**Introduction to Spring Boot Framework**

Use Cases Manual

Sandbox Link [Spring Boot](https://share.percipio.com/cd/J676mja_Z)

Creating a CRUD REST APIService with Spring Boot, JPA and Hibernate

This guide will help you create a CRUD REST API/Service with Spring Boot, JPA and Hibernate. We will create a JPA Repository for a Student Entity and expose it using a Student Resource.

**Tools you will need**

* Maven 3.0+ is your build tool
* Your favorite IDE. We use Eclipse.
* JDK 1.8+

**You will learn**

* What is a RESTful Service?
* Basics of designing a REST API.
* How to create a RESTful Service offering all CRUD operations?
* How to use Spring Boot, Spring MVC, JPA and Hibernate to create a RESTful API?
* How to execute different kinds of REST API with Postman?
* What are the differences between GET, POST, PUT and DELETE request methods?

**Resources Overview**

In this guide, we will create a Student Resource exposing three services using proper URIs and HTTP methods:

Retrieve all Students - @GetMapping(“/students”)

Get details of specific student - @GetMapping(“/students/{id}”)

Delete a student - @DeleteMapping(“/students/{id}”)

Create a new student - @PostMapping(“/students”)

Update student details - @PutMapping(“/students/{id}”)

**A few details:**

**SpringBoot2RestServiceApplication.java** - The Spring Boot Application class generated with Spring Initializer. This class acts as the launching point for application.

**pom.xml** - Contains all the dependencies needed to build this project. We will use Spring Boot Starter **AOP.**

**Student.java** - Student JPA Entity

**StudentRepository.java** - Student JPA Repository. This is created using Spring Data JpaRepository.

**StudentResource.java** - Spring Rest Controller exposing all services on the student resource.

**data.sql** - Initial data for the student table. Spring Boot would execute this script after the tables are created from the entities.

**When do you use JPA?**

SQL Database

Static Domain Model

Mostly CRUD

Mostly Simple Queries/Mappings

**Do not forget to add JPA and H2 as dependencies.**

**Create Your First JPA Entity**

The first step is to create a JPA Entity. Lets create a simple Student Entity with a primary key id.

@Entity

public class Student {

@Id

@GeneratedValue

private Long id;

private String name;

private String passportNumber;

Important things to note:

**@Entity:** Specifies that the class is an entity. This annotation is applied to the entity class.

**@Id:** Specifies the primary key of an entity.

**@GeneratedValue:** Provides for the specification of generation strategies for the values of primary keys.

**public Student():** Default constructor to make JPA Happy

When the application reloads, you can launch H2 console at <http://localhost:8080/h2-console>

You will see that a new table called ‘student’ is created in H2 Console.

**How did the Student table get created?**

Spring Boot Auto Configuration detects that we are using an in-memory database H2. It autoconfigures the property to create the database tables based on the Entities.

Let’s now populate some data into the student table.

**/src/main/resources/data.sql**

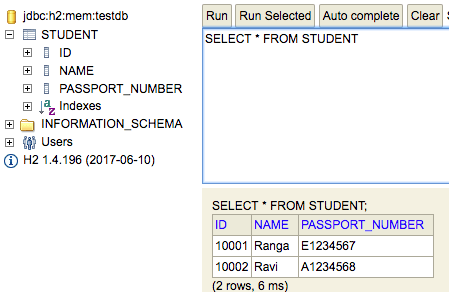
insert into student

values(10001,'Ranga', 'E1234567');

insert into student

values(10002,'Ravi', 'A1234568');

After App Reload, When you login to H2 Console (http://localhost:8080/h2-console) you can see that the student table is created and the data is populated.



**Exposing GET methods on Student Resource**

URL - http://localhost:8080/students

Request Method - GET

URL - http://localhost:8080/students/10002

Request Method – GET

**Expose DELETE Method on Student Resource**

URL - http://localhost:8080/students/10002

Request Method – DELETE

**Expose POST Method to create a new student**

URL - http://localhost:8080/students

Request Method – POST

**Expose PUT Method to update existing student**

Before updating the student we check if the student exists. If the student does not exist, we return a not found status. Otherwise, we save the student details using studentRepository.save method.

URL → http://localhost:8080/students/10002

Request

Method → PUT

**Exposing GET methods on Student Resource**

Let’s create the method to expose the details of all students.

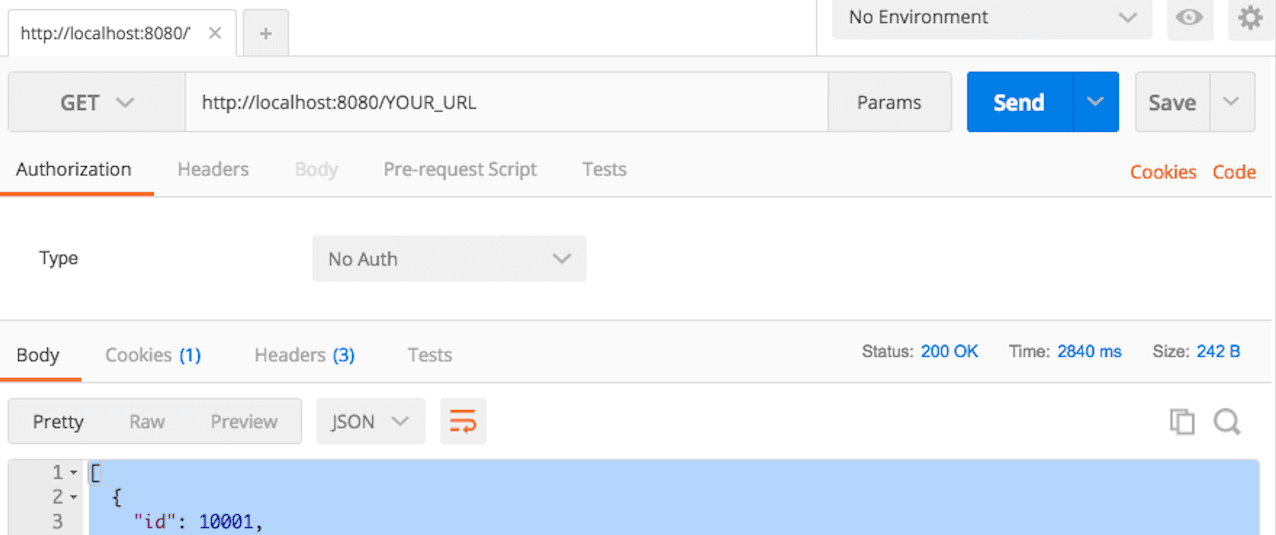
@GetMapping("/students")

public List<Student> retrieveAllStudents() {

return studentRepository.findAll();

}

Below picture shows how we can execute a Get Request Method on a Resource using Postman - my favorite tool to run rest services.



* URL - http://localhost:8080/students
* Request Method - GET

Response

[

{

"id": 10001,

"name": "Ranga",

"passportNumber": "E1234567"

},

{

"id": 10002,

"name": "Ravi",

"passportNumber": "A1234568"

}

]

Let’s create a method to expose the details of a specific student.

@GetMapping("/students/{id}")

public Student retrieveStudent(@PathVariable long id) {

Optional<Student> student = studentRepository.findById(id);

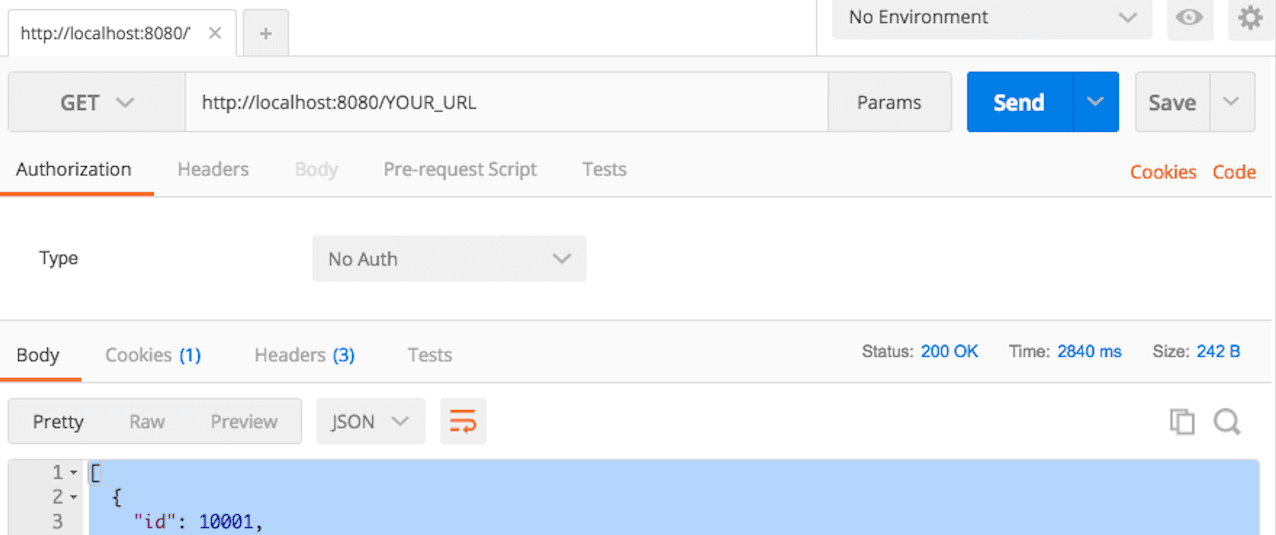
if (!student.isPresent())

throw new StudentNotFoundException("id-" + id);

return student.get();

}

Let’s execute another GET request



* URL - http://localhost:8080/students/10002
* Request Method - GET

Response

{

"id": 10002,

"name": "Ravi",

"passportNumber": "A1234568"

}

**Expose DELETE Method on Student Resouce**

Delete method is simple. All you have to do is to call studentRepository.deleteById(id).

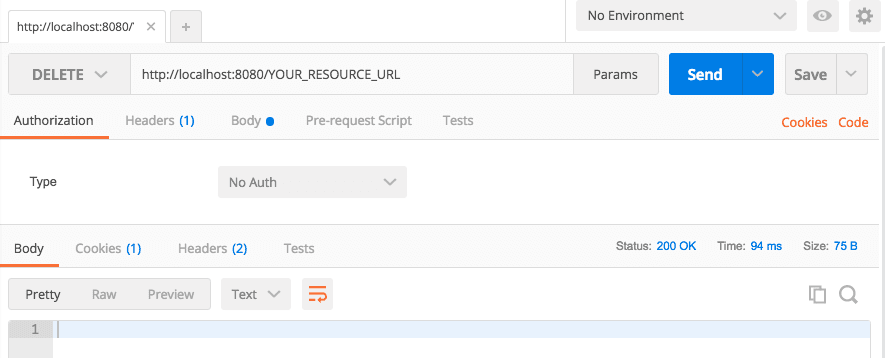
@DeleteMapping("/students/{id}")

public void deleteStudent(@PathVariable long id) {

studentRepository.deleteById(id);

}

Below picture shows how we can execute a DELETE Request method on a Resource from Postman - my favorite tool to run rest services.



* URL - http://localhost:8080/students/10002
* Request Method - DELETE

Request - Empty Body Response with status 200 - Successful

**Expose POST Method to create a new student**

@PostMapping("/students")

public ResponseEntity<Object> createStudent(@RequestBody Student student) {

Student savedStudent = studentRepository.save(student);

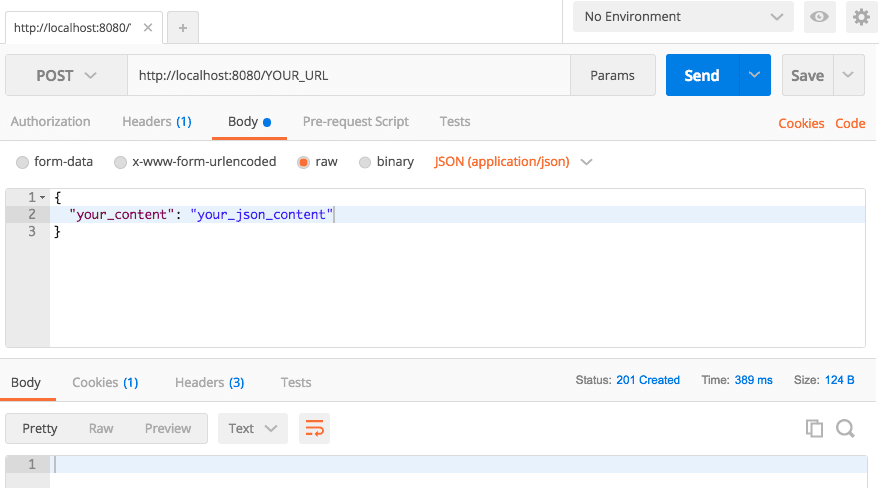
URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")

.buildAndExpand(savedStudent.getId()).toUri();

return ResponseEntity.created(location).build();

}

Below picture shows how we can execute a POST Request method on a Resource from Postman - my favorite tool to run rest services.



* URL - http://localhost:8080/students
* Request Method - POST

Request

{

"name": "Tom",

"passportNumber": "Z1234567"

}

Response

* Status 201 - CREATED
* Header Location →http://localhost:8080/students/2

**Expose PUT Method to update existing student**

Before updating the student we check if the student exists. If the student does not exist, we return a not found status.

@PutMapping("/students/{id}")

public ResponseEntity<Object> updateStudent(@RequestBody Student student, @PathVariable long id) {

Optional<Student> studentOptional = studentRepository.findById(id);

if (!studentOptional.isPresent())

return ResponseEntity.notFound().build();

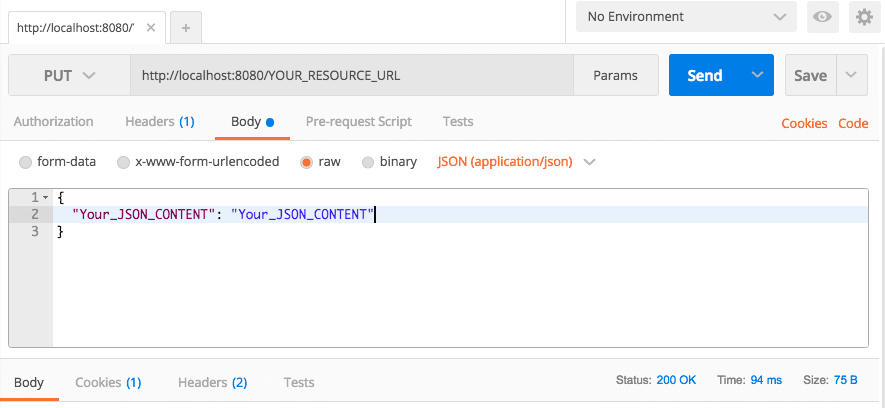
student.setId(id);

studentRepository.save(student);

return ResponseEntity.noContent().build();

}

Below picture shows how we can execute a PUT Request method on a Resource from Postman - my favorite tool to run rest services.



* URL → http://localhost:8080/students/10002
* Request
  + Method → PUT

Request

{

"name": "Tom",

"passportNumber": "Z1234567"

}

Response with status 200 - Successful

Note :

*Remember that you can check the updates in the database using H2 Console http://localhost:8080/h2-console after each of the requests.*