Java Spring Series

Spring Boot API:
Insurance Quote Application

Build APIs that work -
clearly, confidently, and with purpose.

Gerry Byrne

Spring Boot API

Insurance Quote Application

Step by step instructions
for practical hands-on programming

Gerry Byrne

ABOUT THE AUTHOR

Gerry Byrne is a Senior Technical Trainer at a Forbes 100 company, where he specializes in upskilling software engineers who build business critical applications. With a long career as a teacher, lecturer, and corporate trainer, Gerry brings a rare blend of academic rigor and real-world insight to his instruction.

He has delivered technical training across a wide spectrum of languages and frameworks, including Java, Spring, C#, Python, and JavaScript. His expertise in Spring Boot and API development is grounded in years of hands-on experience teaching modern enterprise technologies to engineers at all stages of their careers, from new graduates to those re-entering the workforce, to seasoned professionals.

Gerry's approach to training emphasizes clarity, practical application, and a deep understanding of how software development fits into commercial environments. Whether introducing Test-Driven Development or guiding teams through the intricacies of RESTful API design, he equips learners with the tools they need to thrive in today's fast-paced tech landscape.

DEDICATION

Writing a book is a rewarding undertaking, but it requires time, effort and patience. It requires patience from those who help you write the book and those around you in your life.

So, I start by thanking my family for 'facilitating' me as I worked over many hours, days, weeks and months to write this book.

ACKNOWLEDGMENTS

Writing this book has been a journey, one shaped not only by years of teaching and training, but also by the people who have inspired, challenged, and supported me along the way.

To my colleagues, and the learners I have worked with, thank you for your curiosity, your questions, and your willingness to dive deep into the complexities of software development. You've kept me sharp and reminded me why teaching is a privilege.

To the technical training teams I have worked with over the years, your dedication to excellence and your passion for empowering others have been a constant source of motivation.

I also wish to thank those who have taught me programming over many years and shared their knowledge. I have learnt so much from them and in writing this book their imprint exists.

To the readers of this book, whether you are just starting out, or refining your Spring Boot skills, I hope these pages help you build something meaningful. Thank you for letting me be part of your learning journey.

Table of Contents

Introduction

[1]Embarking on the journey to becoming proficient in Spring Boot, Test-Driven Development (TDD), and Java is akin to training for a marathon. Think of yourself as an athlete preparing for a marathon. Just as a fitness tracker helps you monitor your physical activity, set goals, and track progress, this book will guide you through the steps of mastering these technologies.

A fitness tracker provides insights into your daily steps, exercise routines, and overall health, helping you stay on track and improve over a period of time. Similarly, this book will offer structured lessons with practical coding, and real-world examples to help you build your skills incrementally. By following a disciplined approach, much like adhering to a fitness régime, you'll develop a strong foundation in Spring Boot, Java and TDD. We will do this by developing an application and microservices, enabling you to tackle complex software development challenges with confidence.

Along the way, you will encounter challenges and obstacles, much like the bumps and hurdles faced during physical training. Some may find that they grasp concepts quickly and progress rapidly, while others may take a bit longer to get into the rhythm. This is perfectly normal and should not be a cause for discouragement. We all have different learning curves. The key to success lies in perseverance and commitment. Just as consistent training and dedication can lead to improved physical fitness, a steadfast approach to learning and spending time writing code for Spring Boot, Java and TDD, will ultimately lead to proficiency.

Remember, the path to becoming 'fit' in software development will not always be smooth. There will be moments of frustration and setbacks, but these are part of the learning process. Embrace the challenges, see them as opportunities to grow and strengthen your skills. With determination and a positive mindset, you will reach your end goal, equipped with the knowledge and confidence to tackle complex software development projects.

Mastering Spring Boot, Java and TDD will not only enhance your technical skills but also broaden your understanding of widely used industry technologies. Proficiency in these areas will equip you with the knowledge to contribute to high-quality, maintainable, and scalable software solutions. By the end of this journey, you'll have a strong foundation to tackle complex software development challenges with confidence.

---

[1] Tracker icons created by Eucalyp - Flaticon

[2]What you will learn

On completing the learning, you will:

- Read Agile user stories and gherkins.
- Use JUnit to create unit tests and develop Java code.
- Use IntelliJ to create a Maven Spring Boot application.
- Understand the Maven project structure and manage dependencies.
- Create the structure of a Spring Boot project.
- Create and configure JPA entities.
- Use annotations like @Entity, @Table, @Id, and @GeneratedValue.
- Create repository interfaces extending JpaRepository.
- Implement custom query methods using Spring Data JPA.
- Implement business logic in service classes.
- Understand dependency injection using constructor injection.
- Create RESTful endpoints using @RestController, @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping.
- Handle HTTP requests and responses.
- Handle exceptions in service and controller layers.
- Create custom exceptions.
- Configure database connections.
- Implement logging using SLF4J.
- Configure application properties and logging levels.
- Test the RESTful endpoints using Postman or cURL commands or Swagger.
- Perform CRUD operations on a database using JPA.
- Run a Spring Boot application from the IDE or command line.
- Understand the main application class and its role.
- Build and use microservices.
- Implement pagination and sorting in REST endpoints to efficiently manage and organize large sets of data.

[2] Goal icons created by Uniconlabs - Flaticon

- Apply validation annotations to class fields in a 'model' and use @Valid in our controllers to ensure that only valid data is processed and stored.
- Create dynamic filtering and search functionality, allowing users to retrieve records based on various optional criteria.
- Develop a global exception handler to provide consistent and informative error responses across the application.
- Use Data Transfer Objects (DTOs) to encapsulate and transfer data between different layers of the application.
- Write unit tests for the service layer using Mockito to mock dependencies and verify business logic.
- Create integration tests to ensure that controllers and services work together as expected.
- Integrate a simple frontend using HTML and JavaScript to demonstrate how the backend API can be consumed and interacted with.
- Connect to external microservices, such as customer and product services, to enrich the application's functionality.
- Configure the application properties to manage environment-specific settings and improve maintainability.
- Implement logging to monitor application behavior and assist with debugging and maintenance.
- Understand how to package and deploy the application using Maven to facilitate easy distribution and deployment.
- Generate API documentation using Swagger/OpenAPI to make endpoints easy to understand and consume.

Introduction - Spring Boot Electrical Items Insurance

Using Spring Boot, Java, and Test-Driven Development (TDD), we will create an application for managing insurance related to electrical items. The generic workflow we will follow is to create the models, then the repositories, then the services, then the controllers, and finally the main application class. When we use a DTO layer, we can create the DTO classes after we create the model since a DTO is a view of the model. The main application class will be called InsuredItem and will have the properties: product_type, product_value, quote_amount and customer_account_number. A sample is shown below:

product_type	Mobile Phone
product_value	1200.00
quote_amount	100.00
customer_account_number	ACC123

Once we have this first part of the application developed, we will extend our knowledge by creating two more microservices. The first microservice will be an application that holds details of products i.e., the product type and a description of the policy coverage e.g.,

Product type	Camera
Product description	This policy covers accidental damage, liquid damage, fire, and theft of the insured camera. Coverage includes damage resulting from drops, spills, and electrical surges. Theft is covered if the camera was stored securely. Normal wear and tear, careless handling, unauthorized modifications, and mechanical breakdowns not caused by an insured event are excluded.

Here the product type is Camera, and the description indicates what is and is not covered as part of the policy and quote amount.

The second microservice will be an application that holds details of customers, the account number, the customer's name and the customer's email address e.g.,

account_number	ACC123
name	Gerry Byrne
email	gerry.byrne@example.com

The plan for developing the application, which is itself a microservice, and the other two microservices is:

- Use Test-Driven Development, based on user stories and Gherkins, to create an insurance quote section of the application.
- Build the insurance API to perform CRUD operations.
- Test the CRUD functionality.
- Build the microservice that supplies insurance product descriptions.
- Test this microservice.
- Build the microservice that supplies customer details.
- Test this microservice.
- Build and test endpoints that use one or more microservices.

Source code for this book is available to readers on GitHub (https://github.com/gerardbyrne/ Java-Programming-Spring-InsuranceQuoteApplication.git)

1 Agile User Stories

Agile development prioritizes delivering value to end-users, promoting teamwork, and adapting to change. A fundamental aspect of Agile methodology is the **user story**, which places users at the heart of software development. In the modern software development world, it is essential that developers understand Agile practices including Test-Driven Development and Behavior-Driven Development (BDD). To develop user-centric software applications, developers must fully comprehend and master user stories. We need to be able to read and analyze user stories so we can create code to satisfy the user stories.

What is a User Story?

A **user story** is a brief, informal description of a software feature which is written from the perspective of the end user. It defines three things:

- **Who** the user is we say this is the **persona**.
- **What** they want to achieve we say this is the **goal**.
- **Why** they want to achieve it we say this is the **value**.

A typical format for our user story is:

"As a [persona], I want to [action/goal], so that [value/benefit]."

2 Quick Spring Boot Setup

Creating a Simple Spring Boot Project

1. Open a browser window.
2. In the address bar type: https://start.spring.io/

This is **Spring Initializr**, which is one way to create a starter project for a Spring Boot API as shown in Figure 2-1.

3. Click the **Maven** radio button in the Project section.
4. Click the **Java** radio button in the Language section.
5. Click a version in the Spring Boot section, e.g., 3.5.6.
6. Leave the Project Metadata with the defaults.
7. Click on the **Jar** radio button in the Packaging section.
8. Click on a version in the Java section, e.g., 21.

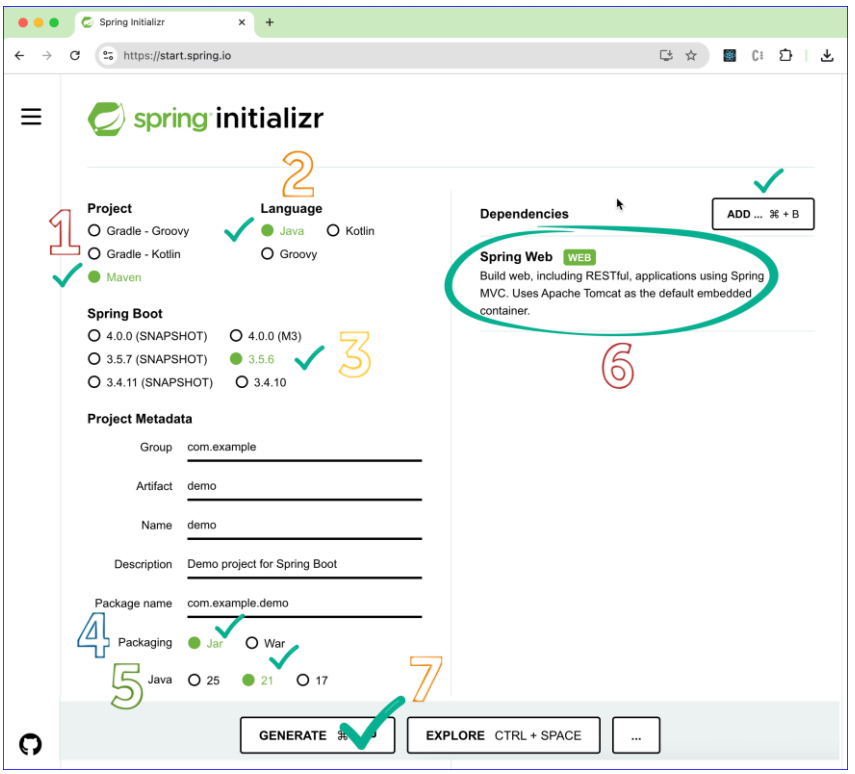


Figure 2-1. Spring Initializr – set up a project

9. Click on the **Add button** in the Dependencies section.

10. Type **Web** in the search box that appears.
11. Click on the **Spring Web** option that appears as shown in Figure 2-2.

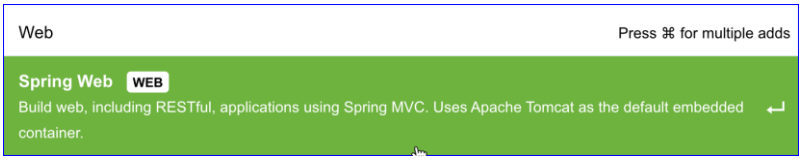


Figure 2-2. Add the Spring Web dependency

Creating a Basic Controller

1. Right click on the **demo** folder (com.example.demo).
2. Choose **New**.
3. Choose **Package**.
4. Name the package **controller** (com.example.demo.controller) as shown in Figure 2-8.

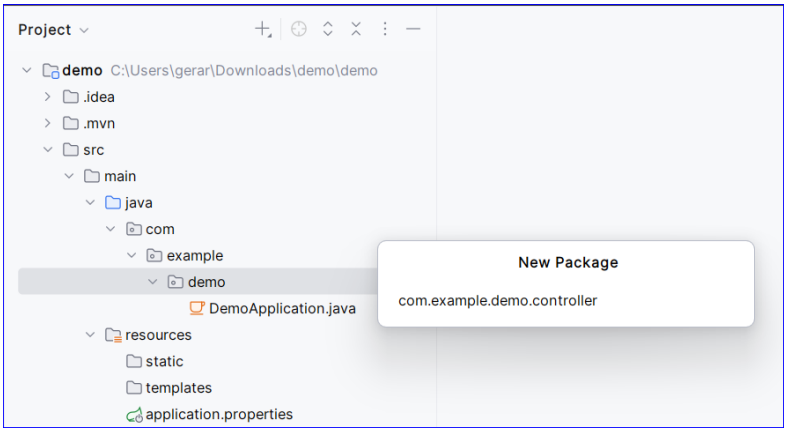


Figure 2-8. Add the controller package

5. Right click on the **controller** package.
6. Choose **New**.
7. Choose **Java Class**.
8. Name the class **HealthCheckController**.
9. Amend the code as shown in Listing 2-2.

Listing 2-1. Controller to check the API works correctly

```
package com.example.demo.controller;

import org.springframework.web.bind.annotation.GetMapping;
```

```java
import org.springframework.web.bind.annotation.RestController;

// RestController is used to create RESTful web services using Spring MVC.
@RestController
public class HealthCheckController {

    // GetMapping is used to map HTTP GET requests onto specific handler methods.
    @GetMapping("/healthcheck")
    public String healthCheck() {
        return "Spring Boot Insurance Quote API!";
    } // End of healthCheck() method

} // End of HealthCheckController class
```

As we add the annotations, @RestController and @GetMapping we may need to import them by hovering over the annotation and choosing Import Class.

Analysis of the HealthCheckController code

What is HealthCheckController.java?

- This is a simple Java class that acts as a REST API controller in a Spring Boot application.
- It provides a basic endpoint and method allowing us to check if the application is running and responding.

Creating a Basic Service

1. Right click on the **demo** folder (com.example.demo).
2. Choose **New**.
3. Choose **Package**.
4. Name the package **service** (com.example.demo.service).
5. Right click on the **service** package.
6. Choose **New**.
7. Choose **Java Class**.
8. Name the class **QuoteCalculator**.
9. Amend the code as shown in Listing 2-6.

Listing 2-2. Service layer with method to calculate the quote value

```java
package com.example.demo.service;

public class QuoteCalculator
{
```

```
    // Method to calculate insurance quote based on product type and value
    public double calculateQuote(String productType, double productValue)
```

Creating a Basic DTO

1. Right click on the **demo** folder (com.example.demo).
2. Choose **New**.
3. Choose **Package**.
4. Name the package **dto** (com.example.demo.dto).
5. Right click on the **dto** package.
6. Choose **New**.
7. Choose **Java Class**.
8. Name the class **QuoteResponseDto**.
9. Amend the code as shown in Listing 2-8.

Listing 2-3. DTO class to be used for a response

```
package com.example.demo.dto;

// Data Transfer Object (DTO) to encapsulate quote response details
public class QuoteResponseDto
{
    // Fields to hold product type, product value, and calculated quote amount
    private String productType;
    private double productValue;
    private double quoteAmount;

    public QuoteResponseDto(String productType, double productValue, double
quoteAmount) {
        this.productType = productType;
        this.productValue = productValue;
        this.quoteAmount = quoteAmount;
    } // End of parameter constructor
```

What is a DTO?

A DTO, or Data Transfer Object, is a simple Java class used to transfer data between different layers of our application. DTOs are designed to carry only the data needed for a specific operation, without any business logic. We will see more about DTOs later when we build the main application and microservices.

Summarizing our basic API

In building our simple API application, we started by organizing our project structure using packages, which helps keep related code grouped together and easy to manage. We created a dedicated controller package and added a HealthCheckController class. This class uses Spring Boot annotations to define a simple REST endpoint. By adding the @RestController annotation, we told Spring Boot that this class would handle web requests and return data. The @GetMapping("/healthcheck") annotation mapped HTTP GET requests to a specific method, allowing us to easily check if our API is running, by visiting a simple URL in the browser.

3 Test-Driven Development

We are creating an application that will, like all applications, use methods, and it is essential that these methods work correctly. One of the essential development tools in software development is Test-Driven Development (TDD), which, as we read earlier, is an Agile software development approach that emphasizes writing tests before any functional code. In modern development, software developers are also testers. There should be no silos, with testers in one group and developers in another. Agile teams have testers and developers working together. Developers should be placing validation at the heart of their processes. Our methods need to be fully tested. The Test-Driven Development process involves the following steps.

Listing 3-1. Sample test class

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class QuoteCalculatorTest
{
    // Declare an object of QuoteCalculator class
    private QuoteCalculator quoteCalculator;

    @BeforeEach
    void setUp() {
        quoteCalculator = new QuoteCalculator();
  } // End of setUp() method

    @Test
    void testQuoteAmount() {
        assertEquals(120, quoteCalculator.quote("Mobile", 1000));
  } // End of testQuoteAmount() method

    @Test
```

```
    void testDiscount() {
        assertEquals(1, quoteCalculator.discount(3, 2));
    } // End of testDiscount() method

} // End of QuoteCalculatorTest class
```

Creating a Maven Project

Start a new project as shown in Figure 3-1**.**

1. Open **IntelliJ IDEA**.
2. Choose **New Project** from within IntelliJ.

Select the project type

3. In the left-hand panel choose **Java**.

Configure Project Name

4. Name the project **InsuranceQuoteBackEnd**.

Choose the Project Location

5. Choose a location for the project.

Set the Build System

6. Choose **Maven** as the build system.

Choose the Project SDK

7. Select the **JDK** (Project SDK) – e.g., **temurin-22** but a higher version is OK as well.

Advanced Settings

8. In the Advanced Settings add the **GroupId** e.g., **org.example**.
9. Amend the **ArtifactId** if required.

Create the project

10. Click the **Create** button.

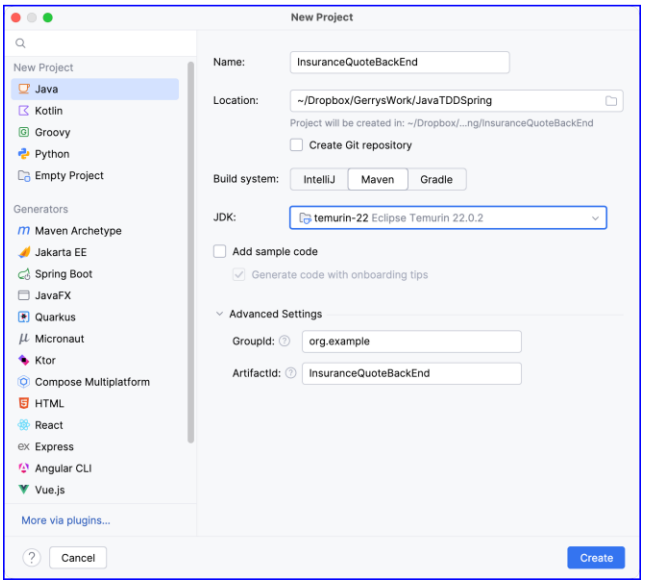


Figure 3-1. Project setup

11. Amend the code as shown in Listing 3-2.

Listing 3-2. Adding the dependencies and dependency management

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>InsuranceQuoteBackend</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>24</maven.compiler.source>
        <maven.compiler.target>24</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <!--
    The <dependencyManagement> section in a pom.xml file is used
    to manage dependency versions in a centralized way.
    This ensures that all modules in a multi-module project use
    the same version of a dependency.

    This dependency management section:
    - Defines a BOM (Bill of Materials):
        The junit-bom is for JUnit 5 and specifies the versions
        of all JUnit artifacts.
    - Sets the version:
        Version 5.9.3 is specified for the junit-bom.
    - Sets Type and Scope:
```

```
    The type is set to pom and the scope to import,
    which tells Maven to import this BOM into the project.

By doing this, all JUnit dependencies in the project will use
version 5.9.3 without needing to specify the version for each
JUnit dependency individually.
-->

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.junit</groupId>
            <artifactId>junit-bom</artifactId>
            <version>5.9.3</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <!--
    This dependency set adds support for JUnit 5, the latest
    version of the JUnit testing framework.

    Jupiter is the new programming and extension model for
    writing tests in JUnit 5.

    - org.junit.jupiter:junit-jupiter
      Convenience dependency that includes both the API and
      the engine for running tests.
    - org.junit.jupiter:junit-jupiter-api
      (included automatically by junit-jupiter) - API used
      to write test code.
    - org.junit.jupiter:junit-jupiter-engine
      (included automatically by junit-jupiter) - Engine that
      runs tests written with the JUnit Jupiter API.
    -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <scope>test</scope>
    </dependency>

    <!--
    To use the Suite features of JUnit 5, add these dependencies:
    - junit-platform-suite-api
      Provides the API for creating test suites.
    - junit-platform-suite-engine
      Executes test suites created with the Suite API.
    -->
    <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-suite-api</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-suite-engine</artifactId>
        <scope>test</scope>
```

```
        </dependency>
    </dependencies>
</project>
```

ALL fourteen tests pass as shown in Figure 3-38.

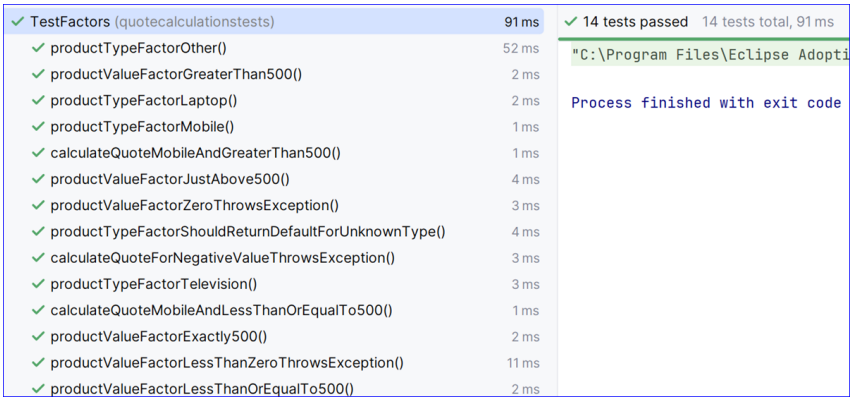


Figure 3-38. All fourteen tests pass

Understanding the manifest and jar files

Currently we have a small application that can create an insurance quote for an electrical item, given the product type and the product value. We have used Test-Driven Development to create the application, and our tests were developed to verify that the user stories were fully complied with. We ran the application from the main() method within the IntelliJ Integrated Development Environment, and all worked well.

4 Spring Boot API – Insurance Quote Backend

In a tiered system, we use a separation of concerns approach and organize our codebase into distinct parts, each with a specific responsibility. This is what we have just completed with our tests and business logic. By using separation of concerns, we make the application easier to manage, test, and maintain.

In a Spring Boot API application, separation of concerns means dividing the codebase into layers or components, each handling a specific responsibility. This approach improves maintainability, testability, scalability, and clarity by ensuring that each part of the application has a clear, focused role.

Explanation of common layers

1. Model Layer

   The model layer contains the domain objects or **entities** that standard Spring Boot applications use to represent the **data** in the application. These classes are typically annotated with JPA annotations to map them to database tables.

2. Repository Layer

   The repository layer contains **interfaces** that define the **CRUD** operations for the domain objects. These interfaces extend the JpaRepository interface provided by Spring Data JPA.

3. Service Layer

   The service layer contains the **business logic** of the application. It interacts with the repository layer to perform CRUD operations on the domain objects.

4. Controller Layer

   The controller layer contains the **REST endpoints** that handle incoming HTTP requests. It interacts with the service layer to process the requests and return the appropriate responses.

5. Main Application Class

   The main application class is the entry point of the Spring Boot application. It contains the main method that starts the Spring Boot application.

6. Application Properties

   The application.properties file contains configuration settings for the Spring Boot application, such as database connection details, server port, and logging settings.

7. Test Classes

   The test classes contain unit tests for the various components of the application, such as the models, repositories, services, and controllers.

8. Exception Handling

   The exception handling classes contain custom exception classes that extend the RuntimeException class to handle errors and exceptions in the application.

9. Security Configuration

   The security configuration classes contain configurations for securing the application, such as authentication, authorization, and access control.

10. Logging Configuration

    The logging configuration classes contain configurations for logging in the application, such as log levels, log file location, and log format.

11. Swagger Configuration

    The Swagger configuration classes contain configurations for generating API documentation using Swagger, such as API version, title, description, and contact information.

12. Docker Configuration

    The Docker configuration files contain configurations for building and running the application in a Docker container, such as Dockerfile, docker-compose.yml, and docker-compose.override.yml.

13. CI/CD Configuration

    The CI/CD configuration files contain configurations for continuous integration and continuous deployment, such as Jenkins file, GitHub Actions workflow, and GitLab CI/CD pipeline.

We can now begin our journey by building our Spring Boot application and making the API. The Maven project we have with the test and business logic classes can now be extended to create a standard Spring Boot application. We will not use all the configurations we have just read about but, we will use layers and configuration to create a fully working and extendible application.

We will create an insurance quote application that enables users to manage insured items, calculate insurance quotes, and view related customer and product information. The main application will provide CRUD operations for insured items, allow users to request insurance quotes based on product type and value, and support advanced search, pagination, and sorting features. Along with this main insurance quote application we will develop two microservices. The first microservice, The Customer Microservice, will be a service that manages customer data, such as account numbers, names, and email addresses. The main application will call this microservice to retrieve and display customer information associated with insured items.

The second microservice, The Product Microservice, will be a service that manages product details, including product types and descriptions. The main application will interact with this microservice to fetch and display product information relevant to each insured item.

Together, our three segregated applications will form a distributed system where the main application orchestrates insurance operations and integrates customer and product data from dedicated microservices, ensuring modularity, scalability, and clear separation of concerns.

In line with what we did in building the quote classes we will look at some user stories and Gherkins that could act as the starting point for our full application. Whilst the user stories are not a complete list, they are the core for what the application is required to achieve. We are simply trying to ensure that we understand that systems are built from a user-centric perspective and that as developers we build our application based on the user stories and acceptance criteria we are given by the Product Owner. As the focus of this book and our learning is how to build a Spring Boot applications, we will concentrate more on designing the application from the perspective of the architecture layers, and in doing so, we will cover the user stories and acceptance criteria.

Project setup

We will now set up the layers we require to get started with our application. We should think back to the demo API we built with the help of Spring Initializr, as it introduced us to a basic structure for an API project. We saw that we segregated our API using the packages, controller, model, service and dto. We also discussed a little about the use of DTOs. We will enhance our knowledge of DTOs and the structure of an API as we build the remaining part of this application and the two microservices.

1. Right click on the **java** package in the main package.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **api**.

5. Right click on the **api** package.
6. Choose **New**.
7. Choose **Package**.
8. Name the package **controller**.

9. Right click on the **api** package.
10. Choose **New**.
11. Choose **Package**.
12. Name the package **exceptions**.

13. Right click on the **api** package.
14. Choose **New**.
15. Choose **Package**.
16. Name the package **model**.

17. Right click on the **api** package.
18. Choose **New**.
19. Choose **Package**.
20. Name the package **repository**.

21. Right click on the **api** package.
22. Choose **New**.
23. Choose **Package**.
24. Name the package **service**.

The new project structure is shown in Figure 4-1.

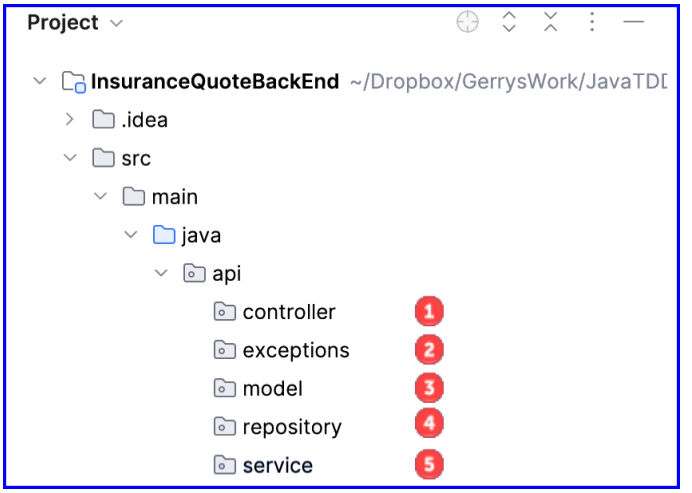


Figure 4-1. API layers

The Model Class

The model class named InsuredItem will represent an entity in our application. The model will consist of fields, constructors, getters and setters and a toString() method, just like any other Java class can have. The purpose of this model is to define the structure for the data that will be stored in the database and to facilitate interaction with the database through an object-oriented approach. This model will be mapped to a database table using the Java Persistence API (JPA) annotations. The annotations we will use are:

The Repository Class

The repository is an interface, which often extends JpaRepository, and provides Create, Read, Update, Delete (CRUD) and query methods backed by JPA. The repository is the persistence boundary, and our services will use repositories to load, save, and delete entities. Our controllers should not call repositories directly, they should use services to do this.

Listing 4-1. Add the code for the repository layer

```
package api.repository;

/*
The JpaRepository interface provides various methods for performing
CRUD (Create, Read, Update, Delete) operations and pagination on the
InsuredItem entity.
*/
import api.model.InsuredItem;
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.List;

/*
The InsuredItemRepository interface extends the JpaRepository interface,
which takes the entity type (InsuredItem) and the type of the primary
key (Long) as type arguments. We use this JpaRepository interface to
interact with the InsuredItem entity in the database. The interactions we
can use include the CRUD operations, sorting, and pagination.
The JpaRepository interface extends the PagingAndSortingRepository
interface, which in turn extends the CrudRepository interface.
*/

public interface InsuredItemRepository extends JpaRepository<InsuredItem, Long>
{
    /* We use List to return a list of InsuredItem objects. */
    List<InsuredItem> findByCustomerAccountNumber(String customerAccountNumber);

} // End of InsuredItemRepository interface
```

The Service Class

Now we will create the service layer for our API. The responsibility of the services will be to encapsulate the business logic of the application. It will be the **intermediary between** the controller layer, which will handle the **HTTP requests** and **responses**, **and the data access layer**, which interacts with the database. The service layer ensures that business rules are applied, and it provides a clean separation of concerns. Our service layer called, **InsuredItemService**, will handle operations such as creating, retrieving, updating, and deleting related to the InsuredItem model. We will see later how the methods in this layer will be called by the methods in the Controller layer.

25. Amend the code as shown in Listing 4-13.

Listing 4-2. Add the code for the Service layer

```
package api.service;

import api.dto.CustomerWithInsuredItemsDTO;
import api.dto.InsuredItemWithoutAccountDTO;
import api.dto.CustomerDTO;
import api.model.InsuredItem;
import api.repository.InsuredItemRepository;
import api.service.quotecalculations.CalculateQuote;
import org.springframework.stereotype.Service;
import api.exceptions.InvalidQuoteException;
import api.exceptions.QuoteNotFoundException;

import java.util.List;

/*
@Service annotation is used to mark the class as a service
provider. The @Service annotation is a specialization of the
@Component annotation. It's a good practice to use @Service over
@Component in service-layer classes
*/
@Service
public class InsuredItemService {

    /*************************************************************
                        DEPENDENCY INJECTION
            Inject CalculateQuote, InsuredItemRepository, and
            ExternalServiceClient for business logic, data access,
            and external service integration
     *************************************************************/
    private final CalculateQuote calculateQuote;
    private final InsuredItemRepository insuredItemRepository;
    private final ExternalServiceClient externalServiceClient;

    /*************************************************************
                  CONSTRUCTOR-BASED DEPENDENCY INJECTION
            Inject CalculateQuote, InsuredItemRepository,
            and ExternalServiceClient into InsuredItemService
```

```
     ************************************************************/
    public InsuredItemService(CalculateQuote calculateQuote,
                              InsuredItemRepository insuredItemRepository,
                              ExternalServiceClient externalServiceClient) {
        this.calculateQuote = calculateQuote;
        this.insuredItemRepository = insuredItemRepository;
        this.externalServiceClient = externalServiceClient;
    } // End of InsuredItemService() constructor

} // End of class InsuredItemService
```

Adding database functionality

To persist and manage data in our API, we will integrate a database. Using a database allows the application to store, retrieve, update, and delete records efficiently. For development and testing purposes, we will use the H2 database, which is an in-memory, lightweight, and fast relational database. H2 requires minimal configuration and runs within the application, making it ideal for prototyping and local development. It is not a database we would use at the production stage. We will add a dependency for the h2 database in the pom.xml.

1. Open the **pom.xml** file.
2. Amend the pom.xml code as shown in Listing 4-39.

Listing 4-3. Add the h2 dependency

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>2.3.232</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Having amended the pom.xml file we need to make sure all dependencies are up to date.

SQL Records

We will now create a **data.sql** file, which will be used to insert initial data into the INSUREDITEM table of our H2 in-memory database. This table will store details about various items, including their product type and value. By populating the database with this data, we will be able to test our application with a predefined set of records. We will use SQL INSERT statements to add multiple rows to the INSUREDITEM table. Each row will

represent an electrical item with product type and product value attributes. This approach ensures that our database is preloaded with data when the application starts, allowing us to focus on developing and testing the application's functionality without manually entering data each time.

3. Right click on the **resources** package.
4. Choose **New**.
5. Choose **File**.
6. Name the file **data.sql**.
7. Press the **Enter** key.
8. Amend the code as shown in Listing 4-41 to add records that will populate the H2 database.

Listing 4-4. Records to be inserted into the H2 database

```
INSERT INTO insureditem(product_type, product_value, quote_amount,
customer_account_number) VALUES ('Mobile Phone', 1200.00, 100.00, 'ACC123');
INSERT INTO insureditem(product_type, product_value, quote_amount,
```

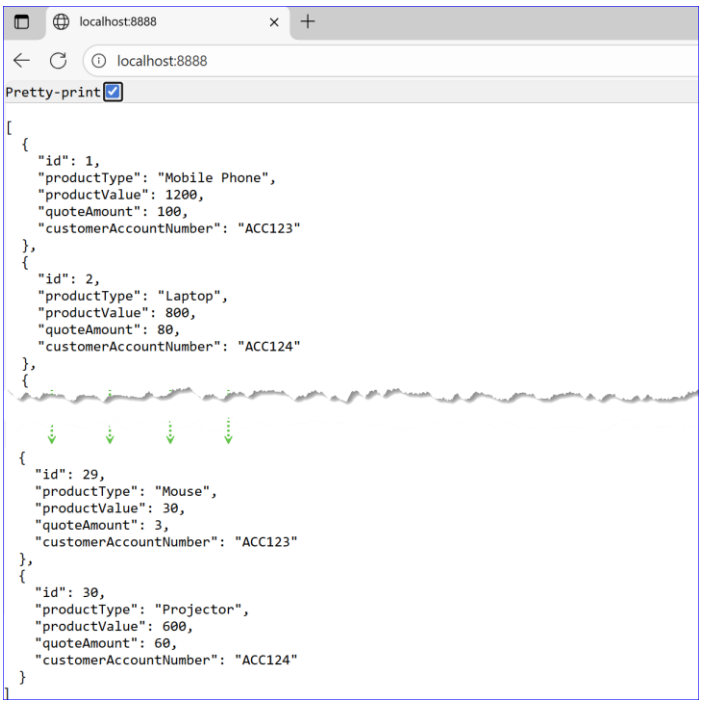


Figure 4-4. All records are displayed

5 Product Description Microservice

A **microservice** is a small, independently deployable application that focuses on a specific business capability. In this microservice (application), we are now going to build functionality for managing information about a Product. In the context of **Spring Boot**, a microservice is typically built as a standalone Spring Boot application that exposes its functionality through RESTful APIs. It uses **Spring Data JPA** to interact with its own dedicated database, mapping Java entities to database tables and handling data persistence. The InsuranceQuoteBackEnd application we just created could considered a microservice. A microservice manages its own data and logic, communicates with other microservices through HTTP APIs and can be developed, deployed, and scaled independently of other services.

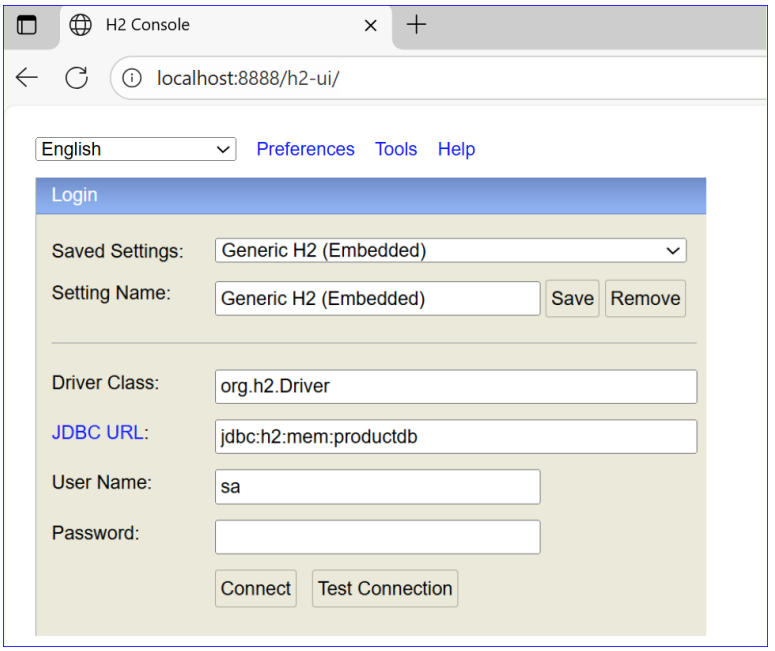


Figure 5-3. Connection to the H2 database

1. Click on the **PRODUCT** table on the left-hand side.
2. Click on the **Run** button.

The list of products in the table is displayed and shown in Figure 5-4. The column order might be different, but this is not important.

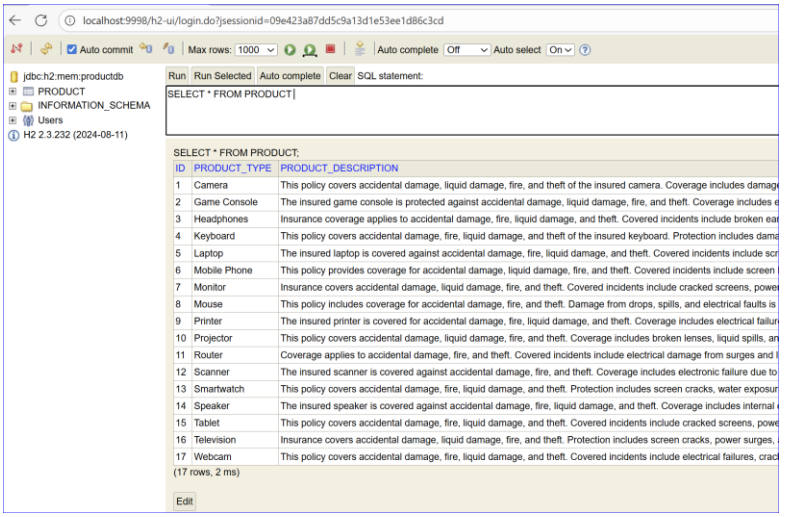


Figure 5-4. Run the SQL Select all command

6 Customer Microservice

As we have read previously, a **microservice** is a small, independently deployable application that focuses on a specific business capability. In our application we have created the Product microservice and now we will create a second microservice to hold data about Customers. The Customer microservice will manage its own data and logic and communicate with other microservices through HTTP APIs.

Figure 6-3. Connection to the H2 database

1. Click on the **CUSTOMER** table on the left-hand side.
2. Click on the **Run** button.

The list of customers in the table will be displayed as shown in Figure 6-4. The column order might be different, but this is not important.

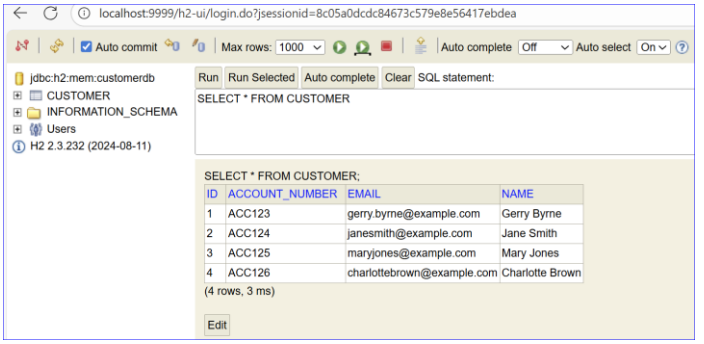


Figure 6-4. Run the SQL Select all command

7 Testing the microservices integration

All customers and their insured items

We will now create code in the InsuranceQuoteBackEnd application that will allow us to see all customers, along with their details and the list of items they have insured. We therefore need to integrate data from two sources. The insured items are stored in the H2 database managed by our InsuranceQuoteBackEnd application. However, customer details are not stored locally, instead, they reside in a separate microservice running on port 9999.

The full set of insured items, each with the customer details, and the product details should be displayed as shown in Figure 7-2.

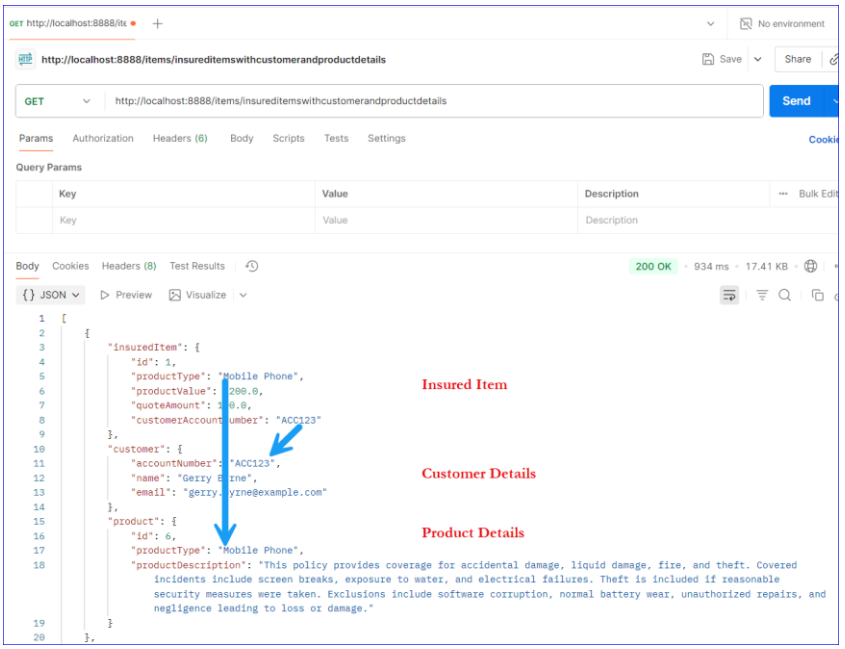


Figure 7-2. Insured items with customer and product details

8 Pagination and dynamic searching

Rather than having a long list of records displayed in the browser or end user's application, we can use **pagination**, which is available with Spring Data JPA. Pagination is a technique we can use to split large sets of data into smaller chunks or pages. Pagination can improve performance and enhance the user experience by allowing clients to request only a subset of the data at a time.

The sixth page with a set of five insured items should be displayed as shown in Figure 8-8, remembering that we asked for descending order by product type, then quote amount by ascending order and customer account number in ascending order.

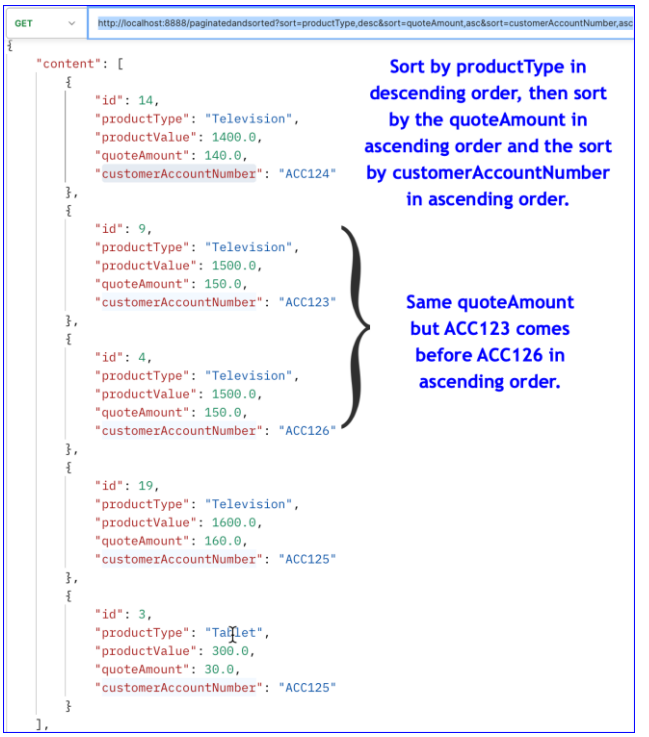
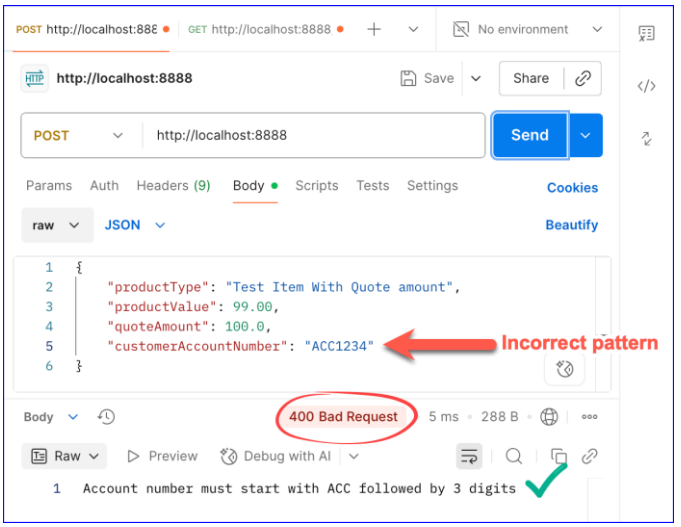


Figure 8-8. Pagination with sort by multiple fields

9 Validation

Validation is not just checking fields, it is the first and most important point where we check for correctness in our Spring Boot API or microservice. There are many reasons why we need to validate in our Spring API application. One simple example would be to follow the important principle of **fail fast**. This principle can help us save effort by rejecting incorrect requests at the controller level, which is the edge of our application receiving the user requests. As an example, there would no point in calling the service layer if the minimum or maximum value in the request is violated.



Listing 9-1. Customer account number pattern incorrect

10 Mocking

Introduction to mocking

We unit tested our code to ensure that the methods in the quote calculator worked, and we got a correct quote value when we passed the product details to a method. When we test our code, the goal is to check whether a particular class or method works as expected. However, as we have seen in building the insurance application, classes usually do not work in isolation. Our insurance quote service depends on two other microservices, one to fetch customer details and the other to retrieve product details. When we run a unit test that calls these services our test will be dependent on systems outside of our control. We may find a test fails because the product service is down, or the customer database is being updated. So, our tests would fail not because our code is incorrect, but because of external factors.

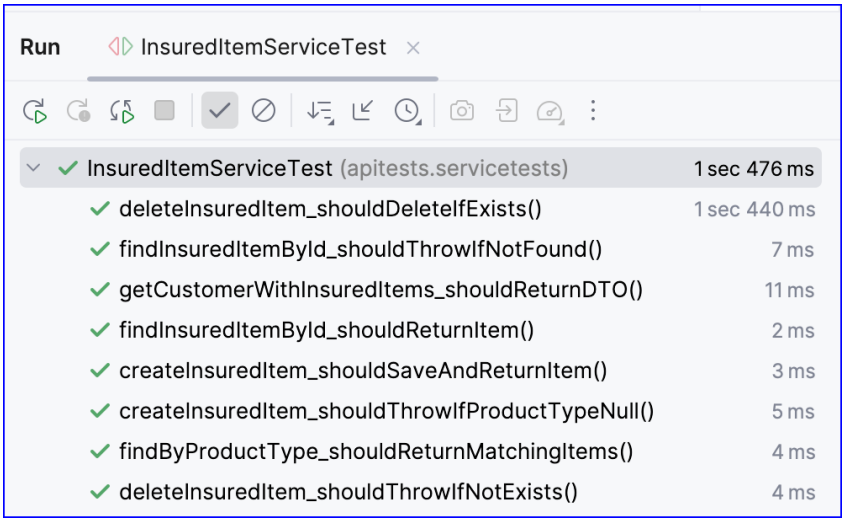


Figure 10-1. All Mockito tests pass

These tests cover creation, validation, retrieval, deletion, and a DTO mapping scenario. Mocking is an important feature of modern software development and in this chapter we have simply introduced ourselves to its use.

11 Logging

Introduction to logging with SLF4J

We can use logging to record information about the execution of our application. As developers we can use logging to track the flow of our program, monitor its behavior, and capture important events or errors that occur during runtime. Logging is essential for debugging, troubleshooting, and maintaining any application, since it provides a history, a record, of what happened in the system when it runs. This is an invaluable source of information when diagnosing issues or understanding user actions.

We will now see the logging information in the console window as shown in Figure 11-8.

Figure 11-8. Logging information for the findProductsByType in external service

This endpoint is missing the last parameter, it should be http://localhost:8888/quote?productType=Laptop&productValue=400 which we used previously.

12 Swagger

Introduction to Swagger

What is Swagger?

Swagger is an open-source framework that allows us to design, build, document, and consume RESTful web services. It provides a standard, language-agnostic interface to REST APIs that allows both humans and computers to discover and understand the capabilities of a Restful service without requiring access to source code or additional documentation. Swagger uses the OpenAPI Specification (OAS), which defines how to describe REST APIs in a machine-readable format. When we access the Swagger UI in a browser, we will see an interactive documentation page that lists all our API endpoints, their parameters, request and response formats, and allows us to test them directly from the interface. We also see documentation for our DTO objects.

The Swagger interface should appear as shown in Figure 12-1.

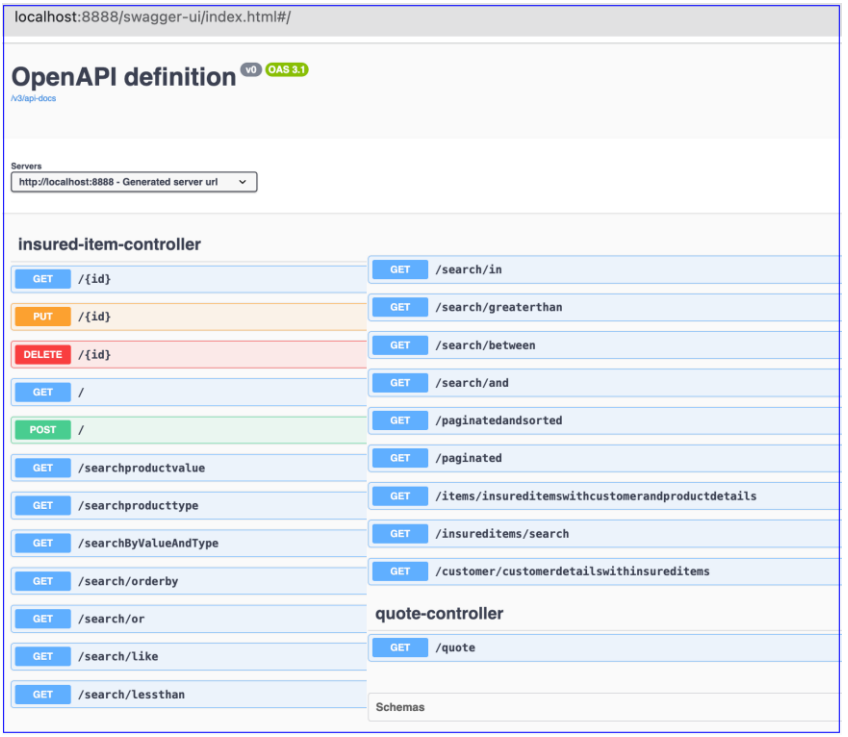


Figure 12-1. Swagger UI Interface

13 RestTemplate and WebClient

In the three microservice we have created and tested, we have the **InsuranceQuoteBackEnd** microservice (application) which can call both external microservices, **CustomerMicroservice and ProductMicroservice** using the RestTemplate.

RestTemplate - A Classic Approach

RestTemplate has been the standard way to make HTTP requests in Spring applications for many years. It is simple, synchronous, and easy to use for straightforward REST calls. For example, in Listing 13-1 we see the code we used to connect to an external customer microservice.

Listing 13-1. Using the RestTemplate getForObject() method

```
public CustomerDTO findCustomerByAccountNumber(String accountNumber)
{
    logger.info("Fetching customer details for account number: {}", accountNumber);
    String url = "http://localhost:9999/api/customer/account/" + accountNumber;
    CustomerDTO customer = restTemplate.getForObject(url, CustomerDTO.class);
    logger.info("Received customer details: {}", customer);
    return customer;
} // End of findCustomerByAccountNumber method
```

This approach works well for traditional, blocking applications.

WebClient - A Modern, Reactive Solution

As applications in the enterprise have become more demanding, requiring better scalability, non-blocking I/O, and support for reactive programming, **WebClient** was introduced as part of Spring WebFlux. **WebClient is fully non-blocking and supports both synchronous and asynchronous operations,** making it ideal for modern, high-concurrency applications.

14 Glossary of Terms

Spring API terms

API
: An Application Programming Interface is a set of defined rules, protocols, and tools that allow different software systems to communicate with each other. When dealing with web APIs, we usually refer to HTTP-based endpoints that handle requests and responses between the client and the server.

WebClient terms

Reactive Programming
: An asynchronous programming approach for handling data streams and event propagation. WebClient leverages this model to handle multiple concurrent requests efficiently without blocking threads.