# **Scaffolding a Spring Boot REST API with Maven Using GitHub Copilot**

## **Overview**

In this guide you will build an Employee Directory REST API for an HR department. The API will manage employee records in memory and expose endpoints to list, retrieve, and add employees. The goal is to simulate a real-world scenario where rapid prototyping, robust error handling, and clean code are essential.

Throughout this exercise, you’ll rely on GitHub Copilot to:

* Generate boilerplate code and configuration files
* Provide explanations and troubleshooting advice
* Assist with writing tests and Git commands

Each step includes sample prompts and challenges that encourage you to craft your own prompts using a guiding structure

## **Prerequisites**

Before you begin, ensure you have:

* **Java JDK 17 or later**
* **Maven** installed
* **GitHub Copilot** enabled in your editor (e.g., VS Code or IntelliJ IDEA)
* Basic familiarity with Spring Boot, REST APIs, and Maven
* An active GitHub account

|  |
| --- |
| **Note**: Below prompts are provided solely as examples of effective prompt construction. As participants, we must engage in an iterative process to refine these examples and develop the optimal prompt for the task at hand |

## **Step 1: Set Up the Project Workspace**

### **Objective**

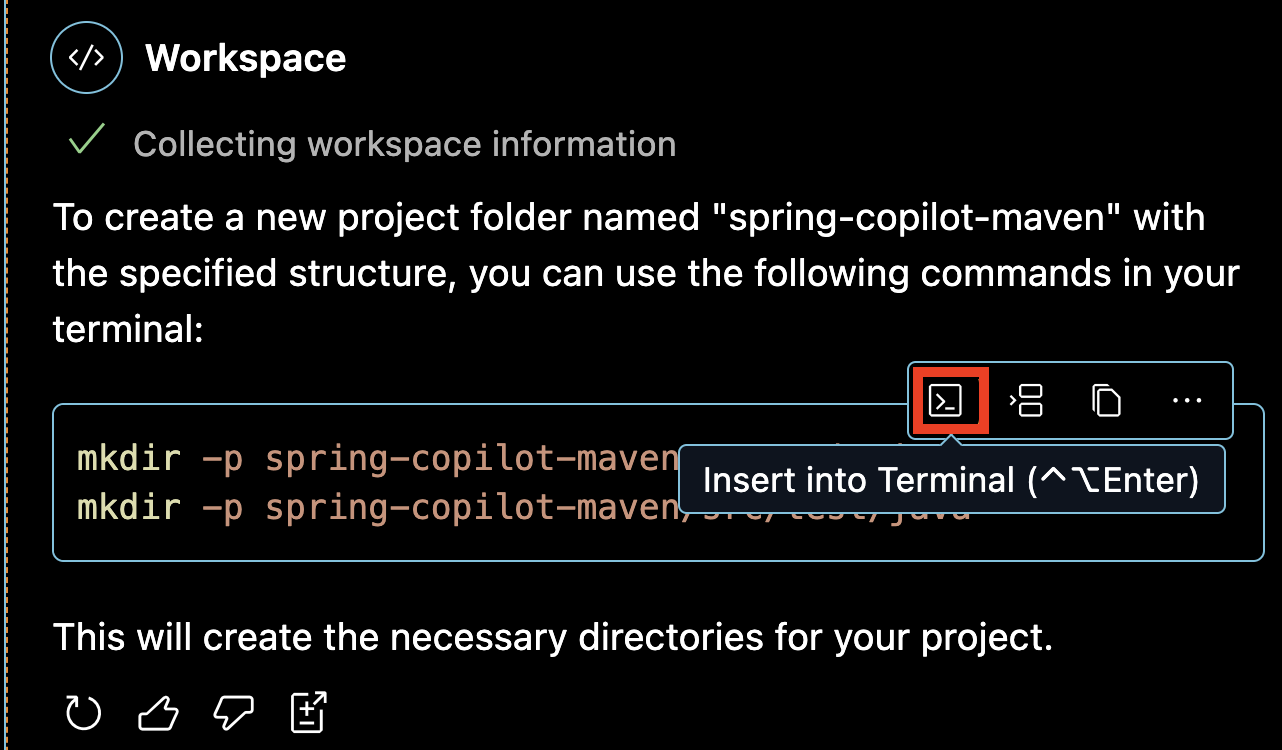
Create the overall project folder and directory structure using only GitHub Copilot Chat.

### **How to Proceed**

Begin by opening **GitHub Copilot Chat in VS Code** and enter a command like:

|  |
| --- |
| @workspace: Create a new project folder named "spring-copilot-maven" with the following structure: - src/main/java - src/test/java |

|  |
| --- |
| **Note**: Above prompt would help you to generate the required commands. Once commands are created, hover on the commands and click on **Insert into Terminal** option |



This command establishes the core folders for your Java source and test files. Once you review the initial output, you might notice that certain common directories—such as src/main/resources for configuration files and static assets—are missing. To address this gap, extend your workspace with:

|  |
| --- |
| /fix: Add the missing directory "src/main/resources" to the project structure |

After that, you may also consider adding folders for test resources or documentation. For instance:

|  |
| --- |
| /fix: Also add "src/test/resources" and a "docs" folder to the project structure |

Finally, verify your updated structure by asking:

|  |
| --- |
| @workspace: List all directories in my project |

### **Challenges for Step 1**

* **Challenge 1:** Review the resulting directory structure and determine if additional folders (e.g., a scripts folder for automation or a config folder for environment-specific settings) could benefit your project. Adjust your commands accordingly until your workspace fully meets your project needs.
* **Challenge 2:** Imagine how you would organize a multi-module project. Consider experimenting with a more complex directory structure and refining your prompts to match that scenario.

**Step 2: Initialize the Maven Project (pom.xml)**

### **Objective**

Generate a Maven configuration file that manages your project’s dependencies and build settings.

### **How to Proceed**

Start by issuing a command on **GitHub Copilot Chat in VS Code** such as:

|  |
| --- |
| /start: Create a pom.xml file for a Spring Boot project with the following configuration: - groupId: com.example - artifactId: employee-api - version: 0.0.1-SNAPSHOT - Parent: spring-boot-starter-parent version 3.1.3 - Java version: 17 - Dependencies: spring-boot-starter-web and spring-boot-starter-test (scope: test) - Build Plugin: spring-boot-maven-plugin |

This prompt produces the basic Maven configuration necessary to build your project. Once generated, you might decide that the pom.xml could be further enhanced by including additional properties or by configuring the Maven compiler plugin properly. Refine your file with:

|  |
| --- |
| /fix: Refine the pom.xml to include additional properties and configuration settings that adhere to Maven best practices |

Later, if you decide that monitoring your application is important, you can extend the configuration by adding:

|  |
| --- |
| /start: Add the dependency for spring-boot-starter-actuator to the pom.xml |

For more clarity on what each dependency does, you may ask:

|  |
| --- |
| /explain: Provide a brief description for each dependency defined in my pom.xml |

### **Challenges for Step 2**

* **Challenge 1:** Modify your pom.xml to include additional plugins, such as those for code coverage, static analysis, or database migration. Document why each plugin is necessary and refine your configuration until it is comprehensive.
* **Challenge 2:** Customize the pom.xml to support multiple environments (development, testing, production) by using Maven profiles. Experiment with different configurations and refine your file until it is flexible for various deployment scenarios.

**Step 3: Create the Main Application Class**

### **Objective**

Build the main entry point of your Spring Boot application to initialize and launch your project.

### **How to Proceed**

Begin with a basic command to generate the main application class within **GitHub Copilot Chat**:

|  |
| --- |
| /start: Create the main application class EmployeeApiApplication.java in the package com.example.employeeapi with: - @SpringBootApplication annotation - A main method that calls SpringApplication.run(EmployeeApiApplication.class, args) |

This command produces a functional main class. To enhance its utility, you might want to include a diagnostic message to confirm that the application has started. For that, issue:

|  |
| --- |
| /start: Add a CommandLineRunner bean to EmployeeApiApplication that prints "Application started successfully" on startup |

If you want deeper insight into the configuration, you can further ask:

|  |
| --- |
| /explain: Explain the purpose of the @SpringBootApplication annotation and how it simplifies configuration compared to using @Configuration, @EnableAutoConfiguration, and @ComponentScan separately |

### **Challenges for Step 3**

* **Challenge 1:** Integrate detailed logging into your main application class. Modify the CommandLineRunner to log not only a startup message but also key configuration details using SLF4J, and explain how this improves production monitoring.
* **Challenge 2:** Experiment with alternative startup configurations (for example, setting custom banners or active profiles) and refine your main class until it provides comprehensive diagnostics.

**Step 4: Create the Employee Model and Service**

### **Employee Model**

#### **Objective**

Define a robust model class that represents an Employee, including all necessary fields, methods, and validations.

#### **How to Proceed**

Open or create the Employee.java file within **GitHub Copilot Chat in VS Code**:

|  |
| --- |
| @workspace #file: Open or create Employee.java in src/main/java/com/example/employeeapi |

Then generate the model class using the following sample prompt:

|  |
| --- |
| /start: Generate a class named Employee with private fields: - id (int) - name (String) - department (String) - email (String) Include a constructor, getters, and setters |

|  |
| --- |
| **Note**: Above prompt would help you to generate the required commands. Once commands are created, hover on the commands and click on **Insert into Terminal** option |

After generating the basic class, review the output and note that inline documentation would improve maintainability. Enhance the file by adding:

|  |
| --- |
| /fix: Add Javadoc comments to the constructor and all getter and setter methods in Employee.java |

To further increase robustness, include data validation:

|  |
| --- |
| /fix: Add validation annotations (e.g., @NotNull, @Size) to the fields in Employee.java |

#### **Challenges for the Employee Model**

* **Challenge 1:** Extend the Employee model by incorporating business logic, such as a method that returns a formatted summary of the employee’s information. Adjust your model until it is robust and self-explanatory.
* **Challenge 2:** Consider edge cases such as null values or improperly formatted inputs. Add error handling or default values as necessary to ensure data consistency.

### **Employee Service**

#### **Objective**

Create a service layer that manages Employee data, implementing core CRUD operations with robust error handling.

#### **How to Proceed**

Now begin by creating the service with a command in **GitHub Copilot Chat** like:

|  |
| --- |
| /start: Create an EmployeeService class in the package com.example.employeeapi. The service should: - Be annotated with @Service - Maintain a private List<Employee> for employee records - Initialize sample data (for example, add two employees with IDs 1 and 2) - Provide methods: getAllEmployees, getEmployeeById, and addEmployee (assigning a new unique ID) |

|  |
| --- |
| **Note**: Above prompt would help you to generate the required commands. Once commands are created, hover on the commands and click on **Insert into Terminal** option |

Once the basic service is in place, consider adding functionality to update employee records:

|  |
| --- |
| /start: Add an updateEmployee method to EmployeeService that updates an employee's information based on a provided Employee object |

Enhance error handling by modifying the retrieval method:

|  |
| --- |
| /fix: Modify getEmployeeById in EmployeeService to throw an EmployeeNotFoundException if the employee is not found |

#### **Challenges for the Employee Service**

* **Challenge 1:** Extend the service to support filtering employees by department. Create a method that returns a list of employees based on a given department query.
* **Challenge 2:** Consider integrating persistence by replacing the in-memory list with a simple database solution. Experiment with these changes and refine your prompts until you achieve a working implementation.

**Step 5: Create the REST Controller**

### **Objective**

Expose your EmployeeService through RESTful endpoints to create a complete API layer.

### **How to Proceed**

Generate a basic REST controller using **GitHub Copilot Chat in VS Code** like:

|  |
| --- |
| /start: Create a REST controller named EmployeeController in the package com.example.employeeapi with the following endpoints: - GET /api/employees: Returns all employees - GET /api/employees/{id}: Returns an employee by ID with proper error handling - POST /api/employees: Adds a new employee |

|  |
| --- |
| **Note**: Above prompt would help you to generate the required commands. Once commands are created, hover on the commands and click on **Insert into Terminal** option |

Review the output and if you see that the GET by ID endpoint could be more robust, refine it:

|  |
| --- |
| /fix: Enhance the GET /api/employees/{id} endpoint in EmployeeController to return a 404 response when an employee is not found |

Next, add additional functionality by creating a search endpoint:

|  |
| --- |
| /start: Add a GET endpoint at /search that accepts a query parameter "department" and returns employees matching that department |

|  |
| --- |
| **Note**: Above prompt would help you to generate the required commands. Once commands are created, hover on the commands and click on **Insert into Terminal** option |

To further ensure data integrity, improve the POST endpoint’s handling of invalid data with:

|  |
| --- |
| /fix: Add input validation annotations to the Employee model and update the POST endpoint in EmployeeController to handle invalid data |

#### **Challenges for the REST Controller**

* **Challenge 1:** Add endpoints for updating (PUT) and deleting (DELETE) employee records. Ensure that each endpoint returns appropriate status codes and messages, and test these endpoints using curl commands or integration tests.
* **Challenge 2:** Integrate comprehensive logging within your controller. Modify your endpoints to log key actions—such as employee creation, updates, and error scenarios—and verify that the logs provide sufficient detail for troubleshooting.

**Step 6: Run and Test Your Application (Optional)**

### **Objective**

Launch your application and thoroughly test all endpoints to ensure robust functionality.

### **How to Proceed**

Start by running your application using **GitHub Copilot Chat in VS Code**:

|  |
| --- |
| #terminalLastCommand: Explain the output of 'mvn spring-boot:run' and suggest corrective steps if any errors occur |

Note: This Github Copilot command would help you to get an explanation of the maven command and you

Review the startup logs carefully. If errors occur, diagnose them with:

|  |
| --- |
| /fix: Identify and resolve any issues shown in the output of 'mvn spring-boot:run' |

Next, test your API endpoints. Begin by generating a curl command for the GET endpoint:

|  |
| --- |
| /test: Generate a sample curl command to test the GET /api/employees endpoint |

Test error handling by simulating a request for a non-existent employee:

|  |
| --- |
| /test: Generate a curl command to test that requesting a non-existent employee returns a 404 error |

Finally, build an integration test to cover all endpoints:

|  |
| --- |
| /test: Generate an integration test class for EmployeeController using @SpringBootTest |

#### **Challenges for Testing**

* **Challenge 1:** Develop tests for all endpoints, including PUT and DELETE (if implemented). Ensure that your tests cover both valid and invalid scenarios, and refine your endpoints based on test results.
* **Challenge 2:** Create edge case tests for scenarios like invalid input data, missing fields, and improper formatting. Adjust your test suite through iterative prompts until it robustly covers all potential issues.

**Additional Advanced Challenges**

Once your basic API is fully operational and tested, consider tackling these advanced challenges to further enhance your project:

* **Challenge A: Implement Update Functionality**Extend your API by adding a PUT endpoint in EmployeeController to update an employee’s details. Ensure the endpoint updates data if the employee is found or returns a 404 error if not, and test the functionality thoroughly.
* **Challenge B: Enhance Logging and Exception Handling**Integrate SLF4J logging into your controller and create a global exception handler using @ControllerAdvice. This should standardize error responses and produce detailed logs for troubleshooting.
* **Challenge C: Add Caching**Integrate Spring Cache in EmployeeService to cache the results of getAllEmployees. Verify that the cache invalidates properly when employee data changes.
* **Challenge D: Prepare the Application for Production**Generate a Dockerfile to containerize your Spring Boot application. Write a detailed commit message describing the production-ready state of your project, including environment configurations, security settings, and any additional relevant details.

## **Conclusion**

This guide has walked you through each step of creating a Spring Boot REST API using Maven with GitHub Copilot. You have:

* Set up a robust Maven project.
* Generated core components (main class, model, service, controller) using inline suggestions.
* Leveraged Copilot Chat with optimized prompts for explanations, troubleshooting, and extended code generation.
* Completed challenges that encourage you to write your own prompts and think critically about extending functionality.

By practicing with both the provided examples and writing your own prompts based on the guiding structure, you will become more adept at using GitHub Copilot to boost productivity, minimize errors, and tackle real-world coding challenges.

Happy coding, and enjoy your journey to efficient, professional development with GitHub Copilot!