whether caching is allowed for the delivered resource. The goal is to improve performance on the client side, while increasing scalability on the server side.
5. **Layered system architecture.** In REST APIs, the calls and responses go through different layers. As a rule of thumb, don't assume that the client and server applications connect directly to each other. There may be a number of different intermediaries in the communication loop. REST APIs need to be designed so that neither the client nor the server can tell whether it communicates with the end application or an intermediary.
6. **Code on demand (optional).** REST APIs usually send static resources, but in certain cases, responses can also contain executable code (such as Java applets). In these cases, the code should only run on-demand.

1.4. How REST APIs work

The underlying idea behind the concept of REST API is to create a web service. Nowadays, all the information exchange occurs via the internet, where on one hand, there are offered services (such as a web page), and on the other hand, there are clients that consume such services. All this information exchange is done using the HTTP protocol.

To accomplish this task, HTTP offers a series of resources, identified by a URI (Uniform Resource Identifier), to which a series of requests can be made. These URIs are well known to us, as we use them daily in our browsers to access resources (for example, <https://www.google.com>).

The requests allowed by the HTTP protocol are basically five:

- GET: Allows retrieving or querying the information of a resource.
- POST: Allows creating a new resource.
- DELETE: Allows deleting a resource.
- PUT: Allows the complete modification of a resource.
- PATCH: Allows the partial modification of a resource.

By using these requests and the HTTP protocol, a REST API can be created to perform standard database functions, such as create, read, update, and delete records (also known as CRUD) using a resource. For example, a REST API would use a GET request to retrieve a record, a POST request to create it, a PUT request to update a record, and a DELETE request to delete it.

Let's suppose we want to create a REST API for our employees and departments database at the following URI: <http://www.myserver.com/myapi>. Here, we should have resources that allow us to perform each of the CRUD operations on both tables, each of these resources will be called endpoints. For example, we could have the following endpoints to retrieve information (using GET):

<http://www.myserver.com/myapi/employees>

<http://www.myserver.com/myapi/employees?job=CLERK>

<http://www.myserver.com/myapi/departments/loc=DALLAS>

<http://www.myserver.com/myapi/departments/20>

Each of these endpoints would return different information. The first one would return a list of all employees, the second only those who are clerks, the third would provide information about the department(s) located in Dallas, and the last one would provide information about the department with code 20. Therefore, a fundamental aspect when creating our REST API is to design it properly to meet all the requests that will be made without having excessive nesting -depth- in it.

When returning the requested information to the client, it can be delivered almost in any format, including JavaScript Object Notation (JSON), HTML, XML, Python, PHP, or plain text. JSON is popular because it is readable by both humans and machines, and it is language-independent. This information will generally be returned within the body of the HTTP response.

Request headers and parameters are also important in REST API calls because they include important identifying information, such as metadata, authorizations, Uniform Resource Identifiers (URIs), caching, cookies, and more. Request headers and response headers, along with conventional HTTP status codes, are used within well-designed REST APIs. Thus, typical status codes when using a REST API could be as follows:

Code	Status	Use
200	Ok	Ok in GET,PUT,PATCH
201	Created	Ok in POST
204	Not Content	Ok without returning data (usually in DELETE)
400	Bad Request	Client invocation error
401	Unauthorized	Incorrect credentials
403	Forbidden	Unauthoirzed (no privileges)
404	Not Found	Resource not found
500	Internal Server Error	Server error processing the request

2. Spring

2.1. Introduction

The Spring Framework is an application framework and inversion of control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE (Enterprise Edition) platform. Although the framework does not impose any specific programming model, it has become popular in the Java community as an addition to the Enterprise JavaBeans (EJB) model. The Spring Framework is open source.

2.2. Spring Boot

Spring Boot is Spring's convention-over-configuration solution for creating stand-alone, production-grade Spring-based Applications that you can "just run".[23] It is preconfigured with the Spring team's "opinionated view" of the best configuration and use of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

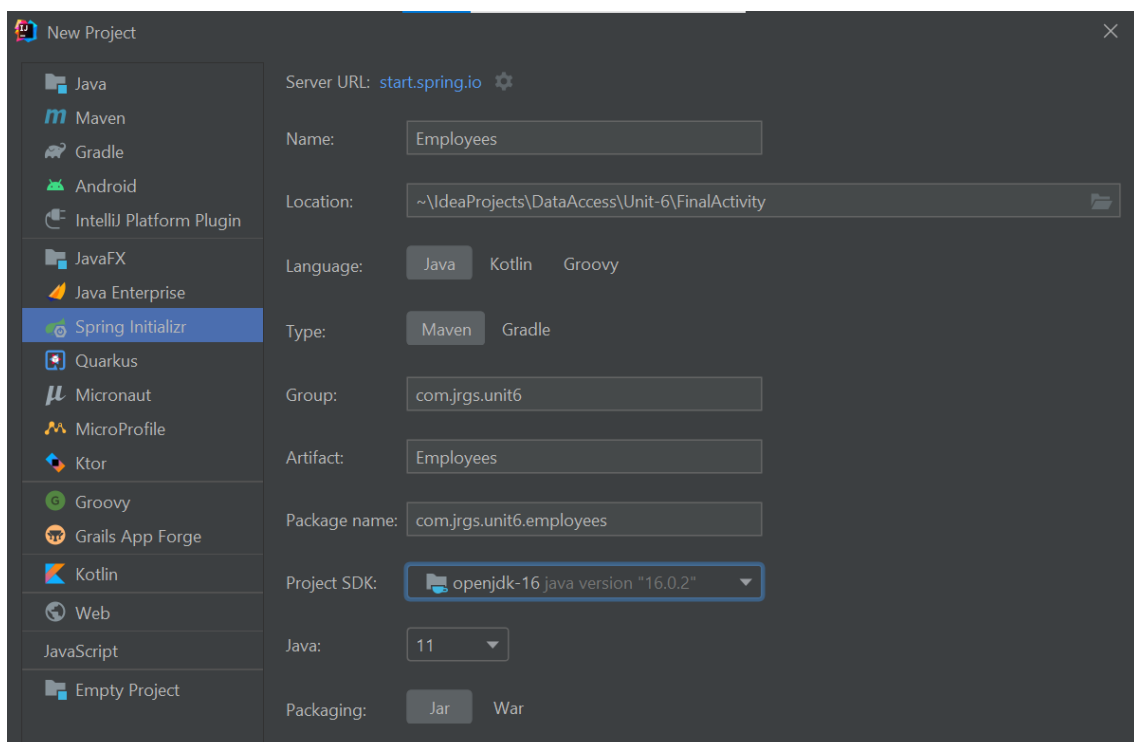
Key Features:

- Create stand-alone Spring applications
- Embed Tomcat or Jetty directly (no need to deploy WAR files)
- Provide opinionated 'starter' Project Object Models (POMs) to simplify your Maven/Gradle configuration
- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration.
- Smooth Integration and supports all Enterprise Integration Patterns.

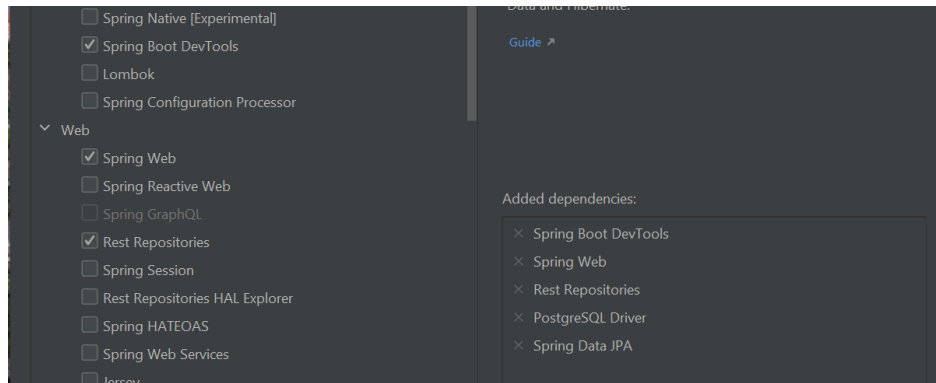
3. Creating a REST API with Spring Boot

3.1. Creating the project

The easiest way you can initialize a new Spring Boot project is by using [Spring Initializr](#), which automatically generates a skeleton Spring Boot project for you. You can use the web page or, alternatively, the same functionality in IntelliJ IDEA:



In the next screen the dependencies to be included in the project must be selected. In order to create a REST API, we should mark the following dependencies:



A tutorial will appear in IntelliJ. You can safely close it (or read it, if you like ;).

The dependencies included are:

- Spring Web: To include Spring MVC and embedded Tomcat into your project
- Spring Data JPA: Java Persistence API and Hibernate
- Spring Boot DevTools: Very useful development tools
- PostgreSQL Driver: JDBC Driver.

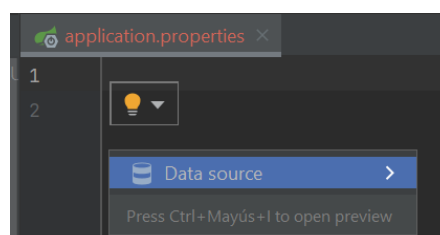
3.2. Connecting Spring Boot to the Database

Next, prior to start working on the application, we have to set up the connection parameters. This can *easily* be done through Spring Data JPA, which allows us to set this connection up with just a couple of parameters.

To tell Spring how to connect to your preferred database, in your *application.properties* file, you'll need to add some basic information:

```
application.properties
1 spring.datasource.username = postgres
2 spring.datasource.password = postgres
3 spring.datasource.driver-class-name = org.postgresql.Driver
4 spring.datasource.url = jdbc:postgresql://192.168.224.125:5432/employees
5 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
6 server.servlet.context-path=/employee-api-rest
```

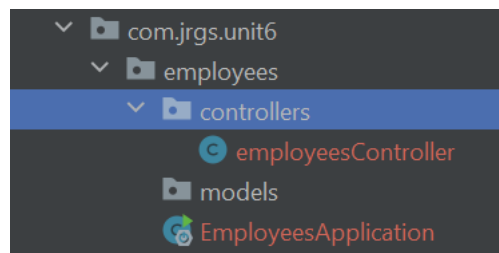
This information can also be automatically added by IntelliJ IDEA if the data source has been previously configured in the *database* window:



You can also specify the property `spring.jpa.hibernate.ddl-auto`. The `jpa.hibernate.ddl-auto` property directly influences the `hibernate.hbm2ddl.auto` property, and essentially defines how Hibernate should handle schema tool management. For production applications, this value is typically set to `none`, as dedicated personnel conduct management. In development, it's most common to use `update`, to allow the schema to be updated each time you restart the application, allowing you flexibility while working on development.

3.3. Setting up the MVC structure

Spring applications follows the MVC (model-view-controller architecture) but, by default, some necessary operations are not performed. So we will have to create a folder both for the controller and the model classes. Classes inside the folder `controller` will have the same suffix. The model classes can be generated with Hibernate, adding the required facet, as we saw in unit 5.



3.4. Creating repository classes

Spring repositories try to simplify the access to our data. We'll start implementing `CrudRepository`, an interface that offers the basic functionality to access every data entity of our model. So, in our example, having two data entities, we will have two repositories. These repositories are also called DAO (Data Access Objects).

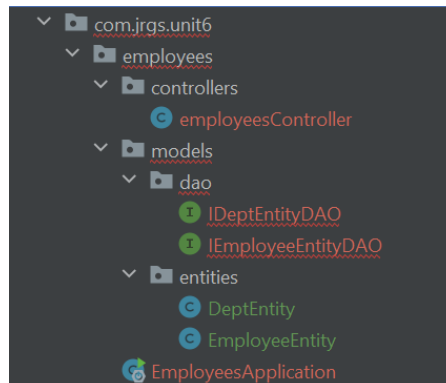
The easiest way to create our DAO's is from Hibernate POJOs, so, at this point, we will need to add the Hibernate facet and automatically create the POJOs (as we saw in Unit 5).

The main goal of a DAO is to perform CRUD operations on our data entities. To do this, we'll specify an interface that extends `CrudRepository`, and annotate it with `@Repository`.

```
@Repository
public interface EmployeeEntityDAO extends CrudRepository<EmployeeEntity, Integer> {
}
```

CrudRepository declares methods like *findAll()*, *findOne()*, and *save()* which constitute the basic CRUD functionality of a repository. You can use this *EmployeeEntityDAO* as-is, to perform CRUD operations on data entities now, with no further setup required, it's set up automatically to help you bootstrap some basic functionality. You can override some of this behavior, if you'd like, though.

The current look of our project should be something like that (I have created separate folder for DAOs and entities):



3.5. Creating the controller

Finally, we must create a controller (we have done it in point 3.3, actually), where we implement the actual logic of processing information, and use the components from the Persistence Layer (repositories), alongside the Model to store data.

Our goal is to create a REST API, so we will mark our controller as a *@RestController*, and add a *@RequestMapping* to it. *@RestController* is just a combination of *@Controller* and *@ResponseBody*, which means that instead of rendering pages, it'll just respond with the data we've given it. This is natural for REST APIs - returning information once an API endpoint has been hit.

Let's go ahead and take a look to the controller:

```
@RestController
@RequestMapping("/api-rest/Employees")
public class EmployeesController {

    @Autowired
    private EmployeeEntityDAO employeeDAO;

    @GetMapping
    public List<EmployeeEntity> findAllUsers() {
        //Implement
    }
}
```

```
@GetMapping("/{id}")
public ResponseEntity<EmployeeEntity> findUserById(@PathVariable(value = "id") int id) {
    //Implement
}

@PostMapping
public EmployeeEntity saveUser(@Validated @RequestBody EmployeeEntity employee) {
    //Implement
}
```

We've *@Autowired* our *EmployeeEntityDAO*. It's used for dependency injection, as the repository class is a dependency here. We've also used the *@GetMapping* and *@PostMapping* annotations to specify which types of HTTP requests our methods are accepting and handling. These are derived variants of the *@RequestMapping* annotation, with a *method = RequestMethod.METHOD* set for the respective types.

Let's start off with the implementation for the *findAll()* endpoint:

```
@GetMapping
public List<EmployeeEntity> findAllEmployees() {
    return (List<EmployeeEntity>) employeeDAO.findAll();
}
```

This method just calls the *employeeDAO* to *findAll()* users, and returns the list as the response.

Next, let's implement the endpoint to get each employee by their *id*:

```
@GetMapping("/{id}")
public ResponseEntity<EmployeeEntity> findEmployeeById(@PathVariable(value = "id") int id) {
    Optional<EmployeeEntity> employee = employeeDAO.findById(id);

    if(employee.isPresent()) {
        return ResponseEntity.ok().body(employee.get());
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

An employee with the given *id* might not be present in the database, so we wrap the returned *EmployeeEntity* in an *Optional*. Then, if the *employee.isPresent()*, we return a *200 OK* HTTP response and set the *EmployeeEntity* instance as the body of the response. Else, we return a *ResponseEntity.notFound()*-HTTP response *404*-.

Finally, let's make an endpoint to save employees:

```
@PostMapping
public EmployeeEntity saveEmployee(@Validated @RequestBody EmployeeEntity employee) {
    return employeeDAO.save(employee);
}
```

The `save()` method from the employee repository saves a new employee if it doesn't already exist. If the employee with the given `id` already exists, it's updated. If successful, it returns the persisted employee, including the auto-generated keys. The `@Validated` annotation is a validator for the data we provide about the employee and enforces basic validity. If the employee info is not valid, the data isn't saved. Also, the `@RequestBody` annotation maps the body of the `POST` request sent to the endpoint to the `EmployeeEntity` instance we'd like to save.

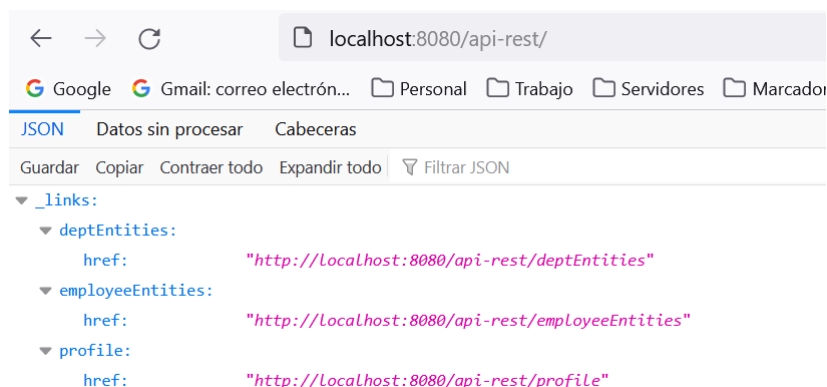
To complete the task, we should create another controller for `DeptEntity`.

3.6. Running and testing the REST API

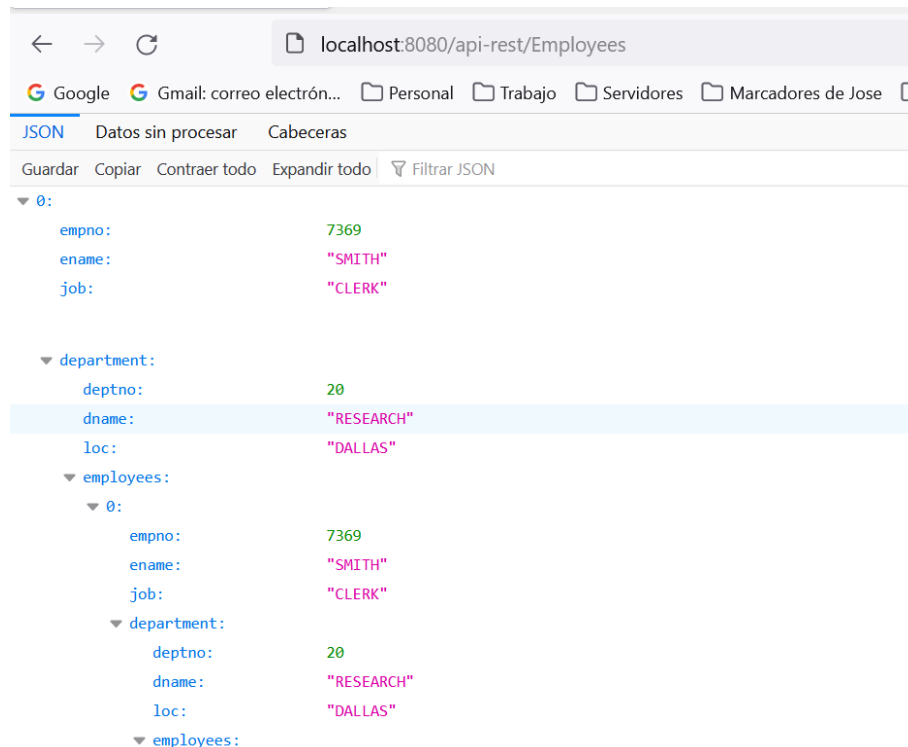
Once the previous tasks are done, we should be able to successfully execute our REST API. If everything is ok, IntelliJ IDEA will inform you that you have a Tomcat server listening on port 8080:

```
: Initialized JPA EntityManagerFactory for persistence unit 'default'
: spring.jpa.open-in-view is enabled by default. Therefore, database queries will always be processed in web thread by Spring.
: LiveReload server is running on port 35729
: Tomcat started on port(s): 8080 (http) with context path ''
: Started EmployeesApplication in 5.729 seconds (JVM running for 6.841)
```

You can check it opening a web browser and typing the URL: `localhost:8080/{name-of-your-api}`.



We see all the end points available in our REST API, we can use the links provided or type our mappings:



If we take a closer look to the JSON set returned, we will notice that we are facing a problem: as the employees belong to a department, when showing the information of a department SPRING try also to show the employees belonging to it, entering an infinite loop.

To solve this, we use the annotation `@JsonIgnoreProperties`. With `@JsonIgnoreProperties`, we instruct Spring to not include a specific property in the final JSON set returned. So, to deal with this issue, we can do this in `DeptEntity`:

```
@OneToMany(mappedBy = "department")
@JsonIgnoreProperties("department")
public List<EmployeeEntity> getEmployees() { return employees; }
```

And also in `EmployeeEntity`:

```
@ManyToOne
@JoinColumn(name = "deptno", referencedColumnName = "deptno")
@JsonIgnoreProperties("employees")
public DeptEntity getDepartment() { return department; }
```

3.7. Completing our CRUD

So far, we have implemented the Create/Update and Read option of our CRUD repository. In order to finish it, we also need to implement the Delete option. So we will need to add this functionality to our controller, adding one endpoint more, corresponding to the DELETE request.

```
@DeleteMapping("/{id}")
public ResponseEntity<?> deleteEmployee(@PathVariable(value = "id") int id) {
    Optional<EmployeeEntity> employee = employeeDAO.findById(id);
    if(employee.isPresent()) {
        employeeDAO.deleteById(id);
        return ResponseEntity.ok().body("Deleted");
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

The *delete* endpoint only tests if the provided *id* exists, executing the deletion.

To conclude, if we want to manage creation and updating separately, we can use the PUT request for updating purposes, leaving the POST request just for creation. We will also have to take care of employee existence.

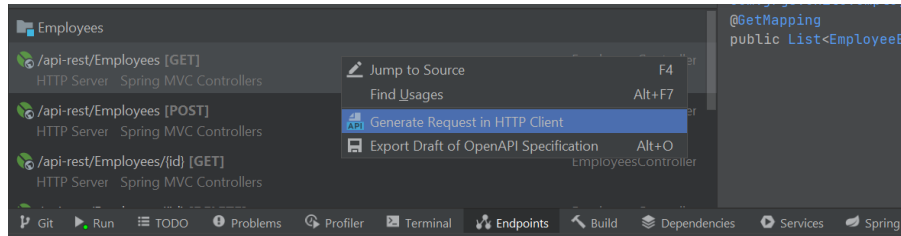
```
@PostMapping
public ResponseEntity<?> saveEmployee(@RequestBody Employee employee) {
    if (!employeeDAO.existsById(employee.getEmpno()))
        return ResponseEntity.ok(employeeDAO.save(employee));
    return ResponseEntity.badRequest().build();
}

@PutMapping("/{id}")
public ResponseEntity<?> updateEmployee(@RequestBody Employee newEmployee,
                                         @PathVariable(value = "id") int id) {
    if(employeeDAO.existsById(id)) {
        employeeDAO.save(newEmployee);
        return ResponseEntity.ok().body("Updated");
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

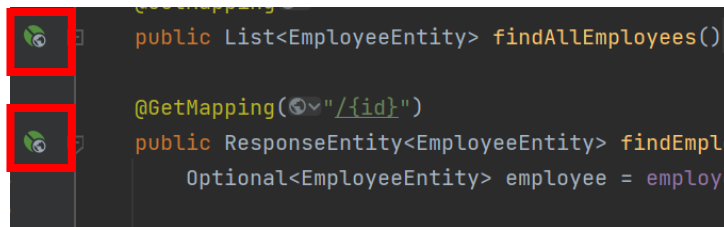
To test the endpoints, we have two options: use IntelliJ IDEA built-in functionality or use an an specific tool designed to test REST APIs, Postman. You can download Postman from <https://www.postman.com/downloads/>.

Using IntelliJ IDEA built-in functionality

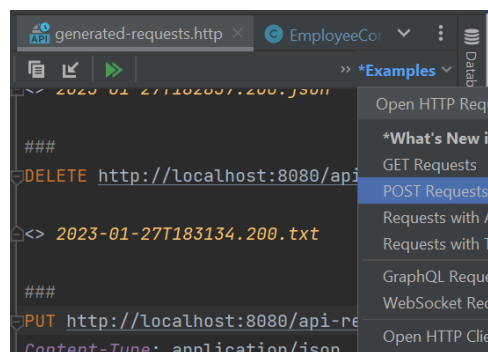
IntelliJ IDEA has all the necessary tools to test our REST API, although is not as powerful as Postman. You can check how many endpoints you have available in your REST API in the *Endpoints* tab.



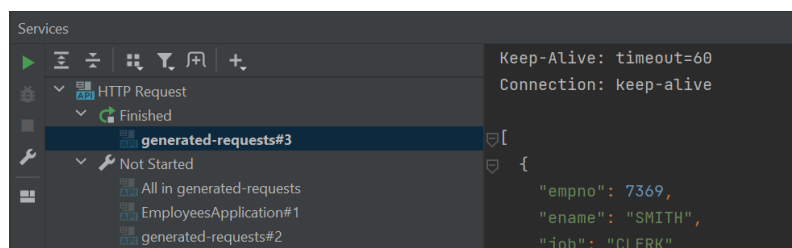
As it can be seen in the upper image, HTTP requests can be generated directly from IntelliJ IDEA, either using the *Endpoints* tab or directly, from the spring endpoint icon showed in the code.



Requests are generated in a file named *generated-requests.http*. This file is easily accessed (just by clicking in the icons and allows the user to individually execute every endpoint. You will also find an examples link that will help you typing the correct format of a POST or PUT request, for example.

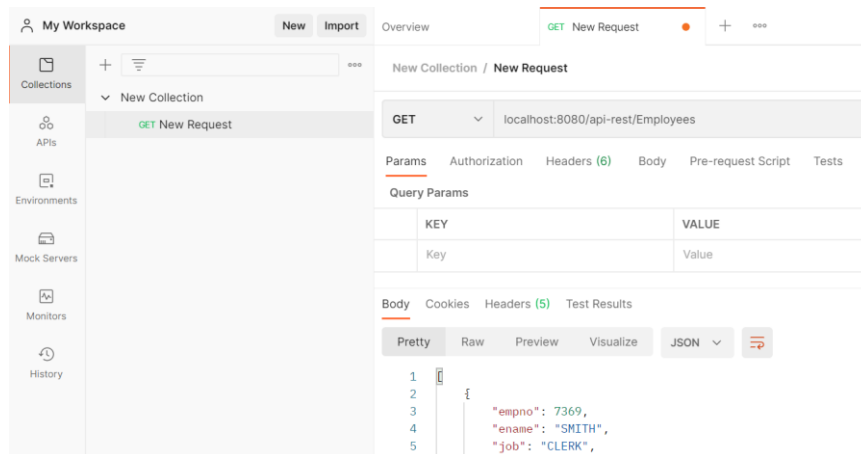


Results are showed in the *Services* tab.

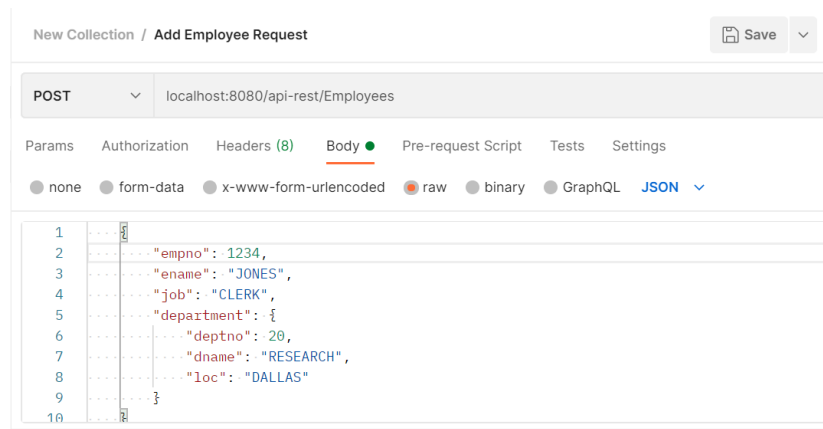


Using Postman

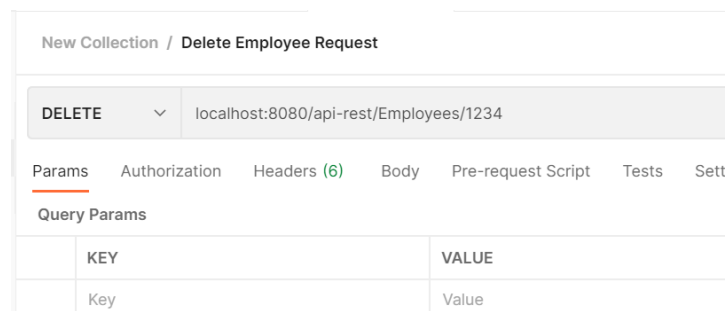
If you prefer to use Postman (is widely used for this purpose), firstly you must create (or access, if exists) your workspace and, inside this, create (and enter) a new collection. Inside a collection we can create as many requests as desired to test our API.



Before testing the DELETE request, we will create a request to (also) test the *save* feature. We will need to add a POST request, sending the information in JSON format:



If we've successfully created a new employee, we can now create a new DELETE request to test if our API is able to execute the deletions successfully:



Please note the following:

1. As *department* has been defined as an entity in *EmployeeEntity*, the full information of a department must be provided. This drawback can be bypassed by sending just the info corresponding to the primary key, as it can be seen in the image below:



2. To update a record we use exactly the same procedure we use to create a new record, changing the desired data.
3. It's not possible to create a new department when we create an employee, the department must be previously created.

3.8. Data Transfer Objects (DTOs)

As we have seen in the previous points, if we use Hibernate's functionality to handle, instead of foreign keys, a reference to the class that references that key, we may encounter issues such as infinite loops (which we solve by using the *@JsonIgnoreProperties* annotation) or having to send the entire class in the request instead of just the value of the foreign key (for example, the department code), even though we can simplify the sending by only giving a value to the key. That's why it's quite common for entities to be an exact reflection of the database tables, without incorporating any additional fields, as we can see below.

```
@Entity
@Table(name = "employee", schema = "public", catalog = "Employees")
public class EmployeeEntity {
    private int empno;
    private String ename;
    private String job;
    private int deptno;
```

However, this raises another issue: What do we do if we need to provide the user with information from more than one table, or simply send them not the entire table, but only a part of it? In Spring, to solve this, we have DTOs (Data Transfer Objects).

A DTO is nothing more than a POJO (Plain Old Java Object, i.e., a class with attributes and setters/getters) that is dynamically filled with the necessary information when accessing the corresponding endpoint.

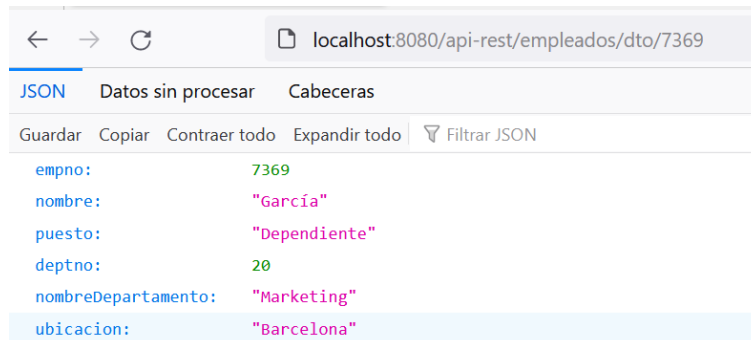
Let's imagine we are using the entity from the previous image and we want to include (as we did previously) the department information with each of the employees (not just the department number). To do this, we should first define the POJO that will contain the information we need:

```
public class EmpleadosDTO {  
    private int empno;  
    private String nombre;  
    private String puesto;  
    private int depno;  
    private String departamentoNombre;  
    private String departamentoUbicacion;  
}
```

Once we have created the POJO (in this case, annotations are not necessary as it doesn't match to any database table), we will create a new endpoint, with a specific mapping (/dto/{id}), where we will provide the information contained in the DTO:

```
@GetMapping("/dto/{id}")  
public ResponseEntity<EmpleadosDTO> buscarEmpleadoDTOPorId(@PathVariable(value = "id") int id) {  
    Optional<EntidadEmpleados> employee = empleadosDAO.findById(id);  
  
    if(employee.isPresent()) {  
        Optional<EntidadDepartamentos> departamento =  
            departamentosDAO.findById(employee.get().getDepartamento().getDepno());  
  
        EmpleadosDTO empleadosDTO = new EmpleadosDTO();  
        empleadosDTO.setEmpno(employee.get().getEmpno());  
        empleadosDTO.setNombre(employee.get().getNombre());  
        empleadosDTO.setPuesto(employee.get().getPuesto());  
        empleadosDTO.setDepno(employee.get().getDepartamento().getDepno());  
        empleadosDTO.setDepartamentoNombre(departamento.get().getNombre());  
        empleadosDTO.setDepartamentoUbicacion(departamento.get().getUbicacion());  
  
        return ResponseEntity.ok().body(empleadosDTO);  
    } else {  
        return ResponseEntity.notFound().build();  
    }  
}
```

As it can be seen, the structure is similar to the GET for the employee, but once we have retrieved the employee information, we access the department information (using the corresponding DAO we added to the controller, *departmentDAO*), and create a new DTO with information from both the employee and the department. If we now access our new endpoint, we should get the following result:



empno:	7369
nombre:	"García"
puesto:	"Dependiente"
deptno:	20
nombreDepartamento:	"Marketing"
ubicacion:	"Barcelona"


However, in the case where our DTO includes many fields, manually mapping it can be somewhat tedious. In this scenario, it may be a good idea to use the `ModelMapper` class, which will automatically map the fields of the entity to the DTO:

```
if(employee.isPresent()) {
    Optional<EntidadDepartamentos> departamento =
        departamentosDAO.findById(employee.get().getDepartamento().getDepno());

    ModelMapper mapper = new ModelMapper();
    EmpleadosDTO empleadosDTO = mapper.map(employee.get(), EmpleadosDTO.class);
    mapper.map(departamento.get(), empleadosDTO);

    return ResponseEntity.ok().body(empleadosDTO);
}
```

In this case, as the DTO has fields from two different tables (employees and departments), we should make two mappings, one for the employee entity, from which we get an `EmpleadoDTO` instance, and a second mapping of the department using this instance. Since both entities (`EmployeeEntity` and `DepartmentEntity`) have a field with the same name (the name of the employee and the name of the department), after the second mapping has been done we can get the following error:



empno:	7369
nombre:	"Marketing"
puesto:	"Dependiente"
deptno:	20
departamentoNombre:	"Marketing"
departamentoUbicacion:	"Barcelona"

As we can see, having the same name for the fields `employees.name` and `department.name`, the second mapping causes the employee's name assigned in the first mapping to be overwritten. This can be resolved by either changing the field name in one of the two entities or adding an exclusion rule in the mapping:

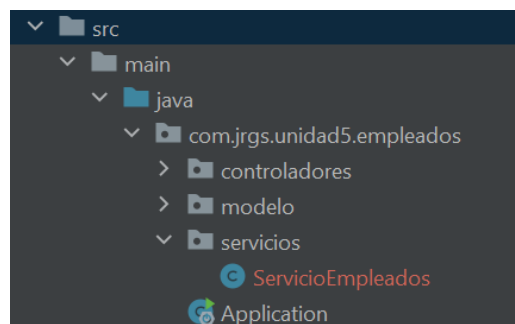
```
ModelMapper mapper = new ModelMapper();
EmpleadosDTO empleadosDTO = mapper.map(employee.get(), EmpleadosDTO.class);
mapper.typeMap(EntidadDepartamentos.class, EmpleadosDTO.class).
    addMappings( mapping -> mapping.skip(EmpleadosDTO::setName) );
mapper.map(departamento.get(), empleadosDTO);
```

As we can observe, after performing the first mapping, we instruct the *ModelMapper* class not to map the field *EmpleadosDTO.setName*, thus avoiding the previous overwriting.

3.9. The service layer.

In the previous section, we saw how we can handle data that does not exactly match the content of the tables using DTOs. Although it is not strictly necessary -especially in those APIs that only offer basic CRUD operations on the tables, as we have seen so far- when developing a Spring project, it is recommended to separate the business logic into a different layer, the service layer.

The service layer would be a layer positioned between the controller (which would be only responsible for handling requests) and the repository (which would be only responsible for accessing the database). Following the hierarchical division we have made so far, we would create services in a separate folder:



Our service class (*ServicioEmpleados*) would contain all the possible business logic of our application (including DTOs handling), as we can see in the following image:

```
12 @Service
13 public class ServicioEmpleados {
14     @Autowired
15     private IEmpleadosDAO empleadosDAO;
16
17     public List<EntidadEmpleados> buscarEmpleados() {
18         return (List<EntidadEmpleados>) empleadosDAO.findAll();
19     }
20
21     public EntidadEmpleados buscarEmpleadoPorCodigo(int id) {
22         Optional<EntidadEmpleados> empleado = empleadosDAO.findById(id);
23         return empleado.isPresent() ? empleado.get() : null;
24     }
25 }
```

Our controller will be limited to handle the web requests:

```
12 @RestController
13 @RequestMapping("/api-rest/empleados")
14 public class ControladorEmpleados {
15
16     @Autowired
17     ServicioEmpleados servicioEmpleados;
18
19     @GetMapping
20     public List<EntidadEmpleados> buscarEmpleados() {
21
22         return servicioEmpleados.buscarEmpleados();
23     }
24 }
```

3.10. Validating data

Once of the problems we are very likely to face when developing our REST API is the one related to non-valid data, that is, an empty name or an invalid code or email, for example. So, how do we deal with that?

To face the data validation management inside Spring, we should add a dependency to our *pom.xml* file:

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

This module will allow us to use annotations to validate the information received. These annotations are contained in the module *javax.validation.constraints*. Some example of annotations could be the following:

```
@Basic
@NotEmpty(message = "The name cannot be blank")
@Size(min = 2, max = 10, message = "Name size must be between 2 and 10")
@Column(name = "ename", nullable = true, length = 10)
public String getEname() { return ename; }
```

Some others useful annotations are:

- *@NotNull*: to say that a field must not be null.
- *@NotEmpty*: to say that a list field must not empty.
- *@NotBlank*: to say that a string field must not be the empty string (i.e. it must have at least one character).

- *@Min* and *@Max*: to say that a numerical field is only valid when it's value is above or below a certain value.
- *@Pattern*: to say that a string field is only valid when it matches a certain regular expression.
- *@Email*: to say that a string field must be a valid email address.

The full list can be found here: <https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html>.

In the example above, we have defined the restrictions to be validated over the DAOs (or DTOs). In this case, we can use the *@Valid* annotation to verify that the whole object matches our restrictions:

```
@PostMapping
public ResponseEntity<> saveEmployee(@Valid @RequestBody EmployeeEntity employee) {
    if (!employeeEntityDAO.existsById(employee.getEmpno()))
        return ResponseEntity.ok(employeeEntityDAO.save(employee));
    return ResponseEntity.badRequest().build();
}
```

Alternatively, we can use the *@Validated* annotation. The *@Validated* annotation is useful when you need to make specific validation in service/controller methods:

```
@PutMapping("/{id}")
public ResponseEntity<> updateEmployee(@Validated @RequestBody EmployeeEntity newEmployee,
                                       @NotNull @Min(0) @PathVariable(value = "id") int id) {
    if (employeeEntityDAO.existsById(id)) {
```

When we try to execute a method annotated with *@Validated* not matching the requisites, we will get a *BadRequest* response:

```
{
  "empno": 1234,
  "ename": "SMITHEEREEN AND SONS",
  "job": "CLERK",
  "department": {
    "deptno": 20,
    "name": "RESEARCH"
  }
}
```

```
{
  "defaultMessage": "Name size must be between 2 and 10",
  "objectName": "employeeEntity",
  "field": "ename",
  "rejectedValue": "SMITHEEREEN AND SONS",
  "bindingFailure": false,
}
```

3.11. Activities

- a. Make the necessary changes to your REST API to support the services layer.
- b. Create a DTO with the following information: department code, name, location, number of employees and code of the manager (if exists) or the president of the company.
- c. Add an endpoint for your DTO.
- d. Make the necessary validations for both the entities and the parameters in your employees REST API.

4. Consuming REST: client side

So far we have seen how to setup a REST API to deliver the data in our DBMS through a web server, using HTTP requests and JSON. But, how do we consume this data?

4.1. JAVA client

JSON file format

JSON (JavaScript Object Notation) is an open standard file format for sharing data that uses human-readable text to store and transmit data. JSON files are stored with the .json extension. JSON requires less formatting and is a good alternative for XML. JSON is derived from JavaScript but is a language-independent data format. The generation and parsing of JSON is supported by many modern programming languages. *application/json* is the media type used for JSON.

JSON data is written in key/value pairs. The key and value are separated by a colon(:) in the middle with the key on the left and the value on the right. Different key/value pairs are separated by a comma(.). The key is a string surrounded by double quotation marks for example "name". The values can be of the following types.

- Number
- String: Sequence of Unicode characters surrounded by double quotation marks ("Car").
- Boolean: True or False.
- Array: A list of values surrounded by square brackets, for example

```
[ "Red", "Green", "Blue" ]
```

- Object: A collection of key/value pairs surrounded by curly braces, for example:

```
{"empno": 1234, "ename": "John", "job" : "CLERK"}
```

Following you have the JSON data returned by our endpoint *departments/10*:

```
{
  "deptno":10,
  "dname":"ACCOUNTING",
  "loc":"NEW YORK",
  "employees":[
    {
      "empno":7782,
      "ename":"CLARK",
      "job":"MANAGER"
    },
    {
      "empno":7839,
      "ename":"KING",
      "job":"PRESIDENT"
    },
    {
      "empno":7934,
      "ename":"MILLER",
      "job":"CLERK"
    }
  ]
}
```

In this JSON data, we have two kind of objects (*department* and *employee*) and one array (*employees*).

GET Request

If our REST API is integrated within a web site, we can develop dynamic pages (using, for example, AJAX) to interact with the users. We can also, though, use JAVA desktop applications to consume from the REST. In the example below we have a very simple application that retrieves the department list from our REST API, using JAVA libraries both to make an HTTP request and deal with JSON data.

```
import org.json.JSONArray;
import org.json.JSONObject;

import java.net.HttpURLConnection;
import java.net.URL;
```



```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        GetRequest();
    }
    public static void GetRequest() {
        HttpURLConnection conn = null;
        try {
            URL url = new URL(
                "http://localhost:8080/api-rest/departments");
            conn = (HttpURLConnection) url.openConnection();
            conn.setRequestProperty("Accept", "application/json");

            if (conn.getResponseCode() == 200) {
                Scanner scanner = new Scanner(conn.getInputStream());
                String response = scanner.useDelimiter("\\Z").next();
                scanner.close();

                JSONArray jsonArray = new JSONArray(response);
                for (int i = 0; i < jsonArray.length(); i++) {
                    JSONObject jsonObject = (JSONObject)
                        jsonArray.get(i);
                    System.out.println(jsonObject.get("deptno") + " "
                        + jsonObject.get("dname"));
                }
            }
            else
                System.out.println("Connection failed.");
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
        finally {
            if (conn != null)
                conn.disconnect();
        }
    }
}
```

The *JSONArray* and *JSONObject* allow to read the corresponding JSON data. In the previous example, we get the data returned by the endpoint *departments* (an array of departments), reading it with a *JSONArray*. Each department inside the array is read using a *JSONObject*. To be able to use this classes, the following Maven dependency may be necessary:

```
<dependency>
  <groupId>com.vaadin.external.google</groupId>
  <artifactId>android-json</artifactId>
```

```
<version>0.0.20131108.vaadin1</version>
</dependency>
```

POST Request

When doing a POST request, a JSON information must be sent to the REST API with our request. For example, if we want to create a new employee (as we did with Postman in page 14, we will have to send the following JSON string:

```
{
  "empno": 1234,
  "ename": "JONES",
  "job": "CLERK",
  "department": {
    "deptno": 20,
    "dname": "RESEARCH",
    "loc": "DALLAS"
  }
}
```

To do this, we have to add to our connection the JSON string, as shown below:

```
public static void PostRequest() {
    HttpURLConnection conn = null;
    String jsonString = new JSONObject()
        .put("empno", "1234")
        .put("ename", "JONES")
        .put("job", "CLERK")
        .put("department", new JSONObject()
            .put("deptno", 20)
            .put("dname", "RESEARCH")
            .put("loc", "DALLAS")
            .toString()).toString();

    try {
        URL url = new URL("http://localhost:8080/api-rest/Employees");
        conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type", "application/json; utf-8");
        conn.setRequestProperty("Accept", "application/json");
        conn.setDoOutput(true);

        try (OutputStream os = conn.getOutputStream()) {
            byte[] input = jsonString.getBytes("utf-8");
            os.write(input, 0, input.length);
        }

        if (conn.getResponseCode() == 200)
```

```
        System.out.println("Employee inserted");
    } else {
        System.out.println("Connection failed");
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
} finally {
    if (conn != null)
        conn.disconnect();
}
}
```

DELETE Request

Finally, in order to delete using our REST API, we can use the following code:

```
public static void DeleteRequest(String codeToDelete) {
    HttpURLConnection conn = null;
    try {
        URL url = new URL("http://localhost:8080/" +
            "api-rest/Employees/" + codeToDelete);
        conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("DELETE");

        if (conn.getResponseCode() == 200)
            System.out.println("Employee deleted");
        else
            System.out.println("Connection failed");
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
    finally {
        if (conn != null)
            conn.disconnect();
    }
}
```

4.2. Thymeleaf web application

Thymeleaf is a Java-based framework that implements an HTML template engine for building dynamic web pages. In other words, it's a technology that, given an HTML template and a data source, allow the user to obtain a new page built with that data. Comparing it to other similar technologies, Thymeleaf serves the same purpose as JSP (Java Server Pages) or ASP (Active Server Pages). In this section,

we will see how to create a basic CRUD (Create, Read, Update, Delete) with Spring Boot and Thymeleaf.

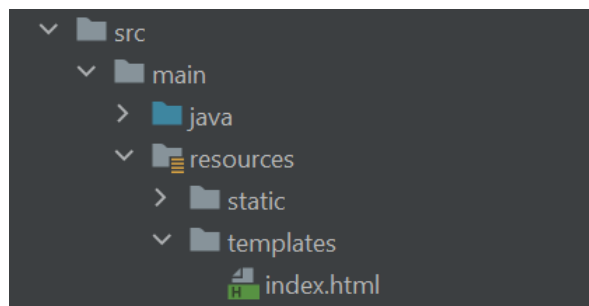
Definition of the main controller

The first step in creating our web application with Thymeleaf is to create a controller that processes the requests received by the application. Obviously, the first request that our controller must process will be the request for the main page of the application (`index.html`).

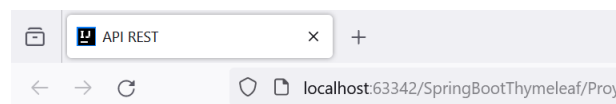
```
@Controller
public class ViewController {

    @GetMapping("/")
    public String index() { return "index"; }
```

In the image, we can see that our controller (which is no longer a REST controller) handles requests made to the root of our web application, returning the name of the page to be displayed (`index.html`). This page should be created in the resources section of our application, specifically in the templates section (`resources/templates`). Here, we will need to create any of the pages that will serve as templates for generating our dynamic pages.



As a main page, a simple HTML page has been created that offers four options: viewing the list of employees, viewing the list of departments, creating an employee, and creating a department.

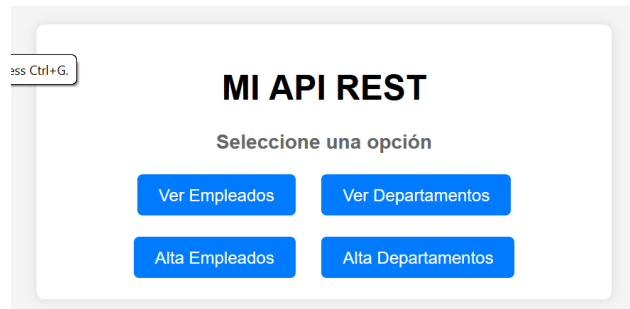


MI API REST

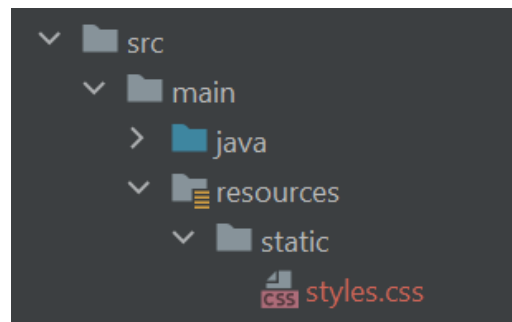
Seleccione una opción

[Ver Empleados](#) [Ver Departamentos](#)
[Alta Empleados](#) [Alta Departamentos](#)

We can add some CSS styling to make the application visually more attractive.



The CSS file must be created in the static resources section.



Within this main page, we need to redirect to four different links that show each of the options displayed. The next step will be to create the pages that respond to those requests.

Creation of templates with Thymeleaf

The web page we created before is a static page, meaning it will always display the same content. However, if we want to create a web page that displays a list of employees, we can't create a static page because the list of employees may vary. In this case, we need to use Thymeleaf. Let's start with the page to display a list of departments. The content of our `index.html` page should be similar to this:

```
<a href="http://localhost:8080/verdepartamentos" class="button">Ver Departamentos</a>
```

The first step would be to make our controller process this request:

```
@GetMapping("/verdepartamentos")
public String mostrarDepartamentos(Model model) {
    return "verdepartamentos";
}
```

We have added a method that receives the request and returns the page to be displayed (`verdepartamentos.html`). Now, this page, for the moment, is static

since we are not providing the information of the departments. To do this, we will use the `model` parameter.

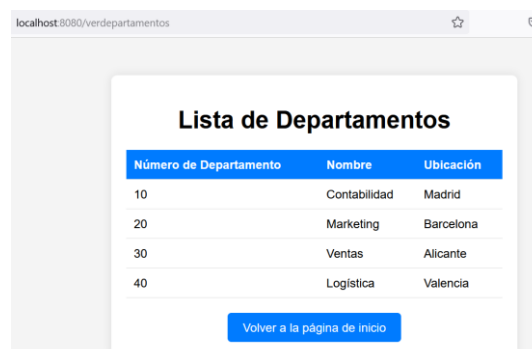
```
@GetMapping("/verdepartamentos")
public String mostrarDepartamentos(Model model) {
    List<EntidadDepartamentos> departamentos = (List<EntidadDepartamentos>) departamentosDAO.findAll();
    model.addAttribute("departamentos", departamentos);
    return "verdepartamentos";
}
```

The `Model` class is a class that allows the exchange of information between the controller and the views. All the information we include as an attribute in the model can be later used in the view (the HTML page). As you can see, an attribute can be a simple data type (an integer or a string) or any other object. In our case, we are using the same `EntidadDepartamentos` class that was used with the REST API, and the same DAO (`departamentosDAO`).

To use this model information within our HTML page, we must use specific *Thymeleaf* tags:

```
<table>
  <thead>
    <th>Número de Departamento</th>
    <th>Nombre</th>
    <th>Ubicación</th>
  </thead>
  <tbody>
    <tr th:each="departamento : ${departamentos}">
      <td th:text="${departamento.depno}"></td>
      <td th:text="${departamento.nombre}"></td>
      <td th:text="${departamento.ubicacion}"></td>
    </tr>
  </tbody>
</table>
```

In our example, we have created a table with the department number, name, and location. The `th:each` tag allows me to iterate over the list of departments that was previously added to the model (`${departamentos}`). Then, with the `th:text` tag, the corresponding information of each department will be displayed. The result will be something like this:



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/verdepartamentos'. The main content area has a title 'Lista de Departamentos' above a table. The table has three columns: 'Número de Departamento', 'Nombre', and 'Ubicación'. It contains four rows of data. Below the table is a blue button labeled 'Volver a la página de inicio'.

Número de Departamento	Nombre	Ubicación
10	Contabilidad	Madrid
20	Marketing	Barcelona
30	Ventas	Alicante
40	Logística	Valencia

Just as we did in the REST API, these requests can be improved by including parameters to allow a more selective searching.

Activity

1. Add the necessary code to provide the 'view employees' option in the application.

Sending Data: POST Requests

To complete our application, let's include the option to create a new department. In this case, we'll need to have a form to be able to make POST requests. The HTML code of our `altadepartamento.html` page should be similar to the following:

```
<h1>Crear Nuevo Departamento</h1>
<form action="#" th:action="@{/altadepartamento}" th:object="${departamento}" method="post">
  <div class="form-group">
    <label for="depno">Número de Departamento:</label>
    <input type="text" id="depno" name="depno" required>
  </div>
  <div class="form-group">
    <label for="nombre">Nombre:</label>
    <input type="text" id="nombre" name="nombre" required>
  </div>
  <div class="form-group">
    <label for="ubicacion">Ubicación:</label>
    <input type="text" id="ubicacion" name="ubicacion" required>
  </div>
  <button class="form_button" type="submit">Crear Departamento</button>
</form>
```

As it can be seen, the only specific *Thymeleaf* tags in the form are those indicating the endpoint to call (`altadepartamento`) and the name of the object in the model (`departamento`). Within the controller, we would have two entry points: one that displays the empty form (corresponding to a GET request) and one that processes the POST request:

```
@GetMapping("/altadepartamento")
public String altaDepartamento(Model model) {
    model.addAttribute("departamento", new EntidadDepartamentos());
    return "altadepartamento";
}

@PostMapping("/altadepartamento")
public String crearDepartamento(@ModelAttribute EntidadDepartamentos departamento) {
    if (!departamentosDAO.existsById(departamento.getDepno()))
        departamentosDAO.save(departamento);
    return "altadepartamento";
}
```

As we can see, in the GET request, we add a new department entity to the model, which will be returned, conveniently filled, in the POST request. It should be noted that this same request can be used to edit an existing department (just remove the check that verifies that the department ID exists).

Improving Our Application

Although the provided code is functional, it would be interesting for the user to have a notification of whether the insertion operation has been successful or not. But how can we do this on a web page? One possible solution is to include user messages within the page using, once again, a *Thymeleaf* tag. To do this, we should have a DIV in our page that would be displayed depending on whether there is a message for the user or not.

```
<div th:if="${mensaje}">
  <p th:text="${mensaje}"></p>
</div>
```

This block would be displayed based on whether we have a message to show (`th:if`). We should add this message to the model within the corresponding event of the controller:

```
@PostMapping("/altadepartamento")
public String crearDepartamento(@ModelAttribute EntidadDepartamentos departamento, Model model) {
    if (!departamentosDAO.existsById(departamento.getDepno()))
    {
        departamentosDAO.save(departamento);
        model.addAttribute("mensaje", "Departamento creado correctamente");
    }
    else {
        model.addAttribute("mensaje", "Error al crear el departamento: clave");
    }
    return "altadepartamento";
}
```

Here, once the POST request is made, the message attribute is added to the model, and the page is displayed again, this time with the corresponding DIV visible, since the attribute `mensaje` exists and has a value. We could still improve it further. Ideally, we would display error messages in one color (for example, red) and confirmation messages in another (for example, green). This could be easily done with two different DIVs and two different messages as well. However, *Thymeleaf* also gives us the option to dynamically adjust the CSS class to which the DIV belongs. To do this, we should pass a new attribute to the model indicating the type of message to display:


```

if (!departamentosDAO.existsById(departamento.getDepno()))
{
    departamentosDAO.save(departamento);
    model.addAttribute("tipo_operacion", "ok");
    model.addAttribute("mensaje", "Departamento creado correctamente");
}
else {
    model.addAttribute("tipo_operacion", "error");
    model.addAttribute("mensaje", "Error al crear el departamento");
}

```

Now we have a new attribute, `tipo_operacion`, which indicates whether the operation was successful ("ok") or not ("error"). To display the DIV with different colors, first, we will create two CSS classes that set these properties:

```

.ok_message {
    background-color: forestgreen;
    color: antiquewhite;
}

.error_message {
    background-color: crimson;
    color: antiquewhite;
}

```

And then we will dynamically assign the class to the specific DIV using the attribute we passed in the model:

```

<div th:if="${mensaje}" th:class="${tipo_operacion+'_message'}">
    <p th:text="${mensaje}"></p>
</div>

```

Now, when we perform an insertion in the database, we will get the following result:

Ubicación:

Departamento creado correctamente

Activity

1. Add the necessary code to provide the 'employee creation' option in the application.

Parameter Passing

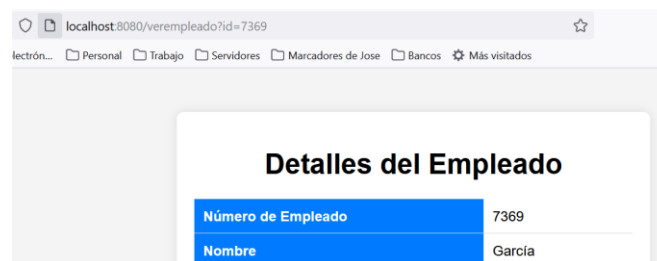
Just as we did in the REST API, when accessing our web application endpoints, we can use parameters. Let's see this through an example: suppose we have designed our page to view the list of employees, and instead of displaying all the data of each employee, we want to enable a link to view their details, roughly as we see in the image below:

Número de Empleado	Nombre	
7369	García	Detalles
7499	López	Detalles
7521	Pérez	Detalles
7566	Gómez	Detalles

To do this, on the page where we display the list of employees, we must dynamically generate the hyperlink for each employee:

```
<td><a th:href="${'/verempleado?id='+empleado.empno}"/>Detalles</a></td>
```

As we can see in the image, we use the `th:href` tag to generate the hyperlink to the `viewemployee` page, passing the `id` as a parameter. The result should be something similar to the following:



Obviously, in order to achieve this, we will have to have, on one hand, a new web page to view the details of an employee (`viewemployee.html`) and on the other hand the corresponding bean in our controller:

```
@GetMapping("/{verempleado}")
public String mostrarEmpleado(Model model, @RequestParam(name = "id", required = true) int id) {
    Optional<EntidadEmpleados> empleado = empleadosDAO.findById(id);
    if (!empleado.isPresent()) {
        model.addAttribute("titulo", "Error");
        model.addAttribute("mensaje", "No se encontro el empleado con el id " + id);
        return "error";
    }
    model.addAttribute("empleado", empleado.get());
    return "verempleado";
}
```

As we can see, this bean receives a parameter (*id*) and looks for the corresponding employee. If it doesn't find it, an error page is displayed -which should also be created- with the information provided in the model. In case of success, the employee information is displayed.

Activity

1. Create the viewemployees, viewemployee, and error pages.

Using Filters in Searches

A very common scenario we face when working with a database is to perform searches that meet certain criteria. Let's imagine, for example, that on our employee display page we want to add a filter to display only employees belonging to a specific department. How can we do it?

The first step would be to add to the model that should be used on the page the corresponding department information (the information we need to filter). We could directly take it from the set of employees, but we would miss important information such as, for example, the department name. Modifying the corresponding bean would be quite simple:

```
@GetMapping("/verempleados")
public String mostrarEmpleados(Model model) {
    List<EntidadDepartamentos> departamentos = (List<EntidadDepartamentos>) departamentosDAO.findAll();
    model.addAttribute("departamentos", departamentos);
}
```

Next, we would have to add this information to our page using some kind of control that allows us to make the selection. As an example, we will use a dropdown combobox:

```
<h1>Lista de Empleados</h1>
<select id="selectDepartamento" onchange="cambiarDepartamento()">
  <option value="">Selecciona un departamento</option>
  <option th:each="departamento : ${departamentos}"
    th:value="${departamento.depno}"
    th:text="${departamento.nombre}">
  </option>
</select>
```

We can see that we have added a couple of values to the combobox, the department number as the selected value (*th:value*) and the department name as the value to be displayed (*th:text*). Additionally, we have to add the necessary JavaScript code to update the page when a change occurs in the combo (*changeDepartment()*):

```

<script>
    function cambiarDepartamento() {
        var depno = document.getElementById("selectDepartamento").value;
        window.location.href = "/verempleados?depno=" + depno;
    }
</script>
</body>

```

In the script, we see that we call the *viewemployees* page using a parameter, the id of the department that has just been selected. We would need to incorporate this parameter and the corresponding logic into our bean:

```

@GetMapping("/verempleados")
public String mostrarEmpleados(Model model, @RequestParam(name = "depno", required = false) Integer depno) {
    List<EntidadDepartamentos> departamentos = (List<EntidadDepartamentos>) departamentosDAO.findAll();
    model.addAttribute("departamentos", departamentos);
    List<EntidadEmpleados> empleados;
    if (depno == null)
        empleados = (List<EntidadEmpleados>) empleadosDAO.findAll();
    else
        empleados = (List<EntidadEmpleados>) empleadosDAO.findByDepno(depno);
    model.addAttribute("empleados", empleados);
    return "verempleados";
}

```

As it can be seen, the parameter is optional. If it is not supplied, the list of all employees is obtained. If, on the contrary, it has a value, only the information corresponding to that department is retrieved. Here, obviously, we would need to declare the *findByDepno()* method in the DAO and perform the corresponding error handling. If everything has been done correctly, the final result would be something similar to this:



Activity

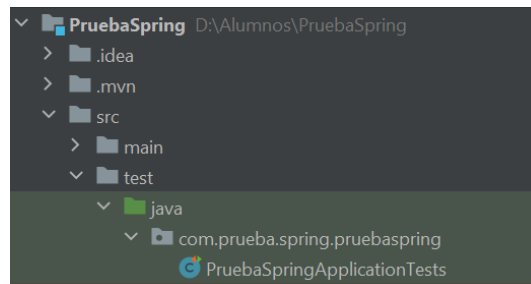
1. Incorporate error handling into our bean.
2. Add a second filtering criterion (filtering by job position).

5. Spring Boot testing

There are several ways to perform unit testing in a Spring Boot project. We can use the Spring Boot Testing Framework, that provides a set of annotations and tools that facilitate the writing of unit tests for REST controllers, or, alternatively,

we can use a very popular Java test framework, Mockito combined with JUnit. Let's take a look at both options.

Within our Spring Boot project, a *test* directory will have been created with a test class inside it.



The test class created by *Spring Initializr* will look similar to the following:

```
@SpringBootTest
class PruebaSpringApplicationTests {

    @Test
    void contextLoads() {
    }

}
```

Where the *@SpringBootTest* annotation indicates that the following class is a test class, containing each of the test methods (annotated with *@Test*). To test the behavior of a controller, we will use the *MockMvc* testing framework. *MockMvc* allows simulating HTTP requests to the controller and verifying the results of these requests. Suppose we want to write a test method for the GET request endpoint of our employees' controller. One possible solution would be as follows:

```
@SpringBootTest
class SpringApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testGetEmployees() throws Exception {
        // Simular una solicitud GET a la ruta /employees
        mockMvc.perform(get("/api/employees"))

        // Verificar que la respuesta tenga un código de estado 200
        .andExpect(status().isOk())
    }
}
```

```
// Verificar que la respuesta contenga una lista de employees
.andExpect(jsonPath("$.employees").isEmpty());
}
}
```

This code simulates a GET request to the `/api/employees` endpoint. Then, it verifies that the response has a status code of 200 (Ok) and contains a non-empty list of employees. This test simply checks that there is a response without verifying that the received response contains the information that should have been received. To create a more reliable test method, where we check that the received response truly corresponds to the information sent, we would need to improve our test method using Mockito:

```
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.springframework.http.ResponseEntity;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.*;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = Application.class)
class ApplicationTests {

    @Mock
    private IEmployeeDAO employeeDAO;

    @InjectMocks
    private EmployeeController employeeController;

    @Test
    void findAllEmployees() {
        // Arrange
        List<EmployeeEntity> employees = new ArrayList<>();
        employees.add(new EmployeeEntity(1000, "Nombre1", "Puesto1",
10));
        employees.add(new EmployeeEntity(1001, "Nombre2", "Puesto2",
20));

        when(employeeDAO.findAll()).thenReturn(employees);

        // Petición
```

```
List<EmployeeEntity> result =
    employeeController.findAllEmployees();

// Assert
assertEquals(2, result.size());
}
}
```

In this code, we've declared a test class (*ApplicationTests*) to test our Spring application (*Application*). We've annotated the test class with *@SpringBootTest*. To perform the test correctly, the testing framework (Mockito) needs access to both the model (*employeeDAO*) and the controller (*employeeController*). Finally, we've declared a test method (*@Test*) where we create two employee entities and add them to a list.

With the following command:

```
when(employeeDAO.findAll()).thenReturn(employees);
```

we're telling the testing framework that the response from our employee DAO for a *findAll()* request should be the list we've just created. Finally, we make the request to our controller and check that the size of the returned list corresponds to our list (we could also check each element individually).

As with any other type of test, we not only need to ensure that correct requests return the expected value but also that, when there is an invalid request, the appropriate error is returned. Suppose we want to expand our test class by adding test methods for searching by *ID*. In this case, we can either find the identifier (getting the employee with that *ID*) or it may not exist, in which case an error should be returned. These two checks would be performed with the following test methods:

```
@Test
void findEmployeesById() {
    int id = 1000;
    EmployeeEntity employee =
        new EmployeeEntity(id, "Nombre1", "Puesto1", 10);
    when(employeeDAO.findById(id))
        .thenReturn(Optional.of(employee));

    // Request
    ResponseEntity<EmployeeEntity> result =
        employeeController.findEmployeeById(id);

    // Assert
    assertEquals(200, result.getStatusCodeValue());
}
```

```
        assertEquals(employee, result.getBody());
    }

    @Test
    void findEmployeeByIdNotExists() {
        int id = 1099; // no existe en la BD

        when(employeeDAO.findById(id)).thenReturn(Optional.empty());

        // Request
        ResponseEntity<EmployeeEntity> result =
            employeeController.findEmployeeById(id);

        // Assert
        assertEquals(404, result.getStatusCodeValue());
    }
}
```

In the first case, we check that the request was successful (200) and that the JSON in the page body matches the searched employee. In the second case, we check that the expected error (404, not found) is returned.

The complete test class for each method of our controller can be found in the annex at the end of the document.

5.1. Activity

1. Make a test class for the department controller.

6. Security with Spring Boot

6.1. Introduction

So far, all the connections we've made to our REST API have used the HTTP protocol, through port 8080. However, there are two problems that we haven't considered until now:

- Access to our web is open to anyone with the URL. Therefore, at any given time, an intruder could have access to the data.
- The use of the HTTP protocol means that data transmission occurs without encryption. This implies that even if the URL were unknown or the access restricted, an intruder with access to the network could easily intercept the data.

To address these two issues, let's explore different options offered by Spring Boot.

6.2. Basic REST API Security

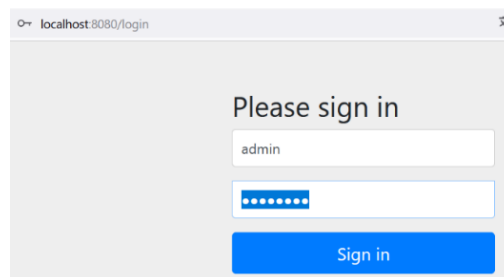
To make use of the security features in Spring Boot, it's necessary to include the following Maven dependency:

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

Once the POM file is reloaded, we can add the following lines of code to the *application.properties* file:

```
spring.security.user.name=admin
spring.security.user.password=password
```

where *admin* and *password* can be any values. If we rebuild our project and connect to the REST API now, we'll see that credentials are requested before allowing the connection:



In this way, non-allowed users are already banned. However, we are still using the HTTP protocol, making it easy for an attacker to intercept our credentials. The solution lies in using the HTTPS protocol.

6.3. Using HTTPS

Certificates

The difference between the HTTP and HTTPS protocols is that, in the first case, information travels over the internet in plain text, while in the second, a protocol (SSL, Secure Socket Layer) is used to encrypt this information before transmitting it. To perform this encryption, a key is required, stored in what we call SSL certificates. These certificates can be:

- **Self-signed certificates:** These certificates are not backed by any certification authority, meaning nobody can guarantee they correspond to

the claimed website. Therefore, web browsers often don't trust these certificates, displaying a warning when connecting to such websites. In our case and for production environments, having such a certificate should be enough. To generate a certificate, the *openssl* tool can be used:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem  
-days 365
```

In this command, a new 4096-bit RSA key has been generated, valid for one year (any number of days can be specified). When you run this command, you will be prompted for important details such as the organization name (alias) and the password to use for the connection. Don't forget these details; otherwise, the certificate will be useless.

- **Certificates issued by Let's Encrypt:** Let's Encrypt is a certification authority that provides free SSL/TLS certificates renewable every 90 days (includes a script for auto-renewal). Certificates can be obtained from the website or by using a client like *certbot*.
- **Commercial certificates (payment):** For production applications, commercial certificates obtained from trusted certificate providers are mostly used. These certificates are recognized by most browsers and security tools. Commercial certificates can be obtained from providers like Comodo, Symantec, GoDaddy, etc.

Using certificates in Spring Boot

Once we have the certificate, we need to perform some steps before being able to use it in our Spring Boot application. The first step is to transform the certificate (*cert.pem*) and the private key (*key.pem*) into the PKCS12 format. Again, we can use the *openssl* utility:

```
openssl pkcs12 -export -in cert.pem -inkey key.pem -out keystore.p12 -  
name el-alias -CAfile cert.pem -caname root
```

Where *the-alias* is the alias value we entered when generating the certificate. This command will create a PKCS12 file (*keystore.p12*) containing the certificate, the private key, and optionally, the certification chain (CAfile).

Once done, we need to add this certificate to our keystore:

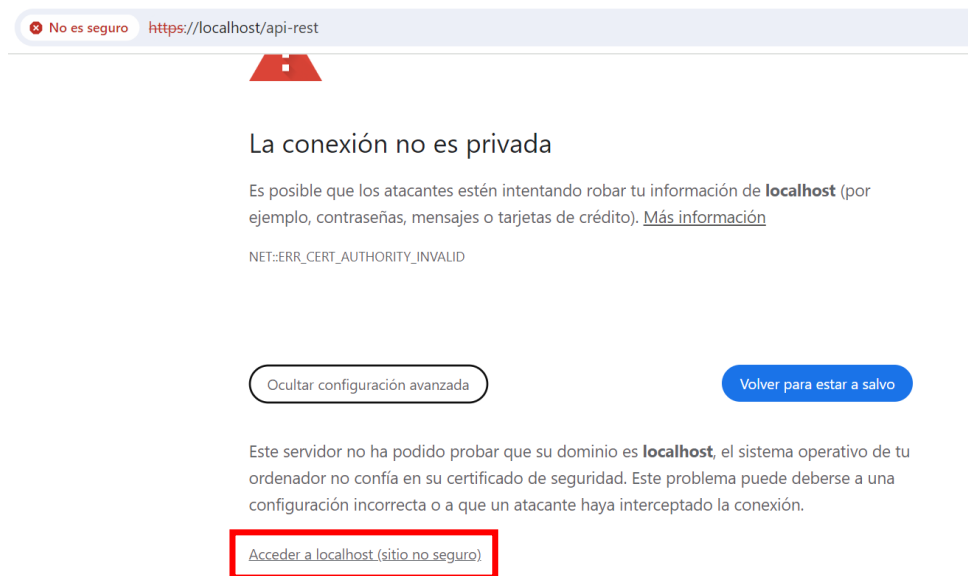
```
keytool -importkeystore -srckeystore keystore.p12 -srcstoretype PKCS12  
-destkeystore el-almacen-de-claves.jks -deststoretype JKS
```

This will generate a file (*the-key-store.jks*) that should be added to the *classpath* of our application (for example, copying it in the *resources* folder).

Finally, we just need to add the following lines to the *application.properties* file:

```
server.port=443
server.ssl.key-store=classpath:el-almacen-de-claves.jks
server.ssl.key-store-password=la-contraseña
server.ssl.key-alias=el-alias
```

If everything has been made properly, we can no longer connect via HTTP. Instead, we should access through HTTPS, and on the first connection, accept the certificate:



Role-based authentication

Although with the current configuration, our REST API is secure and does not allow access to unauthorized individuals, it is common in database access -and therefore, a REST API- that certain resources are available only to specific users or groups, known as roles. We can configure our REST API so that access to endpoints is based on the user's role.

To implement role-based authentication, in versions prior to Spring 2.7.0, we should extend the *WebSecurityConfigurerAdapter* class:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager manager =
            new InMemoryUserDetailsManager();
    }
}
```

```

        manager.createUser(User.withUsername("user")
            .password("{noop}password1").roles("USER").build());
        manager.createUser(User.withUsername("admin")
            .password("{noop}password2").roles("USER", "ADMIN").build());
        manager.createUser(User.withUsername("bdadmin")
            .password("{noop}password3").roles("USER", "BDADMIN")
                .build());
        return manager;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                // Employees tiene acceso abierto
                .antMatchers("/api/employees").permitAll()

                // departments y peticiones POST solo administradores
                .antMatchers("/api/departments")
                    .hasAnyRole("ADMIN", "BDADMIN")
                .antMatchers(HttpMethod.POST, "/api/employees")
                    .hasAnyRole("ADMIN", "BDADMIN")
                .anyRequest().authenticated()
                .and()
                .httpBasic();
    }
}

```

The first method (*UserDetailsService*) allows us to create the roles and users necessary to access each of the endpoints in our REST API. We can see that a user can have more than one role. In the second method (*configure*), we configure the HTTP request, checking access not only based on the URL (*.antMatchers("/api/departments")*) but also on the type of request (*.antMatchers(HttpMethod.POST)*). As seen in the example, it is even possible to allow access to an endpoint with one type of request and deny it with another.

Starting from Spring Security 5.7, it is recommended to use a component-based configuration instead of extending the *WebSecurityConfigurerAdapter* class. This provides greater flexibility and modularity in security configuration.

In this approach, separate beans are created for each aspect of the configuration, such as the authentication service and authorization constraints. Spring Security provides interfaces like *AuthenticationManager* and *WebSecurityCustomizer* to define configuration in a modular way. These

interfaces allow creating custom beans that implement the application-specific security logic. To use the security beans, they must be registered in the Spring context using the `@Bean` annotation or the `ApplicationContext.getBean()` method.

The `WebSecurityConfigurerAdapter` class, rewritten with the new security configuration, would look like this:

```
@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager manager =
            new InMemoryUserDetailsManager();
        manager.createUser(User.withUsername("user").
            password("{noop}password1").roles("USER")
                .build());
        manager.createUser(User.withUsername("admin").
            password("{noop}password2").roles("USER", "ADMIN")
                .build());
        manager.createUser(User.withUsername("bdadmin").
            password("{noop}password3").roles("USER", "BDADMIN")
                .build());
        return manager;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
        throws Exception {
        http.authorizeRequests()
            .requestMatchers("/apijrgs/employees")
                .permitAll()
            .requestMatchers("/apijrgs/departments")
                .hasAnyRole("ADMIN", "BDADMIN")
            .requestMatchers(HttpMethod.POST, "/employees")
                .hasAnyRole("ADMIN", "BDADMIN")
            .anyRequest().authenticated()
            .and()
            .httpBasic(withDefaults());
        return http.build();
    }
}
```

6.4. Securing the client side

In order to use HTTPS instead of HTTP, some changes are needed to be made to our code to be able to connect.

Firstly, URL's must be changed to use the secure protocol:

```
URL url = new URL("https://localhost/api-rest/departments");
```

In addition to this, we will use the *HttpsURLConnection* class instead of the *HttpURLConnection* class.

Secondly, if our server is using a self-signed certificate (or a non-trustable one), which is very usual when developing the application, we need to configure the application to trust in that certificate. We can do that using an instance of *TrustManager* and *SSLContext*, like in the following example:

```
private static SSLContext getSSLContext()
    throws NoSuchAlgorithmException, KeyManagementException {
// Build a SSLContext object with no hostname validation
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(null, new TrustManager[]{
    new X509TrustManager() {
        public void checkClientTrusted(java.security.cert.X509Certificate[] chain,
                                       String authType) {}
        public void checkServerTrusted(java.security.cert.X509Certificate[] chain,
                                       String authType) {}
        public java.security.cert.X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[]{};
        }
    }, null);

// Skip hostname validation
HttpsURLConnection.setDefaultHostnameVerifier((hostname, session) -> true);

return sslContext;
}
```

Note that, in the code above, the hostname validation has also been skipped, because, in development environments, is quite frequent to access the server via the IP address, instead of the DNS name. In production environments, this approach should be avoided, because of the risks involved.

Once we have build the *SSLContext*, we add it to our connection together with the credentials information:

```
String username = "admin";
String password = "password";

SSLContext sslContext = getSSLContext();
HttpsURLConnection.setDefaultSSLSocketFactory(sslContext.getSocketFactory());

// Credentials should be coded with Base64 algorithm
String credentials = username + ":" + password;
```

```
String encodedCredentials =
    Base64.getEncoder().encodeToString(credentials.getBytes());

conn = (HttpsURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setRequestProperty("Accept", "application/json");
// add the credentials information to the connection
conn.setRequestProperty("Authorization", "Basic " + encodedCredentials);
```

If everything is ok, we should be able to connect to our secured REST API. If you experience any problem, especially with POST requests, it could be necessary to disable the Cross Site Request Forgery (CSRF) protection in the server side:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/api-rest-jrgs2122/employees").permitAll()
        ... // specify here the ant matchers
        .and()
        .httpBasic();

    http.csrf.disable();
}
```

6.5. Activity

Secure the employees-department REST API using this guide.

7. REST API documentation

Nowadays, when we purchase any item, it often comes with an instruction manual that explains how to install it and provides usage recommendations. Similarly, when delivering a product like a REST API, we must include the corresponding documentation—developer-oriented—to guide its use. The documentation of an API is essential to ensure that users utilize it properly, indicating, among other things, the necessary data for interaction and the responses it provides.

For API documentation to be effective, it should include the following:

1. **Definition of Endpoints:** Endpoints are the channels the API offers for consuming its resources; all these channels must be properly listed.
2. **Supported HTTP Methods:** Each endpoint may support one or more HTTP methods, allowing different behaviors in each case.

3. **Accepted Parameters:** If the endpoint accepts parameters, they must be detailed, including information such as data types and whether they are mandatory.
4. **Request Responses:** For each request, the endpoint will return an HTTP response (200, 400, 401, etc.), depending on whether the request was successful, an error occurred, and so on.
5. **Authentication Methods:** If the API requires authentication (which is quite common), all available methods should be thoroughly described.

As an example of API documentation, you can refer to the Spotify API documentation: <https://developer.spotify.com/documentation/web-api>.

7.1. OpenAPI

The OpenAPI specification is a specification for describing, producing, consuming, and visualizing web services. Although it was originally developed to support Swagger (a set of tools for API developers), it was established as a separate project in 2015.

An OpenAPI description is a formal representation of an API that certain tools can use to generate code, documentation, and test cases, among other things. OpenAPI descriptions are text files written in JSON or YAML format. Below is an example of an OpenAPI description written in YAML format:

```
openapi: "3.0.0"
info:
  title: "SpringEmployeeExample API"
  description: "SpringEmployeeExample API"
  version: "1.0.0"
servers:
  - url: "https://SpringEmployeeExample"
paths:
  /api-rest/employees:
    get:
      summary: "Get the list of all the employees"
      operationId: "findAllEmployees"
      responses:
        "200":
          description: "The employee list"
          content:
            '*/*':
              schema:
                type: "array"
                items:
                  $ref: "#/components/schemas/Employee"
  /api-rest/employees/{id}:
```



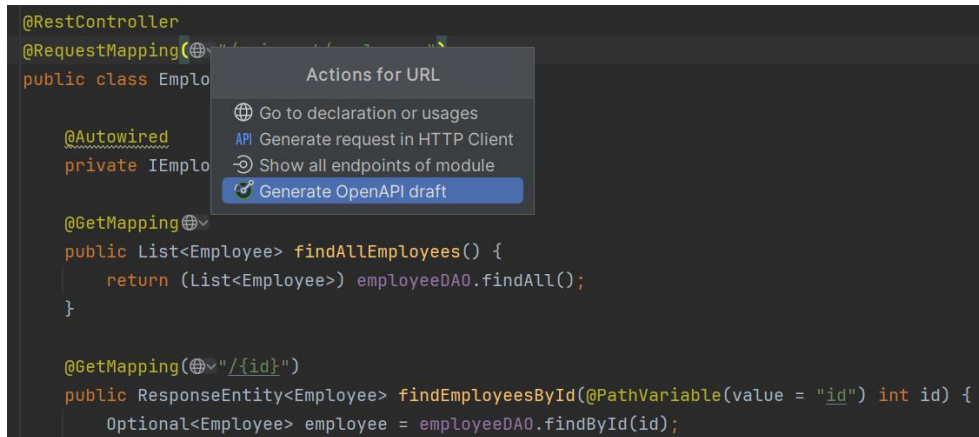
```
get:
  summary: "Get the employee with the corresponding id"
  operationId: "findEmployeesById"
  parameters:
    - name: "id"
      description: "the id of the employee to find"
      in: "path"
      required: true
      schema:
        type: "integer"
        format: "int32"
  responses:
    "200":
      description: "The required employee"
      content:
        '*/*':
          schema:
            $ref: "#/components/schemas/Employee"
    "404":
      description: "The employee has not been found"
  components:
    schemas:
      Employee:
        type: "object"
        properties:
          id:
            type: "integer"
            format: "int32"
          ename:
            type: "string"
          job:
            type: "string"
          deptno:
            type: "integer"
            format: "int32"
```

The most relevant part of this description is the *paths* section, where each of the endpoints available in our API is specified. For these endpoints, the description includes the endpoint itself, the method(s) it supports (GET, POST, etc.), possible responses, and any parameters, if applicable. The *schemas* section (*components/schemas*) is also important, as it provides a detailed specification of the response formats.

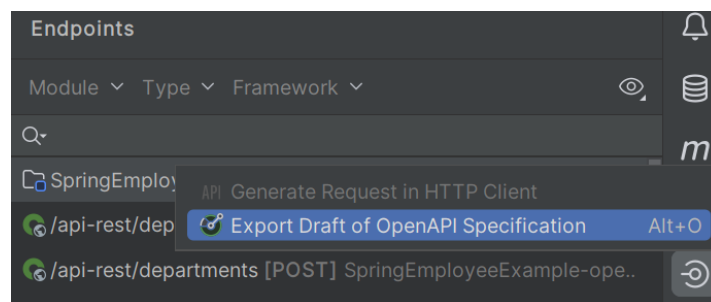
7.2. Generating Documentation with IntelliJ IDEA

Once we have finished developing our API, IntelliJ IDEA can automatically generate its documentation in OpenAPI format. IntelliJ allows us to create

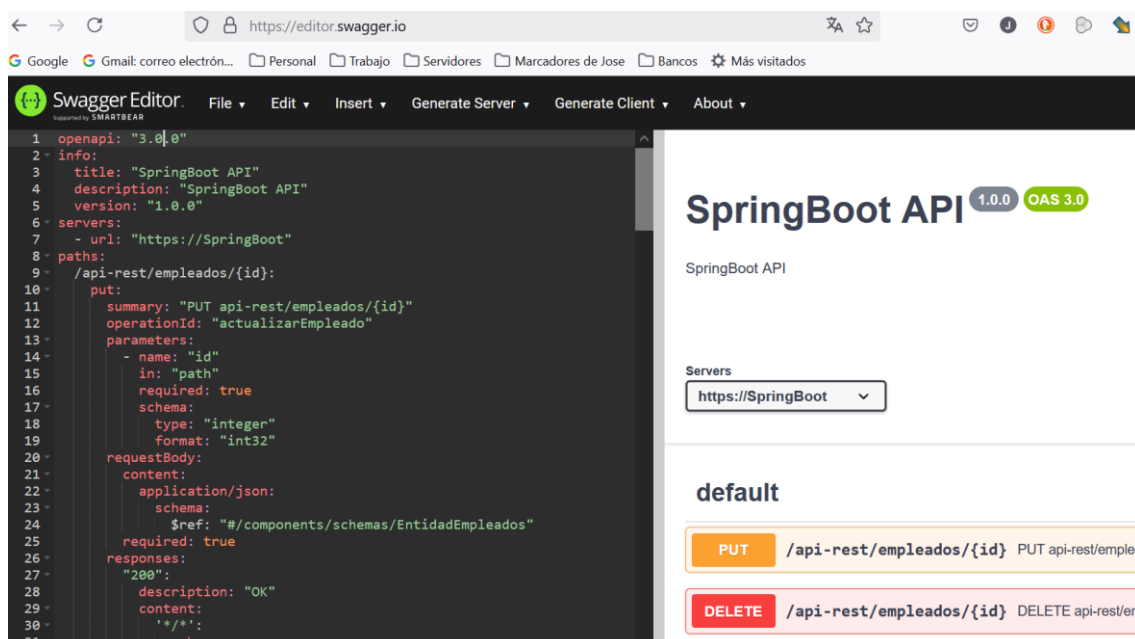
OpenAPI files and generate the corresponding OpenAPI documentation for each controller or individual endpoint.



It's also possible, from the *endpoints* window, to generate the help file for the whole project:



From IntelliJ IDEA, if we select the generated help file, we can preview the API documentation. This same file can be used in services like Swagger Editor to view and modify the documentation.



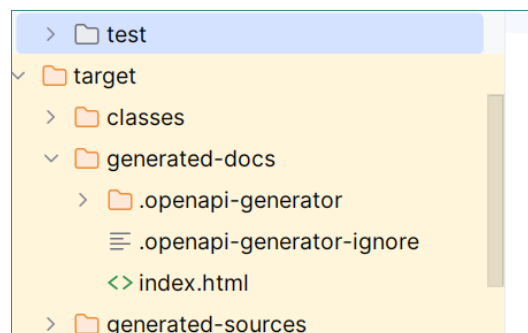
Finally, if we want to generate the documentation in HTML format, we can do so by adding a plugin to our POM file. Once this plugin is included, simply execute the *install* goal to generate the documentation in HTML format.

```
<plugin>
  <groupId>org.openapitools</groupId>
  <artifactId>openapi-generator-maven-plugin</artifactId>
  <version>6.0.0</version>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
  <configuration>

<inputSpec>${project.basedir}/src/main/resources/openapi.yaml</inputSpec>

    <generatorName>html</generatorName>
    <output>${project.build.directory}/generated-docs</output>
  </configuration>
</plugin>
```

After executing the goal, a directory containing the generated documentation should appear in our project.



8. Spring application deployment

The final step, once we have finished testing our application, is to deploy it for use. Spring Boot allows two types of packaging:

- JAR: In this case, the application is packaged along with an embedded Tomcat server, so the only requirement to deploy our application is to have the Java virtual machine installed.

- WAR: Here, the application does not include the Tomcat server. This packaging type is more suitable when we have a Tomcat server installed running one or more web applications.

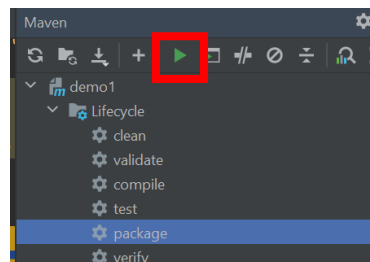
For simplicity, in this document, we will only consider the first scenario.

8.1. Generating the JAR file with IntelliJ IDEA

To perform the packaging of our application, we must first check that we have the Spring Boot plugin for generating JAR files with Maven:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Once verified, generating the JAR file is as simple as going to the Maven lifecycle *package* and executing the corresponding goal:



A new *target* directory will be created in which we will have our JAR file. To simplify, we will copy this file to the same machine where we have our PostgreSQL server running.

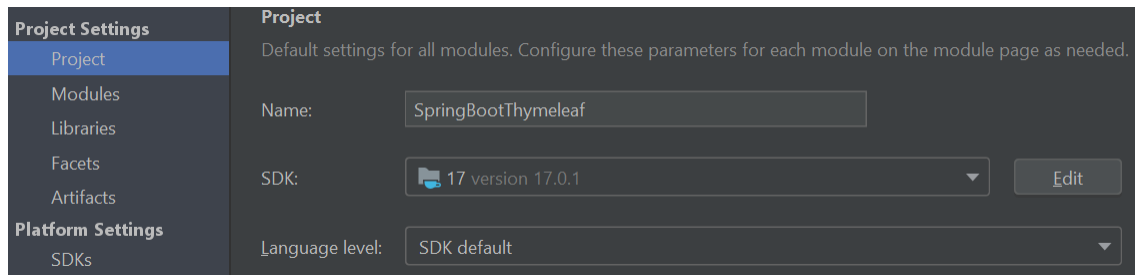
Users > jrgarcia > IdeaProjects > DataAccess > Unit-6 > Employee-Dept-1 > target		
Nombre	Fecha de modificación	Tipo
classes	12/01/2024 20:28	Carpeta
generated-sources	02/02/2022 19:44	Carpeta
generated-test-sources	02/02/2022 19:49	Carpeta
maven-archiver	17/01/2024 13:32	Carpeta
maven-status	17/01/2024 13:31	Carpeta
surefire-reports	17/01/2024 13:32	Carpeta
test-classes	08/01/2024 19:16	Carpeta
demo1-0.0.1-SNAPSHOT.jar	17/01/2024 13:32	Archivo

8.2. Virtual machine deployment

In addition to having the database server on the machine, we must also have the appropriate JRE version installed. To find out the version of JRE we have, we can use the `java --version` command.

```
jose@joseserver:/mnt/hgfs/Shared/certificado$ java --version
openjdk 11.0.21 2023-10-17
OpenJDK Runtime Environment (build 11.0.21+9-post-Ubuntu-0ubuntu120.04)
OpenJDK 64-Bit Server VM (build 11.0.21+9-post-Ubuntu-0ubuntu120.04, mixed mode, sharing)
jose@joseserver:/mnt/hgfs/Shared/certificado$ _
```

The installed version of Java must match the version we used to generate the JAR file with Maven. We can check this in the *Project Structure* section:



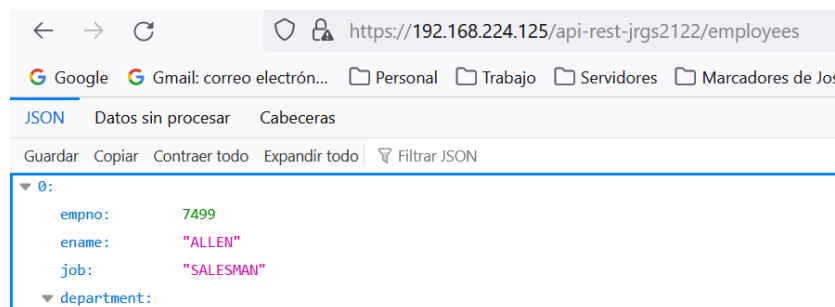
If you are using a SDK newer than the installed JVM, you will need to select the appropriate SDK and build the project again. Language level can differ from the SDK, in case you find compiling issues. Also, we have to take into account Spring Boot minimum requirements:

- Spring Boot versions prior to 3 require Java 8.
- Starting with Spring Boot 3, Java 17 is required.

Once you have generated the correct version of the JAR file, simply execute it on the virtual machine:

```
sudo java -jar miAppSpringBoot.jar
```

The sudo command grants superuser permissions, necessary to run the Spring application. If everything has gone well, the application should now be responsive at the IP address of your server:



8.3. Activity

Deploy in a virtual machine or in a docker container your REST API.

9. Annex: Unit testing class example

```
import com.jrgs.unidad5.employees.Application;
import com.jrgs.unidad5.employees.modelo.dao.IEmployeeDAO;
import com.jrgs.unidad5.employees.modelo.entidades.EmployeeEntity;
import com.jrgs.unidad5.employees.controladores.EmployeeController;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.springframework.http.ResponseEntity;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.*;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = Application.class)
class ApplicationTests {

    @Mock
    private IEmployeeDAO employeeDAO;

    @InjectMocks
    private EmployeeController employeeController;

    @Test
    void findAllEmployees() {
        // Arrange
        List<EmployeeEntity> employees = new ArrayList<>();
        employees.add(new EmployeeEntity(1000, "Nombrel", "Puesto1",
10));
        employees.add(new EmployeeEntity(1001, "Nombre2", "Puesto2",
20));

        when(employeeDAO.findAll()).thenReturn(employees);

        // Act
        List<EmployeeEntity> result =
employeeController.findAllEmployees();

        // Assert
        assertEquals(2, result.size());
    }

    @Test
    void findEmployeesById() {
        // Arrange
        int id = 1000;
        EmployeeEntity employee = new EmployeeEntity(id, "Nombrel",
"Puesto1", 10);

        when(employeeDAO.findById(id)).thenReturn(Optional.of(employee));

        // Act
```

```
        ResponseEntity<EmployeeEntity> result =
employeeController.findById(id);

        // Assert
        assertEquals(200, result.getStatusCodeValue());
        assertEquals(employee, result.getBody());
    }

    @Test
    void findByIdNotExists() {
        // Arrange
        int id = 1099;

        when(employeeDAO.findById(id)).thenReturn(Optional.empty());

        // Act
        ResponseEntity<EmployeeEntity> result =
employeeController.findById(id);

        // Assert
        assertEquals(404, result.getStatusCodeValue());
    }

    @Test
    void saveEmployee() {
        // Arrange
        EmployeeEntity employee = new EmployeeEntity(1000, "Nombre1",
"Puesto1", 10);

        when(employeeDAO.save(any(EmployeeEntity.class))).thenReturn(employee);

        // Act
        EmployeeEntity result =
employeeController.saveEmployee(employee);

        // Assert
        assertEquals(employee, result);
    }

    @Test
    void updateEmployee() {
        // Arrange
        int id = 1000;
        EmployeeEntity nuevoEmpleado = new EmployeeEntity(id,
"NuevoNombre", "NuevoPuesto", 10);
        EmployeeEntity employeeExistente = new EmployeeEntity(id,
"NombreAntiguo", "PuestoAntiguo", 20);

        when(employeeDAO.findById(id)).thenReturn(Optional.of(existingEmployee));

        when(employeeDAO.save(any(EmployeeEntity.class))).thenReturn(existingEmployee);

        // Act
        ResponseEntity<?> result =
employeeController.updateEmployee(newEmployee, id);
```

```
// Assert
assertEquals(200, result.getStatusCodeValue());
assertEquals("Updated", result.getBody());
assertEquals("NuevoNombre", existingEmployee.getNombre()); //
Verificar que el nombre se actualizó correctamente
}

@Test
void updateEmployeeNotExists() {
    // Arrange
    int id = 1099;
    EmployeeEntity newEmployee = new EmployeeEntity(1000,
    "NuevoNombre", "NuevoPuesto", 10);

    when(employeeDAO.findById(id)).thenReturn(Optional.empty());

    // Act
    ResponseEntity<?> result =
    employeeController.updateEmployee(newEmployee, id);

    // Assert
    assertEquals(404, result.getStatusCodeValue());
}

@Test
void deleteExistingEmployee() {
    // Arrange
    int id = 1000;
    EmployeeEntity existingEmployee = new
    EmployeeEntity(id, "NombreAntiguo", "PuestoAntiguo", 10);

    when(employeeDAO.findById(id)).thenReturn(Optional.of(existingEmployee
    ));

    // Act
    ResponseEntity<?> result =
    employeeController.deleteEmployee(id);

    // Assert
    assertEquals(200, result.getStatusCodeValue());
    assertEquals("Borrado", result.getBody());
    verify(employeeDAO, times(1)).deleteById(id); // Verificar que
    se llamó al método deleteById
}

@Test
void deleteNonExistingEmployee() {
    // Arrange
    int id = 1099;

    when(employeeDAO.findById(id)).thenReturn(Optional.empty());

    // Act
    ResponseEntity<?> result =
    employeeController.deleteEmployee(id);

    // Assert
    assertEquals(404, result.getStatusCodeValue());
}
}
```


10. Bibliography

[What is REST?](#)

[What is a REST API?](#)

[Spring Boot reference documentation](#)

[Spring Boot and IntelliJ IDEA](#)

[Relationships in a Spring REST API](#)

[JAVA API for JSON processing](#)

[JAVA HttpURLConnection](#)

[Securing a Spring Boot application](#)

[Deploying a Spring Boot application on AWS](#)

[Mapping DTO's automatically](#)

[Spring Boot unit testing](#)

[Cross Site Request Forgery \(CSRF\)](#)