# What is Swagger?

Swagger is a set of tools that help developers to create, edit, and use API documentation according to the OpenAPI specification. With Swagger, it is no longer necessary to manually write lengthy API docs: the solution is capable of reading the structure of the API you defined in the annotations of your code and automatically converting it into API specification.

What is more, Swagger provides a user interface that generates interactive API documentation that lets users test the API calls in the browser.

Main Swagger components are:

> Swagger Editor for writing and editing API specs,
> Swagger UI for creating interactive API documentation,
> Swagger Codegen for generating server stubs or client libraries for your API.

## Swagger vs OpenAPI

Both terms, Swagger and OpenAPI, are used in the context of API documentation, but they are not the same. OpenAPI is a standard specification for describing API, and Swagger helps to create API docs in line with this specification.

For example, BellSoft uses REST Discovery API based on OpenAPI specification to provide metadata about its products (version, build number, architecture, features, etc.) By querying this information, users can get a comprehensive understanding of product characteristics.

# Integrating Swagger into a Spring Boot project

## Prerequisites

- JDK 17 or later (I will use Liberica JDK recommended by the Spring team)
- Maven
- Docker
- Your favorite IDE (I will use IntelliJ IDEA)

## Create a sample REST API project

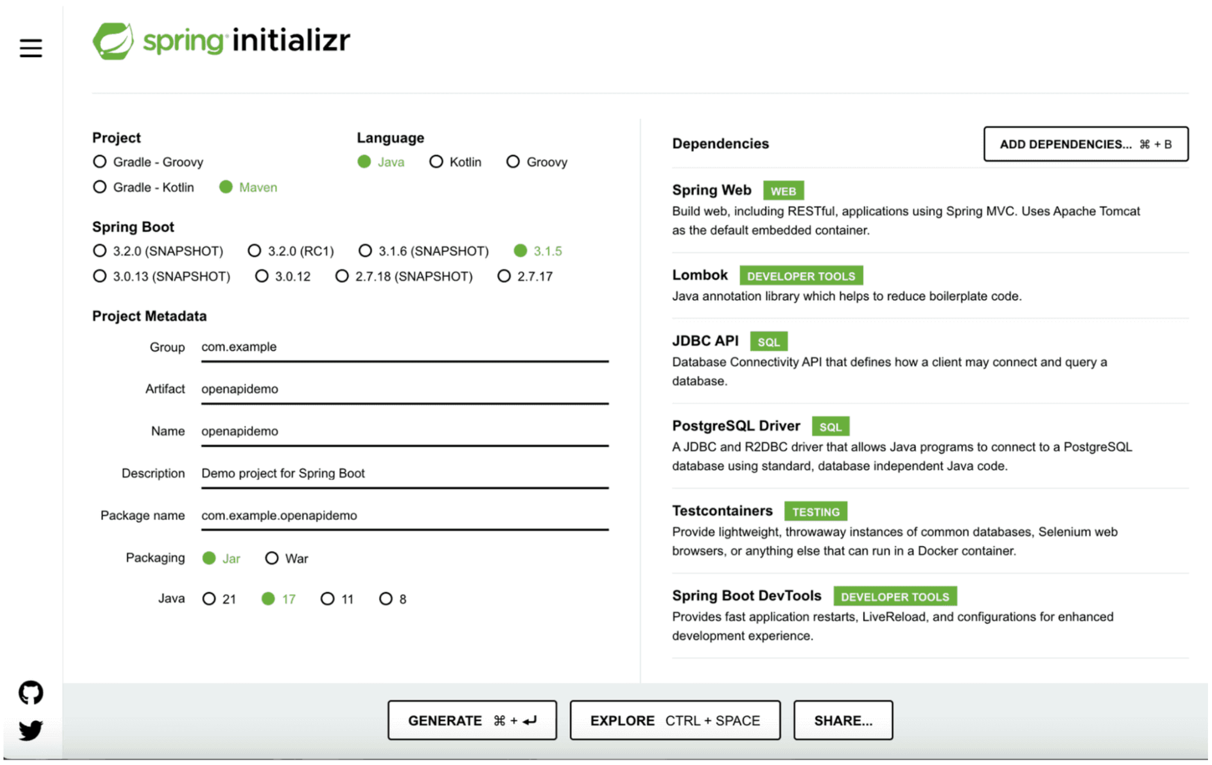
Skip this step if you want to use your own project. You can follow along or implement the examples from the tutorial into your application accordingly.

Our demo Spring Boot application will expose REST APIs for managing employees. The Employee will have id, first name, and last name. Our APIs will allow for getting a list of all employees, getting one employee, adding a new employee, updating the existing employee, and deleting an employee according to the following schema:

| Method | URL | Action |
|---|---|---|
| GET | /employees | Get a list of all employees |
| GET | /employees/{employeeId} | Get one employee by id |
| POST | /employees | Add an employee |
| PUT | /employees | Update an employee |
| DELETE | /employees/{employeeId} | Delete an employee |

Head to Spring Initializr to create a skeleton of your project. Select Java, Maven, define a project name (I have *openapidemo*), and choose Java 17.

To save the effort and avoid winding up a local database, we will use Testcontainers that provide connection to ready dockerized database images. We will also need Spring Web, Lombok, Spring Data JDBC, PostgreSQL Driver, and DevTools dependencies. DevTools is a great time saver because you don’t have to restart the application every time you introduce changes, the recompilation is performed on the fly.



**Project**
Gradle - Groovy
Gradle - Kotlin
[✓ Maven]

**Language**
[✓ Java] Kotlin Groovy

**Spring Boot**
3.2.0 (SNAPSHOT) 3.2.0 (RC1) 3.1.6 (SNAPSHOT) [✓ 3.1.5]
3.0.13 (SNAPSHOT) 3.0.12 2.7.18 (SNAPSHOT) 2.7.17

**Project Metadata**

| | |
|---|---|
| Group | com.example |
| Artifact | openapidemo |
| Name | openapidemo |
| Description | Demo project for Spring Boot |
| Package name | com.example.openapidemo |
| Packaging | [✓ Jar] War |
| Java | 21 [✓ 17] 11 8 |

**Dependencies**

**Spring Web** WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Lombok** DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

**JDBC API** SQL
Database Connectivity API that defines how a client may connect and query a database.

**PostgreSQL Driver** SQL
A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

**Testcontainers** TESTING
Provide lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container.

**Spring Boot DevTools** DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

*Creating the project*

Generate the project and open it in your IDE.

Let's keep the structure super simple. We will need only two classes, `Employee` and `EmployeeController`, and an `EmployeeRepository` interface.

Populate the classes as shown below.

Employee

EmployeeRepository

```
public interface EmployeeRepository extends
ListCrudRepository<Employee, Integer> { }
```

EmployeeController

```
@RestController
public class EmployeeController {
   @Autowired
   private EmployeeRepository repository;

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

   @GetMapping("/employees")
   public List<Employee> findAllEmployees() {
       return repository.findAll();
   }

   @GetMapping("/employees/{employeeId}")
   public Employee getEmployee(@PathVariable int
employeeId) {
       Employee employee =
repository.findById(employeeId)
               .orElseThrow(() -> new
RuntimeException("Employee id not found - " +
employeeId));
       return employee;
   }

   @PostMapping("/employees")
   public Employee addEmployee(@RequestBody Employee
employee) {
       employee.setId(0);
       Employee newEmployee =
repository.save(employee);
       return newEmployee;
   }

   @PutMapping("/employees")
   public Employee updateEmployee(@RequestBody Employee
employee) {
       Employee theEmployee =
repository.save(employee);
       return theEmployee;
   }
}

@DeleteMapping("/employees/{employeeId}")
public String deleteEmployee(@PathVariable int
employeeId) {
   Employee employee = repository.findById(employeeId)
           .orElseThrow(() -> new
RuntimeException("Employee id not found - " +
employeeId));
   repository.delete(employee);
   return "Deleted employee with id: " + employeeId;
}
```

Navigate to the `TestOpenapidemoApplication` class (your name may be different if you gave a different name to your project) in the test folder. Add `@RestartScope` annotation to the `PostgreSQLContainer<?> postgresContainer()` method — this will preserve our DB bean between restarts.

Now, let's provide a database schema. Create a *schema.sql* file in the resources directory with the following content:

```
create table if not exists employee (
id serial primary key,
first_name varchar(255) not null,
last_name varchar(255) not null
);
```

Next, let’s populate our database instance with some data (that’s optional, we don’t need to work with DB data when developing APIs, I just don’t like to see the ugly error page upon starting the app in the browser). Create the data.sql file in resources with the following content:

```
delete from employee;
insert into employee (first_name, last_name) values
('John', 'Doe');
insert into employee (first_name, last_name) values
('Jane', 'Smith');
```

The last thing to do is to add

```
spring.sql.init.mode=always
```

to the *application.properties* file because we are performing a script-based initialization.

Finally, let’s verify that the app is functioning as desired. Run it from the `TestOpenapiApplication` class (but first, make sure that Docker is running on your machine!), and Spring Boot will pull a PostgreSQL database image for you.

You should see the following result at *http://localhost:8080/employees*:

```
[
{
"id": 1,
"firstName": "John",
"lastName": "Doe"
},
{
"id": 2,
"firstName": "Jane",
"lastName": "Smith"
}
]
```

That’s it! Our minimalistic CRUD application is ready for experiments.

## Add springdoc-openapi dependency

To work with Swagger, we need the springdoc-api library that helps to generate OpenAPI-compliant API documentation for Spring Boot projects. The library supports Swagger UI and other useful features such as OAuth2 and GraalVM Native Image.

Add the following dependency for *springdoc-api* to your *pom.xml* file:

```
<dependency>
  <groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifa
ctId>
  <version>2.2.0</version>
</dependency>
```

That’s all, no additional configuration is required!

## Generate API documentation

The OpenAPI documentation is generated when we build our project. So let’s verify that everything is working correctly. Run your application and go to the default page where the API documentation is located: *http://localhost:8080/v3/api-docs*.

You should see the data on your endpoints in JSON format. You can also access the .yaml file at *http://localhost:8080/v3/api-docs.yaml*.

It is possible to change the default path in the application.properties file. For example:

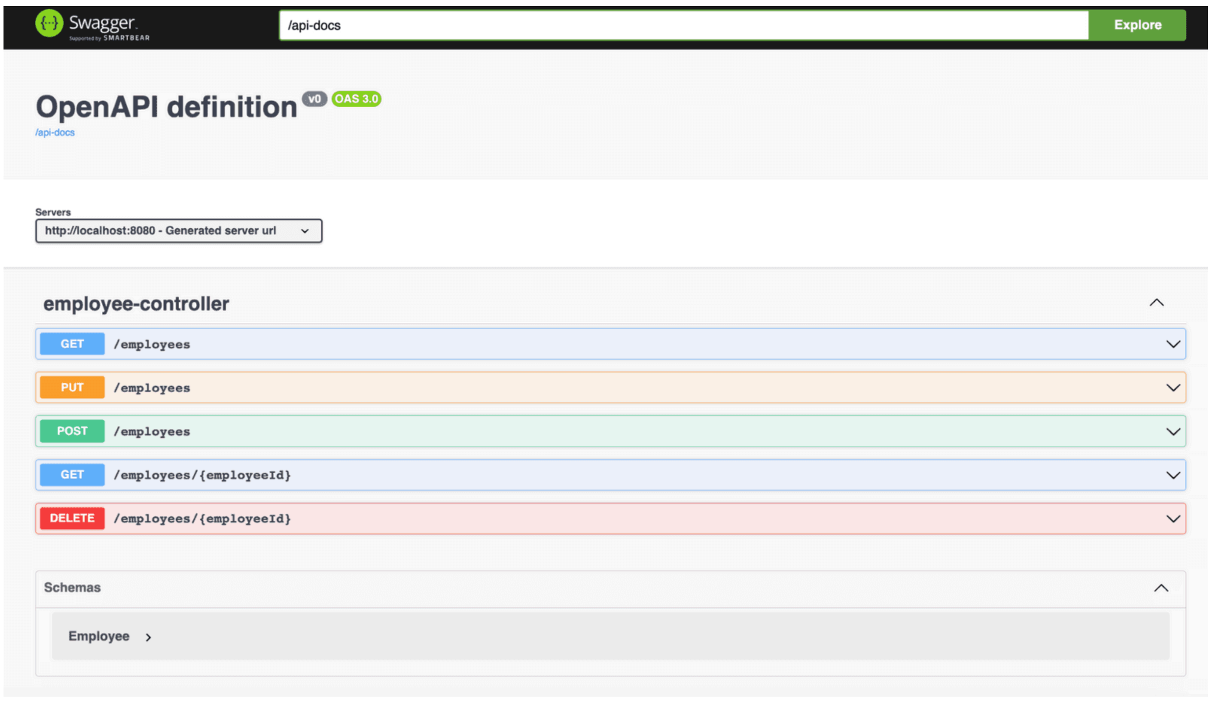
```
springdoc.api-docs.path=/api-docs
```

Now the documentation is available at *http://localhost:8080/api-docs*.

## Integrate Swagger UI

The beauty about springdoc-openapi library dependency is that it already includes Swagger UI, so we don’t have to configure the tool separately!

You can access Swagger UI at *http://localhost:8080/swagger-ui/index.html*, where you will see a beautiful user interface to interact with your endpoints (or similar to the one on the screenshot if you are using your project):



*Swagger UI*

## Configure Swagger 3 in Spring Boot with annotations

Right now, our API documentation is not very informative. We can extend it with the help of annotations added to the application code. Below is the summary of the most common ones.

### Add Swagger API description

First of all, let’s include some essential data about the API, such as name, description, and author contacts. For that purpose, create an OpenAPIConfiguration class and fill in the following code:

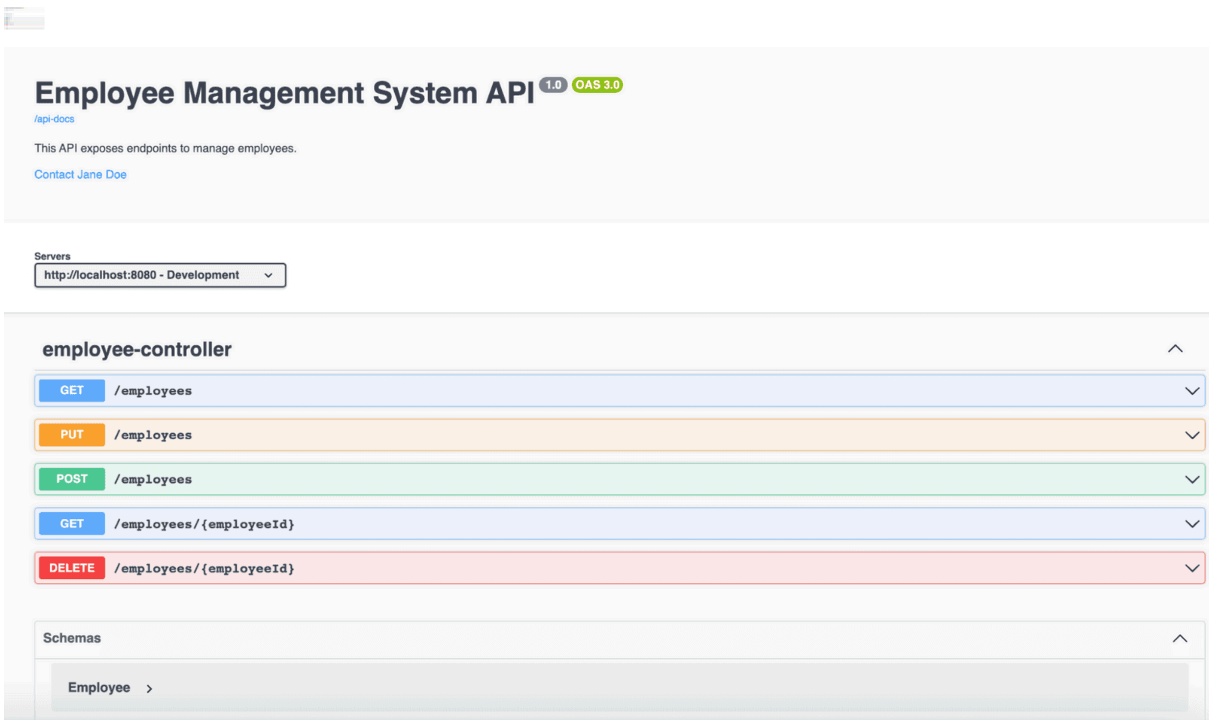
```
@Configuration
public class OpenAPIConfiguration {

   @Bean
   public OpenAPI defineOpenApi() {
       Server server = new Server();
       server.setUrl("http://localhost:8080");
       server.setDescription("Development");

       Contact myContact = new Contact();
       myContact.setName("Jane Doe");
       myContact.setEmail("your.email@gmail.com");

       Info information = new Info()
               .title("Employee Management System API")
               .version("1.0")
               .description("This API exposes endpoints
to manage employees.")
               .contact(myContact);
       return new
OpenAPI().info(information).servers(List.of(server));
   }
}
```

You can also fill in information about applicable License and some other data, but code above is enough for demonstration. Run the app and verify that the main API page includes provided information:


Employee Management System API 1.0 OAS 3.0

/api-docs

This API exposes endpoints to manage employees.

Contact Jane Doe

Servers

http://localhost:8080 - Development

employee-controller

GET /employees

PUT /employees

POST /employees

GET /employees/{employeeId}

DELETE /employees/{employeeId}

Schemas

Employee >

*API description*

## Been validation

The springdoc-openapi library supports JSR 303: Bean Validation (`@NotNull`, `@Min`, `@Max`, and `@Size`), so when we add these annotations to our code, the additional schema documentation will be automatically generated.

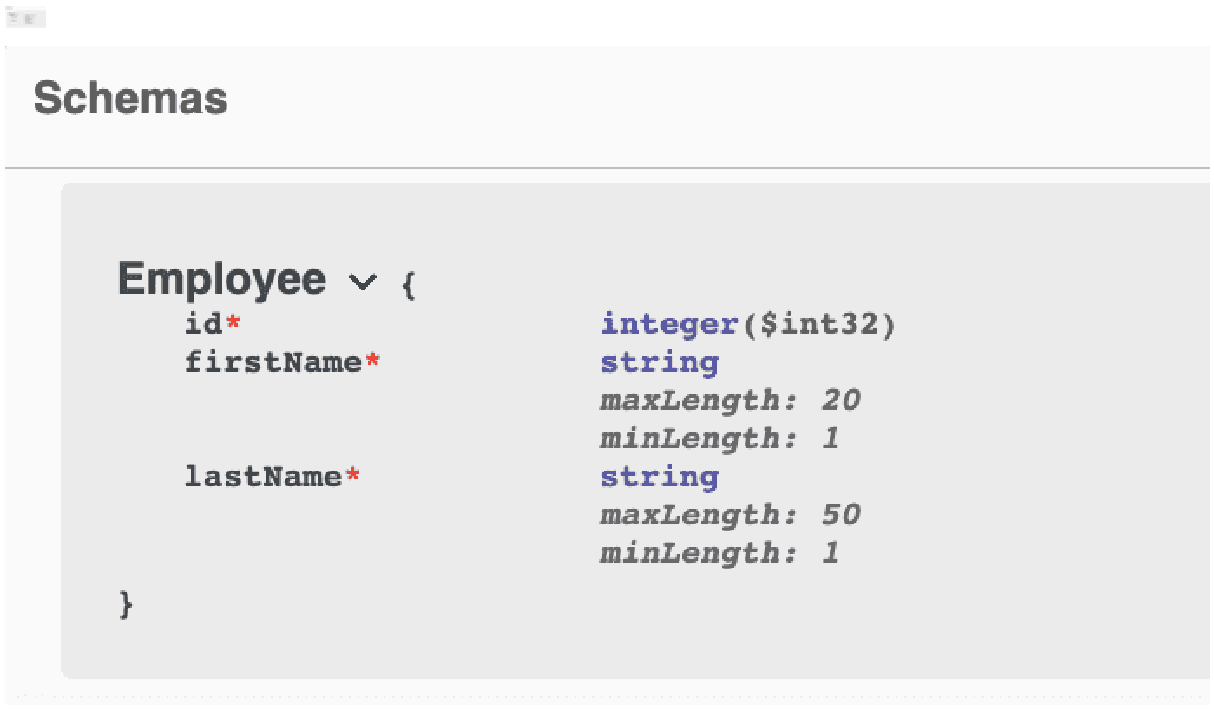
Let’s specify them in `Employee` class:

```
public class Employee {
   @Id
   @NotNull
   private int id;

   @NotNull
   @Size(min = 1, max = 20)
   private String firstName;
```

```
    @NotNull
    @Size(min = 1, max = 50)
    private String lastName;
```

When you recompile your app, you will see that the Schemas section contains the specified info:



```
Schemas

Employee ⌄ {
    id*                integer($int32)
    firstName*         string
                       maxLength: 20
                       minLength: 1
    lastName*          string
                       maxLength: 50
                       minLength: 1
}
```

*API Schema*

## @Tag annotation

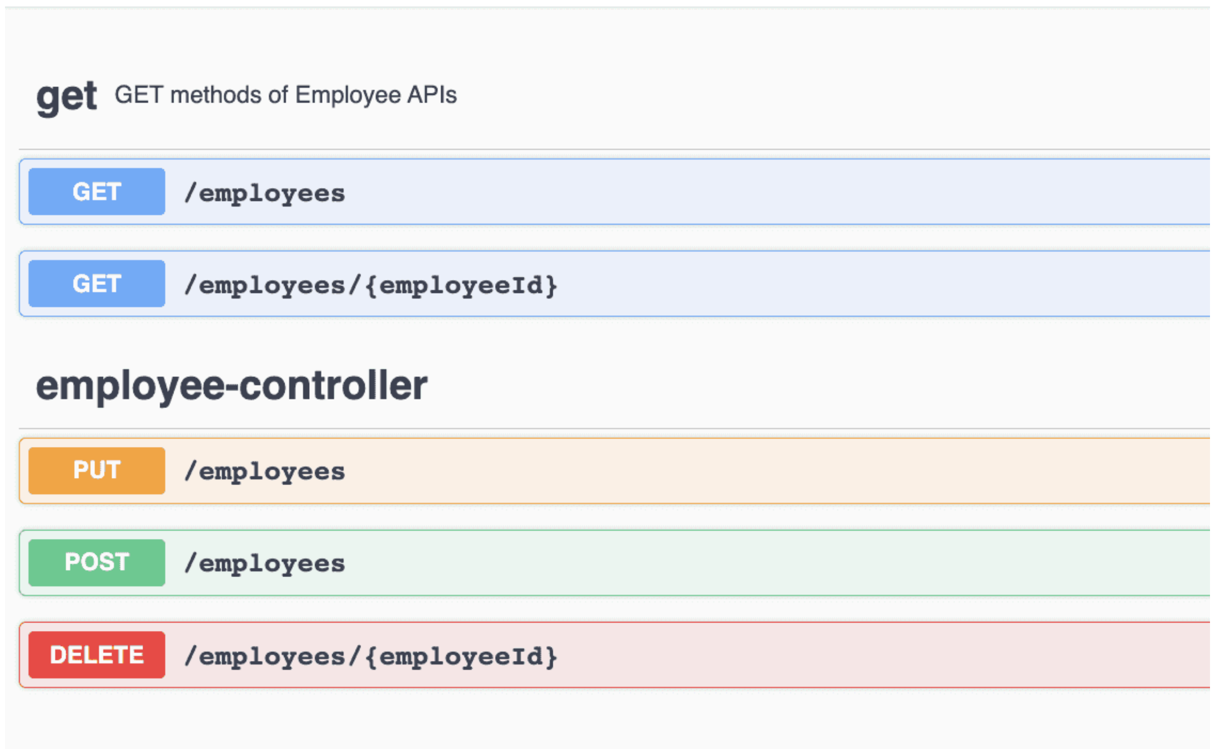
The `@Tag` annotation can be applied at class or method level and is used to group the APIs in a meaningful way.

For instance, let's add this annotation to our GET methods:

```
@Tag(name = "get", description = "GET methods of
Employee APIs")
@GetMapping("/employees")
public List<Employee> findAllEmployees() {
    return repository.findAll();
}

@Tag(name = "get", description = "GET methods of
Employee APIs")
@GetMapping("/employees/{employeeId}")
public Employee getEmployee(@PathVariable int
employeeId) {
    Employee employee = repository.findById(employeeId)
            .orElseThrow(() -> new
RuntimeException("Employee id not found - " +
employeeId));
    return employee;
}
```

You will see that APIs are now grouped differently:



get GET methods of Employee APIs

GET /employees

GET /employees/{employeeId}

employee-controller

PUT /employees

POST /employees

DELETE /employees/{employeeId}

*API grouping*

## @Operation annotation

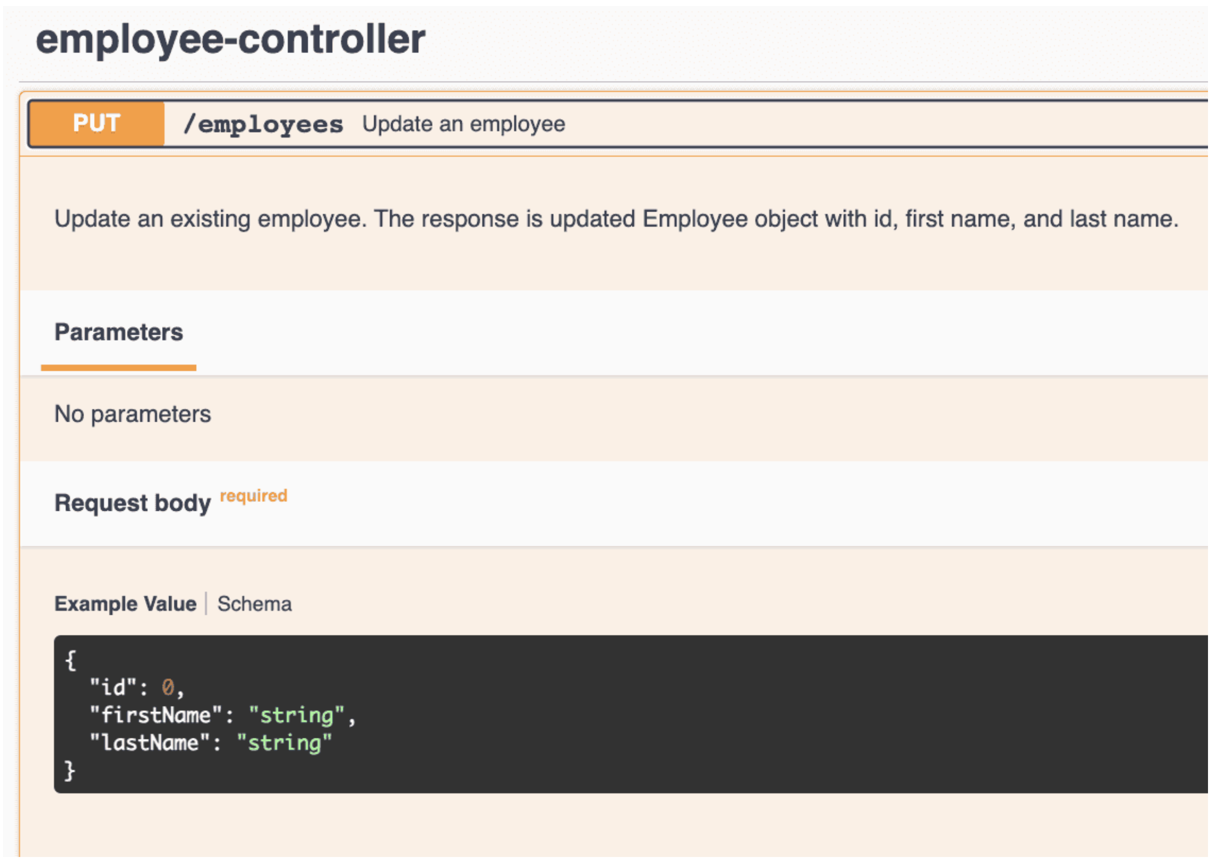
The `@Operation` annotation enables the developers to provide additional information about a method, such as summary and description.

Let’s update our `updateEmployee()` method:

```
@Operation(summary = "Update an employee",
       description = "Update an existing employee. The
response is updated Employee object with id, first
name, and last name.")
@PutMapping("/employees")
public Employee updateEmployee(@RequestBody Employee
employee) {
   Employee theEmployee = repository.save(employee);
   return theEmployee;

}
```

The API description in Swagger UI is now a little more informative:


employee-controller

PUT /employees Update an employee

Update an existing employee. The response is updated Employee object with id, first name, and last name.

Parameters

No parameters

Request body required

Example Value | Schema

```
{
  "id": 0,
  "firstName": "string",
  "lastName": "string"
}
```

*Endpoint description*

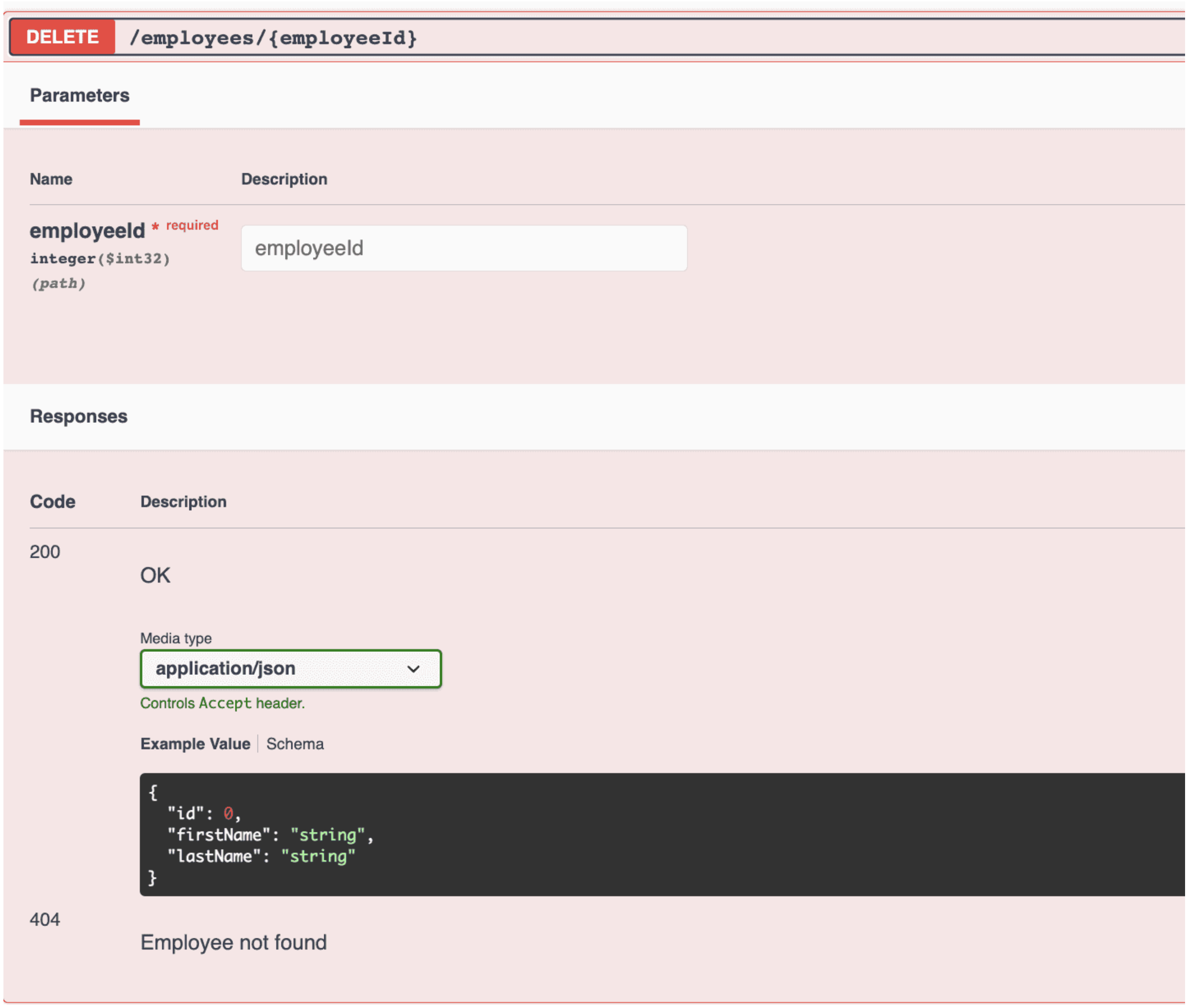
## @ApiResponses annotation

The `@ApiResponses` annotation helps to add information about responses available for the given method. Each response is specified with `@ApiResponse`, for instance

```
@ApiResponses({
       @ApiResponse(responseCode = "200", content = {
@Content(mediaType = "application/json",
               schema = @Schema(implementation =
Employee.class)) }),
       @ApiResponse(responseCode = "404", description =
"Employee not found",
               content = @Content) })
```

```
@DeleteMapping("/employees/{employeeId}")
public String deleteEmployee(@PathVariable int
employeeId) {
   Employee employee = repository.findById(employeeId)
           .orElseThrow(() -> new
RuntimeException("Employee id not found - " +
employeeId));
   repository.delete(employee);
   return "Deleted employee with id: " + employeeId;
}
```

After you recompile the Controller class, the data on responses will be automatically generated:



DELETE /employees/{employeeId}

Parameters

| Name | Description |
|---|---|
| employeeId * required integer($int32) (path) | employeeId |

Responses

| Code | Description |
|---|---|
| 200 | OK |
| 404 | Employee not found |

Media type: application/json

Controls Accept header.

Example Value | Schema

```
{
  "id": 0,
  "firstName": "string",
  "lastName": "string"
}
```

*API responses*

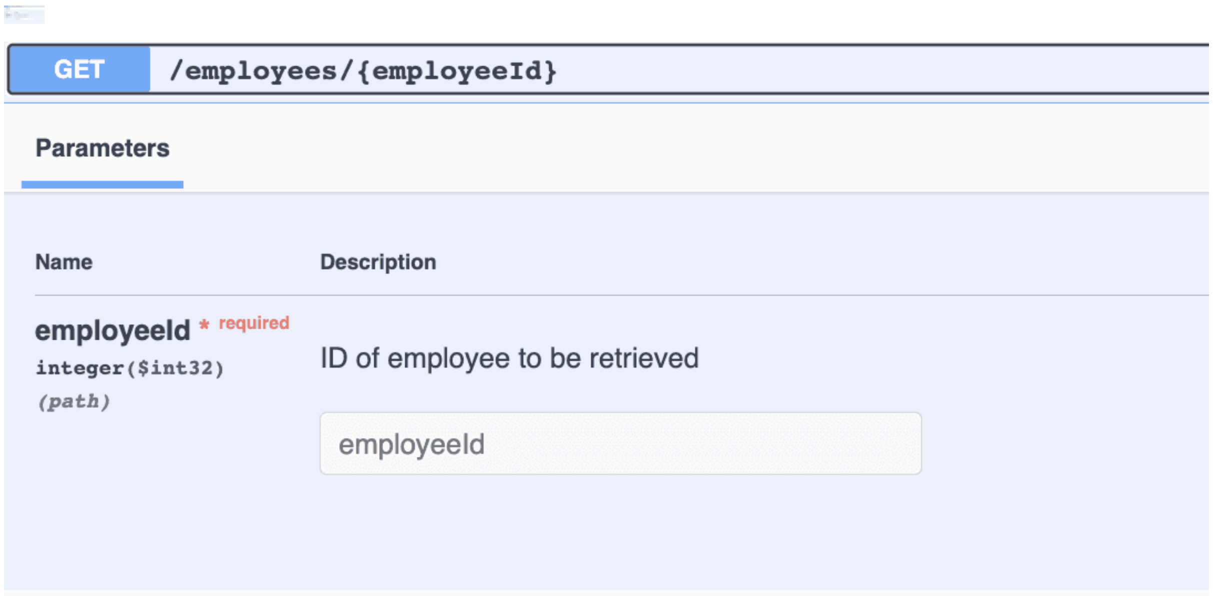
## @Parameter annotation

The `@Parameter` annotation can be used on a method parameter to define parameters for the operation. For example,

```
public Employee getEmployee(@Parameter(
       description = "ID of employee to be retrieved",
       required = true)
       @PathVariable int employeeId) {
   Employee employee = repository.findById(employeeId)
           .orElseThrow(() -> new
RuntimeException("Employee id not found - " +
employeeId));
   return employee;
}
```

Here, the description element provides additional data on parameter purpose, and required is set to true signifying that this parameter is mandatory.

And here's how it looks in Swagger UI:



ID of employee to be retrieved

*Endpoint parameters*