**Building RESTful APIs in Java: A Step-by-Step Tutorial**

<https://medium.com/javarevisited/building-restful-apis-in-java-a-step-by-step-tutorial-e1b9b2d3e6ab>

[Anshumaan Tiwari](https://medium.com/@anshumaantiwari123?source=post_page-----e1b9b2d3e6ab--------------------------------)

·

Follow

Published in

[Javarevisited](https://medium.com/javarevisited?source=post_page-----e1b9b2d3e6ab--------------------------------)

·

5 min read

·

Aug 1, 2023

A purple sky with stars

Description automatically generated

Photo by [Vincentiu Solomon](https://unsplash.com/@vincentiu?utm_source=medium&utm_medium=referral" \t "_blank) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

**Introduction:**

I am delighted to share my knowledge on building RESTful APIs using Java. RESTful APIs have become a fundamental part of modern web development, allowing applications to communicate and share data seamlessly. In this tutorial, we will create a simple yet powerful RESTful API using Java, Spring Boot, and Maven. By the end of this guide, you will have a solid foundation to build sophisticated APIs that cater to diverse needs.

**What is a REST-API?**

A close-up of a sound board

Description automatically generated

Photo by [Yassine Khalfalli](https://unsplash.com/@yassine_khalfalli?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Imagine you have a magical snack stand that can give you tasty treats when you ask for them. A RESTful API works just like that, but instead of snacks, it gives you information from a computer

The word “API” stands for “Application Programming Interface.” It’s like a special window that lets different computer programs talk to each other.

Now, let’s talk about the “RESTful” part. Think of it as a way for computers to have a friendly chat, just like friends do. When you want information, like the weather or a fun fact, you ask your friend, and they give you an answer. A RESTful API lets one computer (like a friend) ask another computer (like the snack stand) for information, and it gets an answer in a way they both understand.

For example, if a computer wants to know the weather in a city, it can ask a RESTful API for that information. The API will understand the question and send back the weather details in a clear and organized way. Just like how your friend tells you the weather for the day!

So, a RESTful API is like a friendly helper that lets computers talk to each other and share information in a way that’s easy for them to understand. It’s like having a magical connection between different computers, making them all work together to provide us with the information we need. Isn’t that amazing?

**Prerequisites:**

Before we begin, ensure you have the following prerequisites in place:

1. **Java Development Kit (JDK) 8 or later**installed on your system.

2. An Integrated Development Environment (IDE) like Eclipse or IntelliJ IDEA.

3. Familiarity with **Java programming,Spring and HTTP** concepts.

**Step 1: Setting Up the Project**

Let’s create a new Spring Boot project using Maven to kickstart our API development journey.

1. Open your IDE and **create a new Maven project.**

2. Choose “**New” -> “Maven Project**” and follow the prompts to set up the project.

3. Ensure you select the **Spring Boot archetype during project creation**.

**Step 2: Defining the Model**

A crucial aspect of building APIs is defining the data model. For this tutorial, let’s create a simple model representing a “Product” entity.

package com.example.api.model;  
  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
  
@Entity  
public class Product {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
  
 private String name;  
 private String description;  
 private double price;  
  
 // Constructors  
 public Product() {  
 // Empty constructor needed for JPA  
 }  
  
 public Product(String name, String description, double price) {  
 this.name = name;  
 this.description = description;  
 this.price = price;  
 }  
  
 // Getters and Setters  
 public Long getId() {  
 return id;  
 }  
  
 public void setId(Long id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getDescription() {  
 return description;  
 }  
  
 public void setDescription(String description) {  
 this.description = description;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 }  
  
 // Other methods (if needed) can be added here  
}

**Step 3: Implementing the Repository**

Now, let’s create a repository interface that will handle CRUD operations for our Product entity.

package com.example.api.repository;  
  
import com.example.api.model.Product;  
import org.springframework.data.jpa.repository.JpaRepository;  
  
public interface ProductRepository extends JpaRepository<Product, Long> {  
  
}

**Step 4: Building the Service Layer**

The service layer acts as a bridge between the controller and the data access layer. Implement the ProductService interface and its implementation.

package com.example.api.service;  
  
import com.example.api.model.Product;  
  
public interface ProductService {  
 Product getProductById(Long id);  
 void saveProduct(Product product);  
 void updateProduct(Product product);  
 void deleteProduct(Long id);  
}

package com.example.api.service;  
  
import com.example.api.model.Product;  
import com.example.api.repository.ProductRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
import java.util.Optional;  
  
@Service  
public class ProductServiceImpl implements ProductService {  
  
 private final ProductRepository productRepository;  
  
 @Autowired  
 public ProductServiceImpl(ProductRepository productRepository) {  
 this.productRepository = productRepository;  
 }  
  
 @Override  
 public Product getProductById(Long id) {  
 Optional<Product> optionalProduct = productRepository.findById(id);  
 return optionalProduct.orElse(null);  
 }  
  
 @Override  
 public void saveProduct(Product product) {  
 productRepository.save(product);  
 }  
  
 @Override  
 public void updateProduct(Product product) {  
 productRepository.save(product);  
 }  
  
 @Override  
 public void deleteProduct(Long id) {  
 productRepository.deleteById(id);  
 }  
  
 // Additional method to get all products (optional)  
 public List<Product> getAllProducts() {  
 return productRepository.findAll();  
 }  
}

**Step 5: Creating the Controller**

The controller handles incoming HTTP requests and directs them to the appropriate service methods. Define the ProductController class as follows:

package com.example.api.controller;  
  
import com.example.api.model.Product;  
import com.example.api.service.ProductService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/api/products")  
public class ProductController {  
  
 private final ProductService productService;  
  
 @Autowired  
 public ProductController(ProductService productService) {  
 this.productService = productService;  
 }  
  
 //Implement the Endpoints  
}

**Step 6: Implementing the Endpoints**

A person with white hair and black hat

Description automatically generated

Now, let’s implement the CRUD operations for the Product entity in the ProductController class.

package com.example.api.controller;  
  
import com.example.api.model.Product;  
import com.example.api.service.ProductService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/api/products")  
public class ProductController {  
  
 private final ProductService productService;  
  
 @Autowired  
 public ProductController(ProductService productService) {  
 this.productService = productService;  
 }  
  
 @GetMapping("/{id}")  
 public Product getProductById(@PathVariable Long id) {  
 return productService.getProductById(id);  
 }  
  
 @GetMapping  
 public List<Product> getAllProducts() {  
 return productService.getAllProducts();  
 }  
  
 @PostMapping  
 public void addProduct(@RequestBody Product product) {  
 productService.saveProduct(product);  
 }  
  
 @PutMapping("/{id}")  
 public void updateProduct(@PathVariable Long id, @RequestBody Product product) {  
 Product existingProduct = productService.getProductById(id);  
 if (existingProduct != null) {  
 product.setId(id); // Ensure the correct ID is set  
 productService.saveProduct(product);  
 }  
 }  
  
 @DeleteMapping("/{id}")  
 public void deleteProduct(@PathVariable Long id) {  
 productService.deleteProduct(id);  
 }  
}

Set up the database configuration and other necessary configurations in application.properties.

# Database Configuration  
spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.username=sa  
spring.datasource.password=  
spring.datasource.driver-class-name=org.h2.Driver  
  
# Hibernate Properties  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.show-sql=true

**Step 8: Testing the API**

You can now test your API using tools like Postman or cURL. Verify the endpoints for creating, reading, updating, and deleting product data.

**Conclusion:**

In this tutorial, we successfully built a RESTful API in Java using Spring Boot and Maven. We defined the data model, implemented the controller, service, and repository layers, and tested our API endpoints. RESTful APIs are essential for developing scalable and interconnected applications, and Java, with the Spring framework, provides a robust platform for this purpose. As you advance, explore more features of Spring, add security measures, and handle advanced use cases to create robust APIs that cater to real-world scenarios. Happy coding!

A person with white hair and a purple jacket

Description automatically generated

YO WAI MO

**THANKS GUYS FOR GIVING YOUR PRECIOUS TIME TO READ MY ARTICLE**

[Software Development](https://medium.com/tag/software-development?source=post_page-----e1b9b2d3e6ab---------------software_development-----------------)

[Programming](https://medium.com/tag/programming?source=post_page-----e1b9b2d3e6ab---------------programming-----------------)

[Tech](https://medium.com/tag/tech?source=post_page-----e1b9b2d3e6ab---------------tech-----------------)

[Technology](https://medium.com/tag/technology?source=post_page-----e1b9b2d3e6ab---------------technology-----------------)

[Software Engineering](https://medium.com/tag/software-engineering?source=post_page-----e1b9b2d3e6ab---------------software_engineering-----------------)

286

1

[[](https://medium.com/@anshumaantiwari123?source=post_page-----e1b9b2d3e6ab--------------------------------)](https://medium.com/@anshumaantiwari123?source=post_page-----e1b9b2d3e6ab--------------------------------)

[[](https://medium.com/javarevisited?source=post_page-----e1b9b2d3e6ab--------------------------------)](https://medium.com/javarevisited?source=post_page-----e1b9b2d3e6ab--------------------------------)

[Written by Anshumaan Tiwari](https://medium.com/@anshumaantiwari123?source=post_page-----e1b9b2d3e6ab--------------------------------)

[58 Followers](https://medium.com/@anshumaantiwari123/followers?source=post_page-----e1b9b2d3e6ab--------------------------------)

·Writer for

[Javarevisited](https://medium.com/javarevisited?source=post_page-----e1b9b2d3e6ab--------------------------------)

Software Developer having a little passion for technical content writing

Follow

More from Anshumaan Tiwari and Javarevisited