

## Java + Spring Boot: A Simple RESTful API Guide

### Introduction

Before proceeding with the activity, make sure you are comfortable with Git and GitHub, since we will be managing our codebase with version control.

👉 If you are new to Git/GitHub, a complete guide has been prepared for you in this repository:

 [GitHub Guide](#)

This guide includes:

- 📁 How to clone a repository
- 📄 Basic Git commands (commit, push, pull)
- 🔀 Branching and merging basics
- 🚀 Other helpful Git features

⚡ Important: Before starting this activity, ensure that you have successfully cloned the repository where your project will be stored.

If you already know how to navigate and use GitHub, you may proceed directly with the activity.









### Part 1: Prerequisites


Before we dive into creating our RESTful API, make sure you have the following installed:

- ☕ **JDK 17** – Java Development Kit (ensures compatibility with the Spring Boot project).
- 💻 **IntelliJ IDEA** or **VS Code** – Your coding environment (IDE).
- 🌱 **Git** – For version control and project management.
- 🛠️ **Postman** – To test and send requests to our API endpoints.

## Part 2: Creating a Spring Boot Project (Spring Initializer)

We'll use Spring Initializer to quickly generate a starter project.


1. **Open Spring Initializer** (<https://start.spring.io/>)
2. Fill out the project metadata:
  -  **Group** → Usually your organization or package name (e.g., com.accenture). *Purpose:* Groups related projects under a unique identifier.
  -  **Artifact ID** → The name of your project/application (e.g., student-api). *Purpose:* Becomes your project folder name and base identifier for the app.
  -  **Project Type** → Select **Maven**. *Why Maven?* It's a dependency/build management tool widely used with Spring.
  -  **Language** → Java (our main programming language).
  -  **Spring Boot Version** → Stick to the default stable release.
  -  **Packaging** → Choose **Jar**. *Jar:* Runs as a standalone application. *War:* Typically used for deploying to external servers (not needed here).
  -  **Java Version** → Select **17** (must match your installed JDK).
3. **Dependencies** → Add:
  -  **Spring Web** – Provides tools for building REST APIs.

 *For future dependencies:* You can always visit Maven Repository to search and add more libraries as your project grows.

4. Click **Generate**, unzip the file, then open it in **VS Code** (or IntelliJ).

## Part 3: Exploring the Project Structure

Inside your project, locate the file:

 src/main/resources/application.properties

- **Purpose:** This is your application's configuration file.
- You can set different properties like:
  - Server port

- Database connection
- Logging levels

**Example:** Change the default port (8080) to another port (e.g., 9090):

```
server.port=9090
```

## Part 4: Setting Up Layered Architecture

We will follow a **layered architecture** (MVC-inspired), but since we're using an in-memory **HashMap** instead of a database, we'll skip the Repository layer for now.

Your package structure should look like this:

```
src/main/java/com/accenture/studentapi
├── model      # 📦 Holds your data classes (e.g., Student.java)
├── service    # ⚙️ Contains business logic (e.g., StudentService.java)
└── controller # 🌐 Handles API endpoints (e.g., StudentController.java)
```

### 💡 Explanation of Layers

- **Model (Entity)** → Defines the structure of your data (e.g., Student object).
- **Service** → Contains the logic to process data (e.g., storing/retrieving students in a HashMap).
- **Controller** → Exposes endpoints to the outside world (GET, POST, etc.).

## Part 5: Creating the Student Model

Now that our architecture is ready, let's build our first **Model**: the **Student** object.

### 🎯 Why start with the Model?

In software development, the **Model** represents the actual data of our application. For our case, this will be the **Student** information. Later, this object will be passed around between the **Service** and **Controller**.

---

### 🛠️ Step 1: Create the Model Folder

Inside your project folder structure, navigate to:

```
src/main/java/com/accenture/studentapi/model
```

Right-click → **New** → **Java Class** → name it **Student**.

## 🔧 Step 2: Add Attributes

Each student needs to have:

- **pkStudentID (long)** → This acts as the **Primary Key (PK)**, a unique identifier for each student.
- **name (String)** → The name of the student.
- **course (String)** → The course the student is enrolled in.

👉 All of these should be **private**. This is part of **Encapsulation**, one of the four pillars of OOP.

Encapsulation means we hide the internal data (fields) and only allow controlled access through methods.

---

## ⚙️ Step 3: Add Constructors

Constructors help us create objects.

- **Default constructor** → An empty constructor that allows frameworks like Spring Boot to initialize the object if needed.
  - **Parameterized constructor** → A constructor that allows us to pass values (ID, name, course) right away when creating a student.
- 

## ✂️ Step 4: Generate Getters and Setters

Since our attributes are **private**, we cannot access them directly.

To solve this, we add:

- **Getter methods** → Allow reading the value of a private field.
- **Setter methods** → Allow updating the value of a private field.

👉 This ensures data protection and controlled access.

## Part 6: Creating the Service Layer


The **Service Layer** is where we put our **business logic**. Instead of putting logic directly in the **Controller**, we keep it separate for **clean code** and **scalability**.

Here, we will also apply the **OOP Pillar: Abstraction**.

---

### What is Abstraction?

- **Abstraction** means **hiding the implementation details** and exposing only the necessary methods.
- Think of it like a **TV remote**: you press a button (method), but you don't need to know how the circuits inside the TV work.

 In Java, we achieve abstraction using **Interfaces** or **Abstract Classes**.

---



### What if we don't use Abstraction?

- If we don't separate the **what** (methods) from the **how** (implementation), our Controller will directly depend on a concrete class.
  - This makes the code harder to maintain, test, or replace.
  - By using an **Interface**, we can easily **swap implementations** in the future (for example: replacing HashMap with a real Database).
- 

### Step 1: Create StudentService Interface




Inside the service package, create a new file named **StudentService.java**.

This interface will define the operations we expect in our service layer:

-  `getAllStudents()` → returns a **List of Students**  
*Expected HTTP method: GET*  
*Success code: 200 OK*  
*Fail code: 404 NOT FOUND (if no students found)*
-  `getStudentById(Long id)` → returns an **Optional<Student>**  
*Expected HTTP method: GET*

Success code: 200 OK

Fail code: 404 NOT FOUND (if student doesn't exist)

-  addStudent(Student student) → returns the **Student object** (the one added)  
Expected HTTP method: **POST**  
Success code: 201 CREATED  
Fail code: 400 BAD REQUEST (if request body is invalid)
  -  updateStudent(Long id, Student student) → returns the **updated Student**  
Expected HTTP method: **PUT**  
Success code: 200 OK  
Fail code: 404 NOT FOUND (if student doesn't exist)
  -  deleteStudent(Long id) → returns a **boolean** (true if deleted, false if not found)  
Expected HTTP method: **DELETE**  
Success code: 204 NO CONTENT (successfully deleted)  
Fail code: 404 NOT FOUND (if student doesn't exist)
- 

## ✂ Step 2: Create StudentServiceImpl

Inside service, create a new folder named **impl**.

Inside it, create a class **StudentServiceImpl.java**.

This class must:

- Use implements StudentService → meaning it **must provide actual code** for all methods defined in the interface.
  - Be annotated with @Service.
- 

## What is @Service Annotation?

- @Service tells Spring Boot: *"This class is a service component. Manage its lifecycle for me."*
- Without @Service, Spring won't recognize this class as a **bean**, and dependency injection will fail. (Meaning, our Controller won't be able to use it automatically.)

## Thought Process for Each Method

- Inside `StudentServiceImpl`, we'll simulate a database using a **`HashMap<Long, Student>`**

```
java                                                                    Copy Edit

private Map<Long, Student> studentDB = new HashMap<>();
```

#### **`getAllStudents()`** →

*Thought:* Retrieve all values from the `HashMap` and return as a `List`.

#### **`getStudentById(Long id)`** →

*Thought:* Check if the given `id` exists in the `HashMap`, wrap it in an `Optional` to safely handle null cases.

#### **`addStudent(Student student)`** →

*Thought:* Store the `Student` in the `HashMap` using `pkStudentID` as the key, then return the same object.

#### **`updateStudent(Long id, Student student)`** →

*Thought:* Check if a student with that `id` exists. If yes, update the record and return the updated object. If not, handle failure (return null or throw exception).

#### **`deleteStudent(Long id)`** →

*Thought:* Check if student exists in the `HashMap`. If yes, remove it and return true. If not found, return false.

## **Part 7: Creating the Controller Layer**

The **Controller Layer** is where we define our **REST API endpoints**.

- It connects the **outside world (clients, Postman, browsers, other apps)** with our **service layer**.
  - The Controller doesn't handle business logic – it just forwards requests to the Service and sends back responses.
-

## What is a RESTful Controller?

A **RESTful Controller** in Spring Boot is simply a class annotated with:

- **@RestController** → Tells Spring that this class will handle web requests and return responses in **JSON** format (instead of HTML).
- **@RequestMapping("/api")** → Defines the **base URL** for all endpoints inside this controller.

👉 Example: If base URL is /api and endpoint is /students, the final path is:

```
http://localhost:8080/api/students
```

### Step 1: Create the Controller

Inside your **controller** package, create a class called **StudentController**.

- Annotate it with @RestController.
- Add @RequestMapping("/api") so all endpoints will start with /api.
- Connect it to the **StudentService** so that the controller can call the methods we already defined.

---

## Step 2: Defining Endpoints

Now let's plan out the operations we want. Remember, we are not writing code yet – just understanding the scenarios.

---

### 1 Get All Students

- **Scenario:** A client wants to see all students stored in our system.
  - **HTTP Method:** GET
  - **Endpoint:** /api/students
  - **Logic:** The controller calls the service to fetch all students. The service then retrieves all records from our HashMap and returns them as a list.
-



## 2 Get Student by ID

- **Scenario:** A client wants details of a specific student.
  - **HTTP Method:** GET
  - **Endpoint:** /api/students/{id}
  - **Logic:**
    - The controller receives the student ID from the URL.
    - The service looks inside the HashMap to check if a student exists with that ID.
    - If found → return the student.
    - If not found → return nothing or a message saying “Student not found.”
- 

## 3 Add a New Student

- **Scenario:** A client wants to register a new student.
  - **HTTP Method:** POST
  - **Endpoint:** /api/students
  - **Logic:**
    - The client sends a JSON request body with student details (id, name, course).
    - The controller receives it and passes it to the service.
    - The service adds this new student into the HashMap.
    - Return the new student as confirmation.
- 

## 4 Update Student Information

- **Scenario:** A client wants to update a student’s information.
- **HTTP Method:** PUT
- **Endpoint:** /api/students/{id}
- **Logic:**
  - First, the controller takes the ID from the URL.
  - The service checks if a student with this ID exists.
  - If **not found** → return an error or message “Student not found.”
  - If **found** → proceed with updating.
  - Important: In updates, clients may not always send complete information (e.g., maybe only course is provided but not name).
    - In this case, we must **only update the provided fields** and leave the rest unchanged.

- Example: If John Doe already exists and client only sends { "course": "Math" }, then the course will update to Math but name remains John Doe.
  - Finally, return the updated student details.
- 

## **5 Delete a Student**

- **Scenario:** A client wants to remove a student from the system.
  - **HTTP Method:** DELETE
  - **Endpoint:** /api/students/{id}
  - **Logic:**
    - The controller takes the ID from the URL.
    - The service checks if that ID exists in the HashMap.
    - If **found** → remove the student from the HashMap and confirm deletion.
    - If **not found** → inform the client that the student does not exist.
- 

## **Step 3: Testing in Postman**

Once all endpoints are created, here's how we test them step by step in **Postman**:

1. **Get All Students**
  - Method: GET
  - URL: http://localhost:8080/api/students
2. **Add New Student**
  - Method: POST
  - URL: http://localhost:8080/api/students
  - Body (JSON):