Sample Rest Api Project With Swagger

We will need only two classes, Employee and EmployeeController, and
an EmployeeRepository interface.

Employee

```
@Getter

@Setter

@NoArgsConstructor

public class Employee {

  @Id

  private int id;

  private String firstName;

  private String lastName;


  public Employee(String firstName, String lastName) {

     this.firstName = firstName;

     this.lastName = lastName;

  }

}
```

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;
  }
}
```

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.

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

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <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:

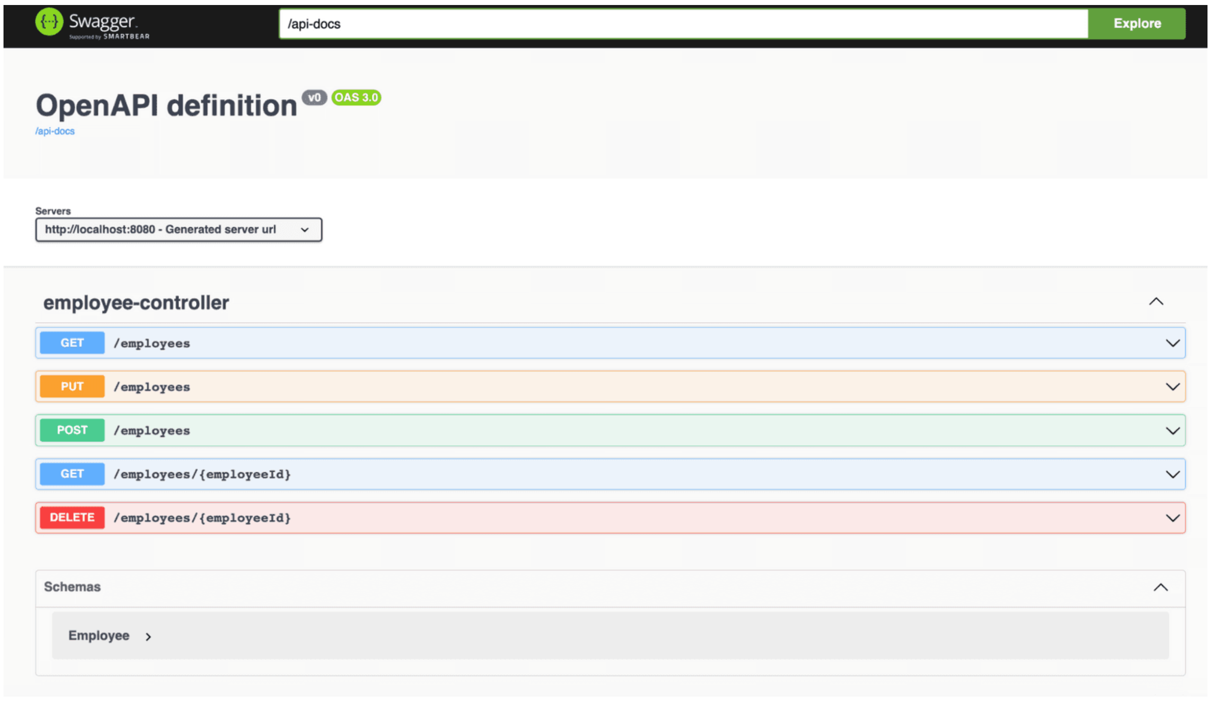
```
springdoc.api-docs.path=/api-docs
```

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 .



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.

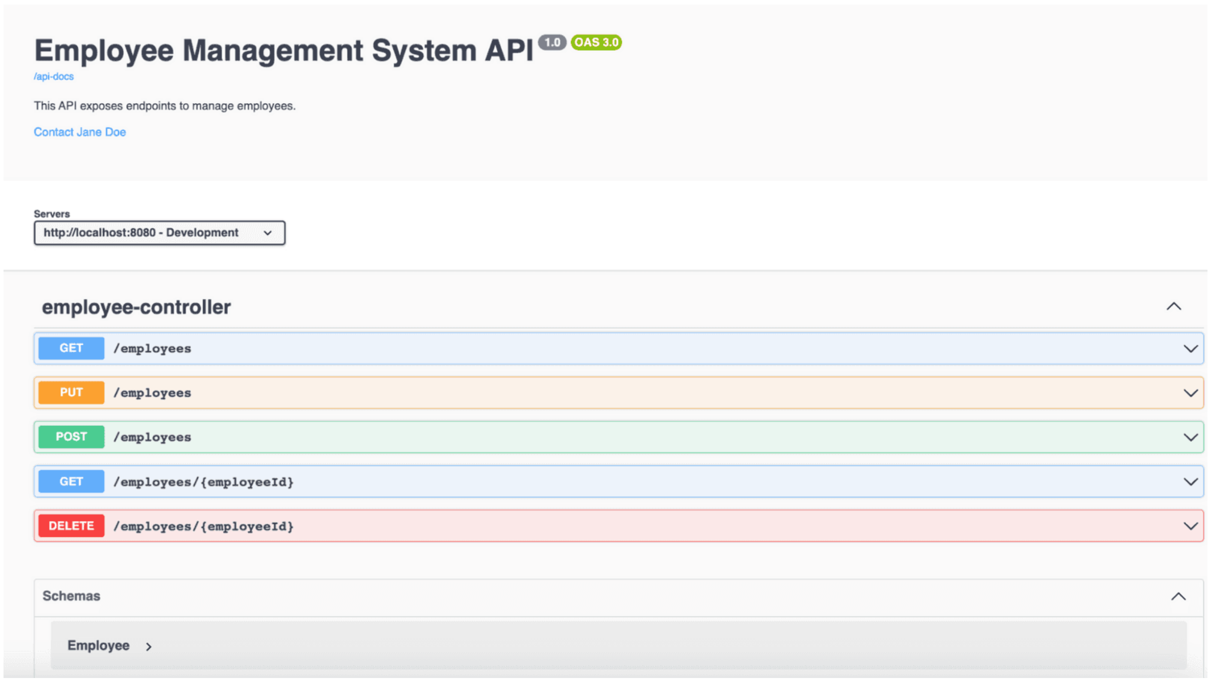
Add Swagger API description

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));
    }
}
```

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

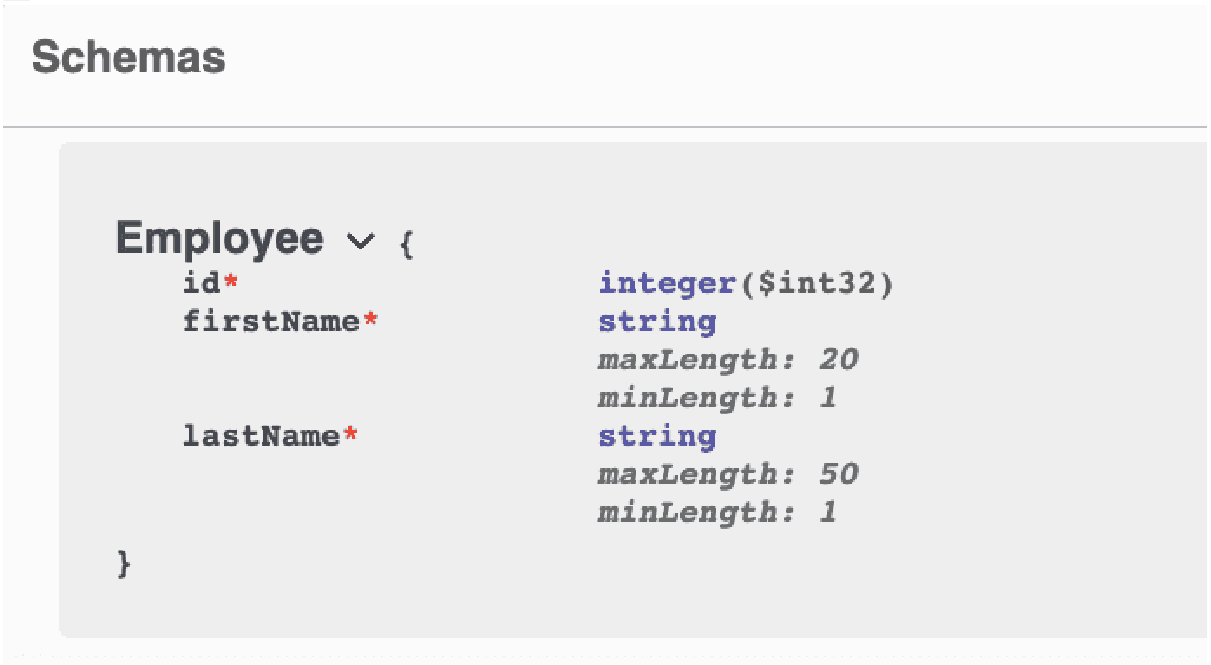
Been validation

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

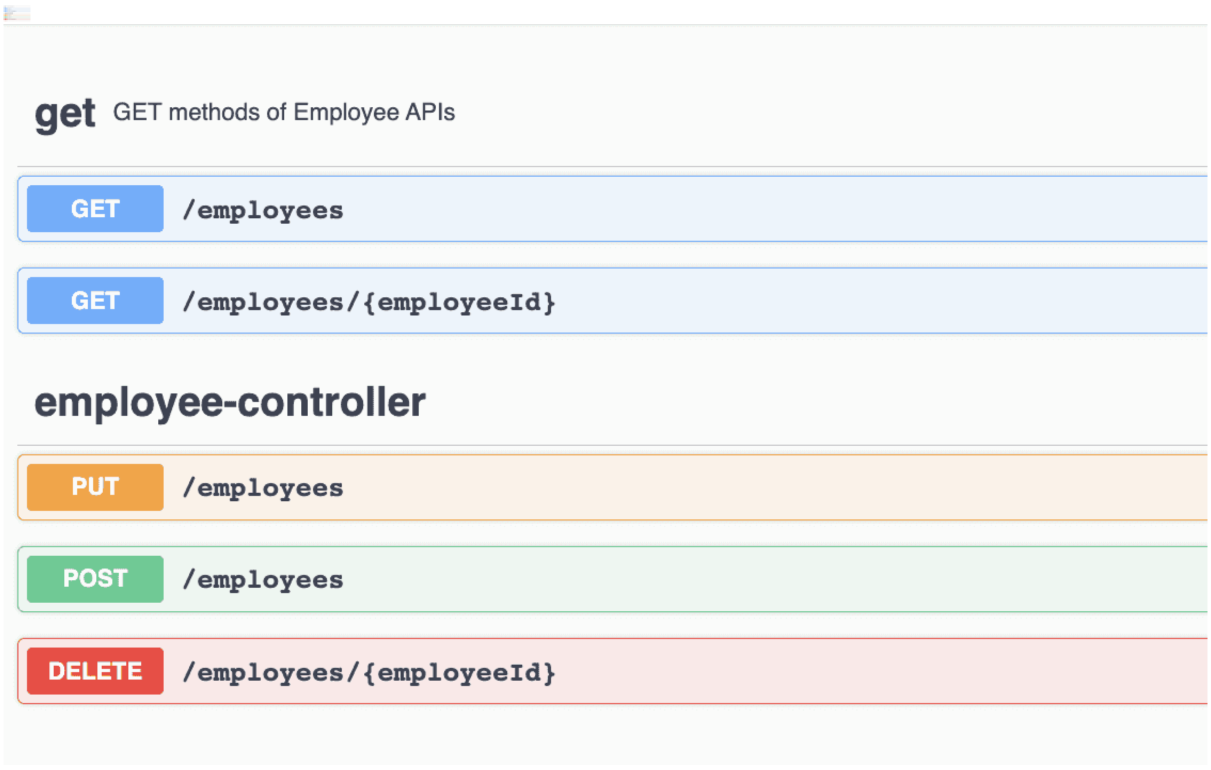
@Tag annotation

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

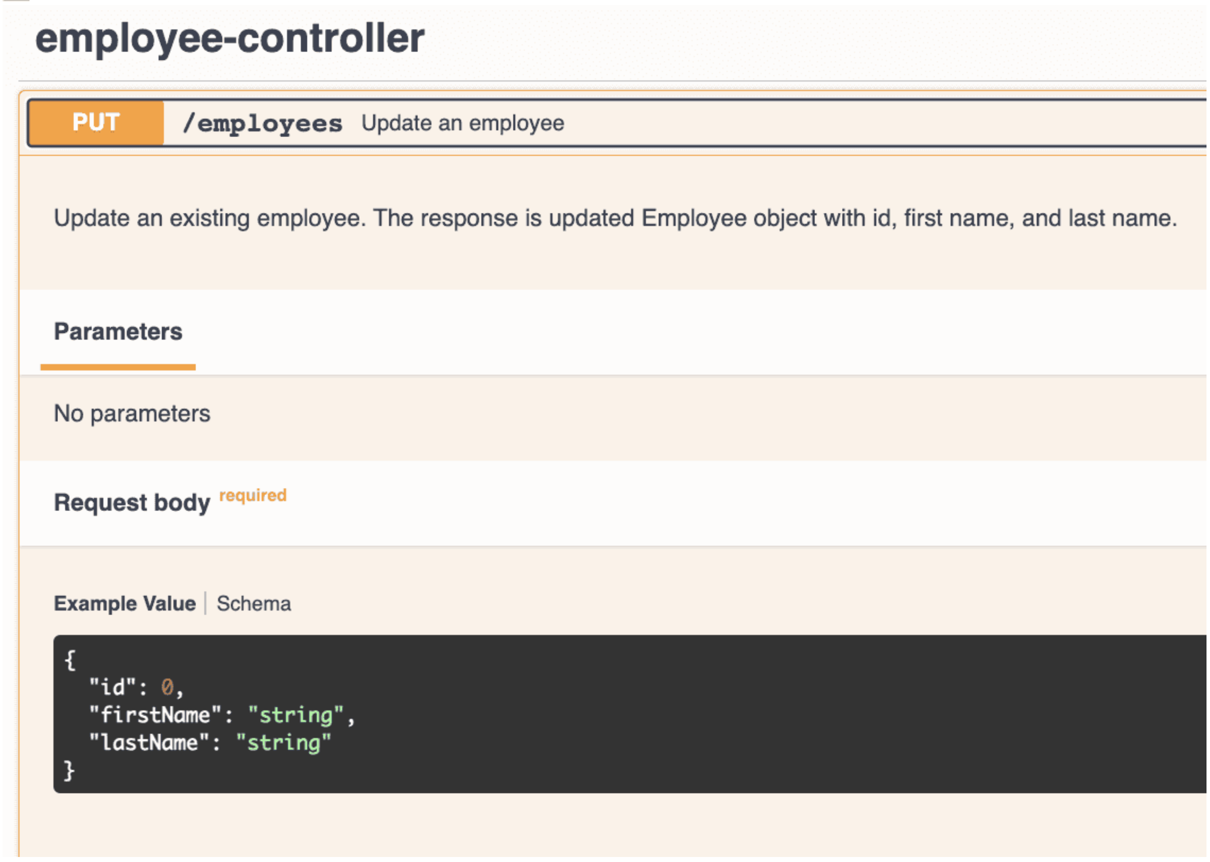
@Operation annotation

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"
}
```

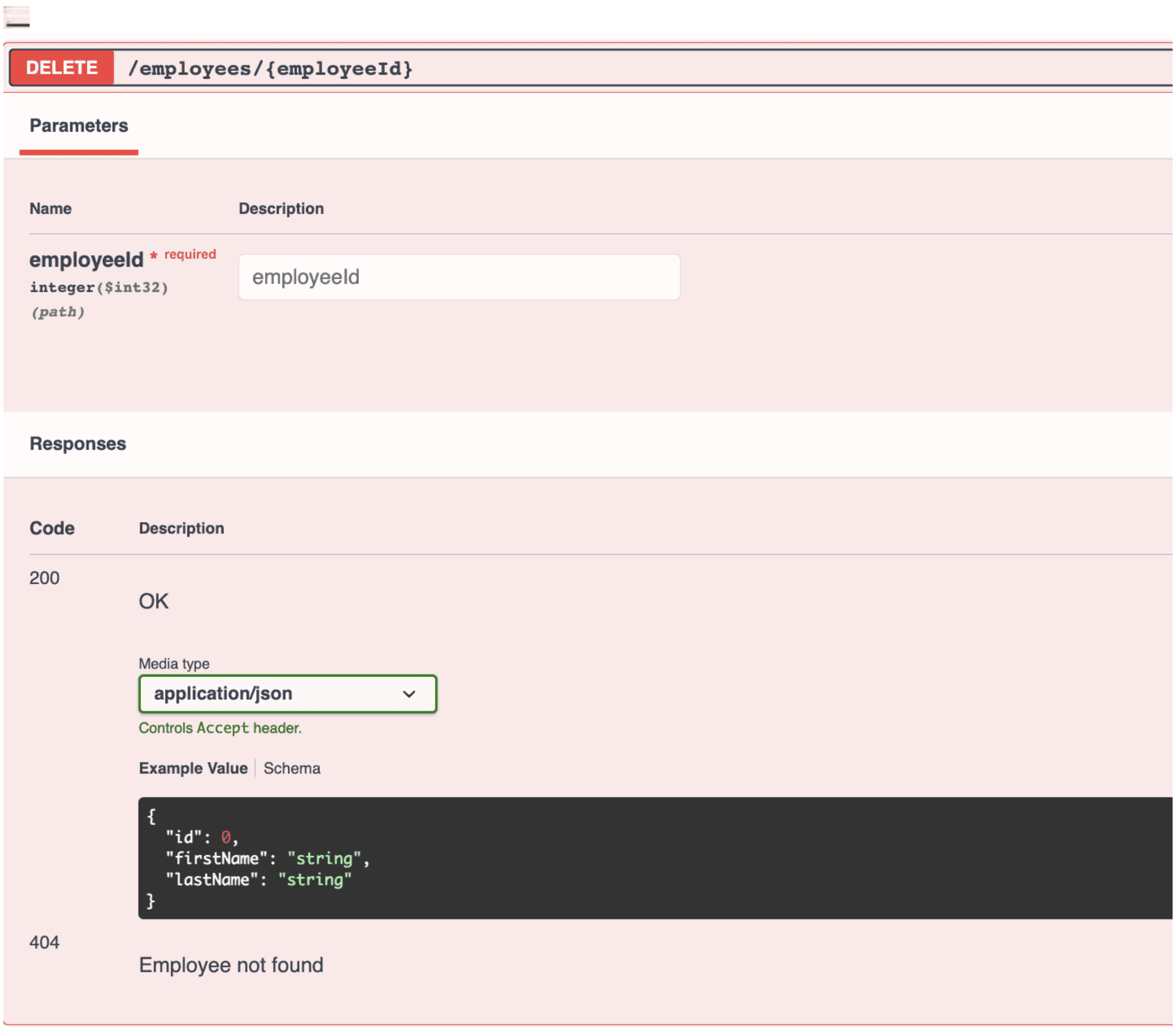
Endpoint description

@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 Media type: application/json Controls Accept header. Example Value | Schema
404	Employee not found

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

API responses

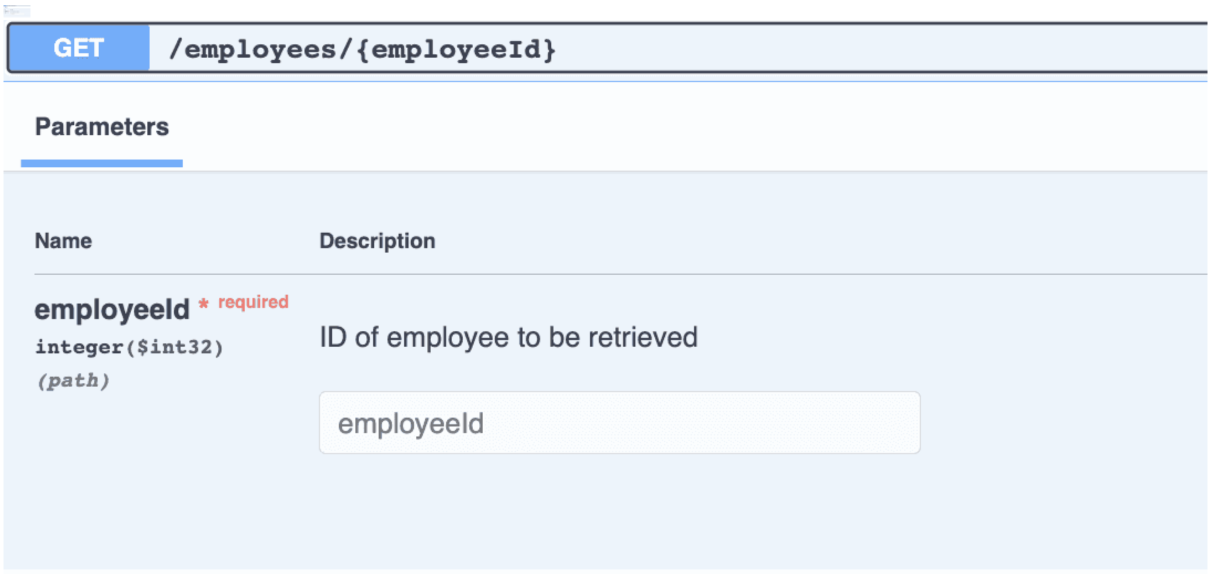
@Parameter annotation

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:



/employees/{employeeId}

Parameters

Name	Description
employeeId * required integer($int32) (path)	ID of employee to be retrieved employeeId

Endpoint parameters