Chapter 2: Getting Started with MicroProfile

Introduction

In this chapter, you'll embark on your MicroProfile journey! We will guide you through creating your first microservice, equipping you with the essential understanding to leverage this robust framework for building modern, cloud-native applications. This journey begins with setting up your development environment, then diving into creating a microservice.

Topics Covered

- Development Environment Setup
- Configuring Build Tools
- Creating a Java Project for MicroProfile
- Choosing Right Modules for your MicroProfile application
- Building a RESTful Web Service
- Deploying your MicroProfile Application
- Testing your Microservices
- Exploring Further with MicroProfile

Development Environment Setup

Let's begin by preparing your workspace for MicroProfile development:

Java Development Kit (JDK)

MicroProfile, a Java framework, runs on the Java Virtual Machine (JVM), making the Java Development Kit (JDK) an essential component of your development environment.

To install JDK, follow the steps below:

- **Download:** Visit the official OpenJDK website and download the JDK version compatible with your operating system.
- **Install:** Follow the installation instructions provided on this official OpenJDK Installation guide.
- **Verify:** After Installation, run the following command in your command line or terminal to verify the Installation:

```
java -version
```

You should an output similar to the one below:

```
openjdk 17.0.10 2024-01-16 LTS
OpenJDK Runtime Environment Microsoft-8902769 (build 17.0.10+7-LTS)
OpenJDK 64-Bit Server VM Microsoft-8902769 (build 17.0.10+7-LTS, mixed mode,
sharing)
```

This confirms that JDK 17 has been successfully installed on your system.

Note	For most MicroProfile implementations, JDK 11 or later is recommended. In this tutorial, we will be using JDK 17. While OpenJDK is used here, other JDK providers such as Oracle JDK, Amazon Corretto, and Azul Zulu also offer compatible JDK distributions.

Build Tools (Maven or Gradle)

Build tools like Apache Maven or Gradle are commonly used for managing project dependencies and building Java applications. You can install the one that best fits your project needs. Here's a brief overview to help you decide:

- **Apache Maven:** Known for its convention-over-configuration approach, Maven is a popular choice due to its simple project setup and extensive plugin repository.
- **Gradle:** Offers a flexible, script-based build configuration, allowing for highly customized build processes. Gradle is renowned for its superior performance, due to its incremental builds and caching mechanisms. It's a great choice for complex projects requiring customization.

Important	If your existing project's build uses Maven wrapper (`mvnw`) or Gradle wrapper (`gradlew`), you don't have to install any of these build tools. These wrappers help ensure a consistent build environment without requiring the build tools to be installed on your system.

Installing Apache Maven

To install Maven follow the steps below:

1. Visit the Installing Apache Maven web page to download the latest version.
2. Follow the installation instructions provided on the site.
3. Verify the Maven installation by running this command in your terminal or command line.

```
mvn -v
```

You should see output similar to:

```
Apache Maven 3.9.6 (bc0240f3c744dd6b6ec2920b3cd08dcc295161ae)
Maven home: /usr/local/sdkman/candidates/maven/current
Java version: 17.0.10, vendor: Microsoft, runtime:
/usr/lib/jvm/msopenjdk-current
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "6.2.0-1019-azure", arch: "amd64", family: "unix"
```

After Maven is installed, you can configure the *pom.xml* file in your project to include the MicroProfile dependencies.

Installing Gradle

To install Gradle follow the step below:

1. Visit the Gradle | Installation web page to download the latest version.
2. Follow the installation instructions provided on the site.
3. Verify the installation by running this command in your terminal or command line.

```
gradle -version
```

You should see output similar to:

```
Welcome to Gradle 8.6!

Here are the highlights of this release:
 - Configurable encryption key for configuration cache
 - Build init improvements
 - Build authoring improvements

For more details see https://docs.gradle.org/8.6/release-notes.html

------------------------------------------------------------
Gradle 8.6
------------------------------------------------------------

Build time:   2024-02-02 16:47:16 UTC
Revision:     d55c486870a0dc6f6278f53d21381396d0741c6e

Kotlin:       1.9.20
Groovy:       3.0.17
Ant:          Apache Ant(TM) version 1.10.13 compiled on January 4 2023
JVM:          17.0.10 (Microsoft 17.0.10+7-LTS)
OS:           Linux 6.2.0-1019-azure amd64
```

After Gradle is installed, you can configure the *build.gradle* file in your project to include the MicroProfile dependencies.

Whether you opt for Maven's stability and convention or Gradle's flexibility and performance, understanding how to configure and use your chosen build tool is important for MicroProfile development.

Integrated Development Environments

Integrated Development Environments (IDEs) enhance developer productivity by providing a rich set of features and extensions such as project bootstrapping, dependency management, intelligent code completion, configuration assistance, test runners, build, hot deployment and debugging tools. For MicroProfile development, the choice of IDE can significantly affect your development speed and efficiency. Below is a list of popular IDEs and their key features related to Java and MicroProfile development:

Eclipse for Enterprise Java and Web Developers

Overview: Eclipse is a widely used IDE for Java development, offering extensive support for Java EE, Jakarta EE, and MicroProfile, among other technologies.

Getting Started: The official Eclipse documentation containing instructions about creating Java projects - Creating your first Java Project

IntelliJ IDEA

Overview: IntelliJ IDEA by JetBrains supports a wide range of programming languages and frameworks, including Java, Kotlin, and frameworks like Spring, Jakarta EE, and MicroProfile.

Getting Started: Refer to this IntelliJ IDEA guide on Creating a Java Project Using IntelliJ IDEA 2024.1.

Apache NetBeans

Overview: NetBeans is an open-source IDE that supports Java development, including Java SE, Java EE, JavaFX, and more.

Getting Started: Check out this NetBeans Java Quick Start Tutorial for a tutorial on creating a Java application.

Visual Studio Code

Overview: Visual Studio Code is a lightweight, powerful source code editor that supports Java development through extensions.

Getting Started: To start with Java in VS Code, follow this Getting Started with Java in VS Code documentation.

Selecting an IDE should be based on personal preference, as the best choice varies depending on individual needs, familiarity, and the specific features that enhance your productivity. Each IDE offers unique advantages for MicroProfile development.

Setting up MicroProfile Runtime

MicroProfile applications require a runtime that supports MicroProfile specifications or a MicroProfile-compatible server to run your applications. Below are some popular options, each with unique features tailored to different needs:

Open Liberty

Open Liberty is a flexible server framework from IBM that supports MicroProfile, allowing developers to build microservices and cloud-native applications with ease.Open Liberty is known for its dynamic updates and lightweight design, which enhances developer productivity and application performance.

Downloading Open Liberty page provides access to its latest releases and documentation to help you set up your environment.

Quarkus

Quarkus is known for its container-first approach, offering fast startup times and low memory footprint. It aims to optimize Java for Kubernetes and cloud environments

This Getting Started with Quarkus page will guide you through creating your first Quartus project and exploring its cloud-native capabilities.

Payara Micro

Payara Micro is a lightweight middleware platform suited for containerized Jakarta EE and MicroProfile applications.

The Payara Platform Community Edition enables easy packaging of applications into a single, runnable JAR file, simplifying deployment and scaling in cloud environments. This site about Payara Platform Community Edition offers downloads and documentation to get started.

WildFly

WildFly is a flexible, lightweight, managed application runtime that offers full Jakarta EE and MicroProfile support. WildFly is designed for scalability and flexibility in both traditional and cloud-native environments.

WildFly Downloads page offers the latest versions and documentation to get you started.

Helidon

Developed by Oracle, Helidon MP implements MicroProfile specifications. It provides a familiar programming model for Jakarta EE developers and enables efficient microservice development.

Helidon Documentation provides comprehensive resources to help developers get started with the framework, understand its core concepts, and develop microservices efficiently.

Apache TomEE

TomEE integrates several Apache projects with Apache Tomcat to provide a Jakarta EE environment. It offers support for MicroProfile, allowing developers to build and deploy microservices using the well-known Jakarta EE technologies with additional MicroProfile capabilities.

TomEE Downloads and TomEE MicroProfile Documentation page provide the necessary resources to get started with TomEE for MicroProfile development.

MicroProfile Starter

To kickstart your MicroProfile project, use the MicroProfile Starter to generate a sample project with your chosen server and specifications. This tool provides a customizable project structure and generates necessary boilerplate code and configuration.

1. Visit the MicroProfile Starter page - the official website for generating the MicroProfile project templates.
2. Provide a `groupId` for your project, it's an identifier for your project and should be unique to avoid conflicts with other libraries or projects.

Tip	Its recommended convention is to start your `groupId` with the reverse domain name of your organization (for example, `io.microprofile`).

3. Enter the artifact ID, which is the name of your project (e.g., product-ws).
4. Select the Java SE version your project will use.
5. Select the MicroProfile version you want to use. Ideally, you should choose the latest version for the most up-to-date features but also consider the runtime's support.
6. Select the specifications you want to include in your project. These could be Config, Fault Tolerance, JWT Auth, Metrics, Health, Open API, Open Tracing, Rest Client. Choose what is relevant to your application.
7. Click the "Download" button.
8. Unzip the generated project and import it into your IDE.

This completes the development environment setup. Now we are all set to begin development using MicroProfile.

Important	At the time of writing this tutorial, the latest MicroProfile released version was 6.1. The MicroProfile Starter does not currently support this version. Hence, we will not be using MicroProfile Starter to generate the project structure.

Creating a Java Project for MicroProfile Development

Using Your IDE:

Most modern IDEs have built-in support for creating Java and Maven projects. Depending on your chosen IDE, look for options like "New Project", or "New Maven Project" to get started. These options typically guide you through the setup process, including specifying the project's `groupId`, `artifactId`, and dependencies.

Using Maven from Command Line (Terminal)

For developers who prefer using the command line or for those setting up projects in environments without an IDE, Maven can generate the base structure of a Java project through its archetype mechanism. To create a project using Maven, open your terminal or command line and run the following command:

```
mvn archetype:generate -DgroupId=io.microprofile.tutorial -DartifactId=store
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Explanation:

- `mvn archetype:generate` goal in this command initializes the project creation process and instructs Maven to generate a new project based on an archetype template.
- `-DgroupId=io.microprofile.tutorial` Specifies the groupId for the project. The groupId is a unique identifier for your project and is usually based on the domain name of your organization in reverse. In this case, io.microprofile.tutorial represents a project related to MicroProfile tutorial.
- `-DartifactId=store`: Sets the `artifactId` for the project. The `artifactId` is the name of the project and is used as the base name for the project's artifacts (e.g., WAR files). Here, `store` is used as the project name to indicate this project is related to an e-commerce store application.
- `-DarchetypeArtifactId=maven-archetype-quickstart`: Indicates which archetype to use for generating the project. The maven-archetype-quickstart is a basic template for a Java application, providing a simple project structure that includes a sample application (App.java) and a unit test (AppTest.java).
- `-DinteractiveMode=false`: This option disables interactive mode, meaning Maven will not prompt you for input during the project generation process. All necessary information is provided via the command-line options, allowing the command to execute without further user interaction.

Next, using your file explorer or command line, create the following folder structure:

```
.
├── pom.xml
├── readme.md
└── src
    └── main
    │   └── java
    │       └── io
    │           └── microprofile
    │               └── tutorial
    │                   └── store
    │                       └── product
    │                       │   ├── entity
    │                       │   │   └── Product.java
    │                       │   └── service
    │                       │       └── ProductService.java
    │                       └── ProductRestApplication.java
    └── test
        └── java
            └── io
                └── microprofile
                    └── tutorial
                        └── store
                            └── product
                                └── ProductServiceTest.java
```

The standard location for your Java source code is this folder:

```
src/main/java
```

And, the standard location for your test code is this folder:

```
src/test/java
```

You can delete `App.java` and `AppTest.java` files from the poject as these are not required in MicroProfile development.

The heart of your Maven project is `pom.xml` (Project Object Model) file. It defines project metadata, dependencies, build configurations and plugins.

The content for the *pom.xml* file should look as below, ensure MicroProfile depencency is added:

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

  <groupId>io.microprofile.tutorial</groupId>
  <artifactId>mp-ecomm</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
```

```
  <!-- Setting the source and target of the Java Compiler !>
  <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputE
ncoding>
     <maven.compiler.source>17</maven.compiler.source>
     <maven.compiler.target>17</maven.compiler.target>
  </properties>
...
...
   <!-- Add Lombok dependency -->
   <dependency>
     <groupId>org.projectlombok</groupId>
     <artifactId>lombok</artifactId>
     <version>1.18.26</version>
     <scope>provided</scope>
   </dependency>


   <!-- Adding Jakarta EE dependencies -->
   <dependency>
     <groupId>jakarta.platform</groupId>
     <artifactId>jakarta.jakartaee-api</artifactId>
     <version>10.0.0</version>
     <scope>provided</scope>
   </dependency>


   <!-- Adding MicroProfie dependency -->
   <dependency>
     <groupId>org.eclipse.microprofile</groupId>
     <artifactId>microprofile</artifactId>
     <version>6.1</version>
     <type>pom</type>
     <scope>provided</scope>
   </dependency>


   <!-- JUnit Jupiter API for writing tests -->
```

```
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.8.2</version>
        <scope>test</scope>
    </dependency>

    <!-- JUnit Jupiter Engine for running tests -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.8.2</version>
        <scope>test</scope>
    </dependency>
...
```

Below is the list of essential dependencies you need to add to your Maven pom.xml for a MicroProfile project:

- Lombok Dependency - Simplifies your model by auto-generating getters, setters, constructors, and other boilerplate code.
- Jakarta EE API Dependency - Provides the APIs for Jakarta EE, which are often used alongside MicroProfile for enterprise Java applications.
- MicroProfile Dependency - This is the core MicroProfile dependency that allows you to use MicroProfile specifications in your project.
- JUnit Jupiter API for Writing Tests - Essential for writing unit tests for your MicroProfile services.
- JUnit Jupiter Engine for Running Tests - Enables the execution of JUnit tests.

These dependencies provide a foundation for building MicroProfile applications, including aspects like model simplification with Lombok, the application of Jakarta EE APIs for building robust enterprise applications, and testing with JUnit. Remember to adjust the versions based on your project requirements and the compatibility with your MicroProfile runtime.

Tip	Execute the `$ mvn validate command`. This checks the *pom.xml* file for correctness, ensuring that all necessary configuration is present and valid.

Choosing Right Modules for Your MicroProfile Application

Choosing the right modules for your MicroProfile application is crucial for ensuring that your application is lean, maintainable, and only includes the necessary functionalities to meet its requirements.

Before diving into MicroProfile modules, it's essential to have a clear understanding of your application's requirements. Consider aspects such as configuration needs, security, health checks, data metrics, fault tolerance, and the need for distributed tracing. Mapping out these requirements will guide you in selecting the most relevant MicroProfile specifications. MicroProfile provides a selection of APIs that you can choose from based on the specific needs of your application. However, with the variety of specifications available, it's important to understand which ones best fit your project's needs.

This section aims to help you make informed decisions about which MicroProfile modules to use.

Use the Entire MicroProfile Dependency

If you're beginning a new MicroProfile-based project and are unsure which specifications you will need, starting with the entire MicroProfile dependency can give you immediate access to the full suite of MicroProfile APIs. This approach allows you to explore and experiment with different specifications without modifying your pom.xml to add or remove dependencies frequently.

For projects that aim to leverage a wide range of MicroProfile specifications, including advanced features like telemetry, metrics, and fault tolerance, including the entire MicroProfile 6.1 dependency ensures that you have all the necessary APIs at your disposal. This approach simplifies dependency management, especially for complex applications.

Maven

```
<!-- MicroProfile 6.1 API -->
<dependency>
    <groupId>org.eclipse.microprofile</groupId>
    <artifactId>microprofile</artifactId>
    <version>6.1</version>
    <type>pom</type>
    <scope>provided</scope>
</dependency>
```

Gradle

```
dependencies {
 compileOnly 'org.eclipse.microprofile:microprofile:6.1'
}
```

Use Individual MicroProfile Specification Dependencies

For applications where size and startup time are critical (e.g., serverless functions, microservices with stringent resource constraints), including only the necessary MicroProfile specifications can help minimize the application's footprint. This selective approach ensures that your application includes only what it needs, potentially reducing memory usage and startup time.

To prevent potential conflicts or security issues associated with unused dependencies, it's prudent to include only the specifications your application directly uses. This practice follows the principle of minimalism in software design, reducing the surface area for bugs and vulnerabilities.

The list below is provided to help you select the appropriate modules for your MicroProfile application:

- MicroProfile Config provides a way to fetch configurations from various sources dynamically. You should use this dependency in your microservices if they require external configuration or need to be run in different environments without requiring repackaging.

Maven

```
<dependency>
  <groupId>org.eclipse.microprofile.config</groupId>
  <artifactId>microprofile-config-api</artifactId>
  <version>3.1</version>
</dependency>
```

Gradle

```
implementation 'org.eclipse.microprofile.config:microprofile-config-api:3.1'
```

- MicroProfile Health allows you to define health endpoints easily. If you're deploying your application in an environment where the service needs to report its health status.

Maven

```
<dependency>
  <groupId>org.eclipse.microprofile.health</groupId>
  <artifactId>microprofile-health-api</artifactId>
  <version>4.0.1</version>
</dependency>
```

Gradle

```
implementation 'org.eclipse.microprofile.health:microprofile-health-api:4.0.1'
```

- MicroProfile Metrics offers a way to generate various metrics from your application, which can be consumed by monitoring tools. You should use this

dependency in your microservices if you need to monitor the performance of your application.

Maven

```
<dependency>
  <groupId>org.eclipse.microprofile.metrics</groupId>
  <artifactId>microprofile-metrics-api</artifactId>
  <version>5.1.0</version>
</dependency>
```

Gradle

```
implementation 'org.eclipse.microprofile.metrics:microprofile-metrics-api:5.1.1'
```

- MicroProfile Fault Tolerance helps applications in implementing patterns like timeout, retry, bulkhead, circuit breaker, and fallback. Applications requiring resilience and reliability, especially those facing network latency or failure in microservices environments, will benefit from it.

Maven

```
<dependency>
  <groupId>org.eclipse.microprofile.fault-tolerance</groupId>
  <artifactId>microprofile-fault-tolerance-api</artifactId>
  <version>4.0.2</version>
</dependency>
```

Gradle

```
implementation
'org.eclipse.microprofile.fault-tolerance:microprofile-fault-tolerance-api:4.0.2'
```

- MicroProfile JWT Authentication provides a method for using JWT tokens for securing your microservices, especially where propagation of identity and authentication information is needed across services.

Maven

```
<dependency>
  <groupId>org.eclipse.microprofile.jwt</groupId>
  <artifactId>microprofile-jwt-auth-api</artifactId>
  <version>2.1</version>
</dependency>
```

Gradle

```
implementation
```

```
'org.eclipse.microprofile.jwt:microprofile-jwt-auth-api:2.1'
```

- MicroProfile OpenAPI offers tools for generating OpenAPI descriptions of your endpoints automatically for documenting your REST APIs.

Maven

```
<dependency>
  <groupId>org.eclipse.microprofile.openapi</groupId>
  <artifactId>microprofile-openapi-api</artifactId>
  <version>3.1.1</version>
</dependency>
```

Gradle

```
implementation 'org.eclipse.microprofile.openapi:microprofile-openapi-api:3.1.1'
```

- MicroProfile Rest Client simplifies calling RESTful services over HTTP for type-safe invocations of HTTP services for type-safe invocations of HTTP services.

Maven

```
<dependency>
  <groupId>org.eclipse.microprofile.rest.client</groupId>
  <artifactId>microprofile-rest-client-api</artifactId>
  <version>3.0</version>
</dependency>
```

Gradle

```
implementation 'org.eclipse.microprofile.rest.client:microprofile-rest-client-api:3.0'
```

- MicroProfile Telemetry integrates OpenTelemetry for distributed tracing For applications that need to trace requests across microservices to diagnose and monitor.

Maven

```
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-api</artifactId>
    <version>1.29.0</version>
</dependency>
```

Gradle

```
implementation 'io.opentelemetry:opentelemetry-api:1.29.0'
```

- Jakarta EE Core Profile dependency provides the API set included in the Jakarta EE 10 Core Profile, which is optimized for developing microservices and cloud-native Java applications with a reduced set of specifications for a lighter runtime footprint.

Maven

```
<dependency>
    <groupId>jakarta.platform</groupId>
    <artifactId>jakarta.jakartaee-core-api</artifactId>
    <version>10.0.0</version>
    <scope>provided</scope>
</dependency>
```

Gradle

```
compileOnly 'jakarta.platform:jakarta.jakartaee-core-api:10.0.0'
```

For rapidly evolving projects or those in the exploratory phase, starting with the full MicroProfile dependency might be advantageous. However, for production applications with well-defined requirements, opting for individual specifications can lead to more efficient and maintainable solutions.

When choosing MicroProfile modules, start with the minimal set that meets your application's core requirements. You can always integrate additional specifications as your application evolves. This approach keeps your application lightweight and focused on its primary functionalities, improving maintainability and performance. Always consider the compatibility between different versions of MicroProfile and your runtime environment to ensure seamless integration and deployment.

Manually

Create a Java Project with the following folder structure:

First create a folder named `mp-ecom-store`. This will be the root of your Maven project. Next, Inside the root folder, create a file named `pom.xml` and add the project's metadata (groupId, artifactId, version). Finally, add essential dependencies for Microprofile, Jakarta EE and Lombok.

The content for the *pom.xml* file should look as below, ensure MicroProfile dependency is added:

```
<?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>io.microprofile.tutorial</groupId>
 <artifactId>mp-ecomm-store</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>

   <!-- Setting the source and target of the Java Compiler -->

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputE
ncoding>
   <maven.compiler.source>17</maven.compiler.source>
   <maven.compiler.target>17</maven.compiler.target>

...
...
 <dependencies>

   <!-- Add Lombok dependency -->
   <dependency>
     <groupId>org.projectlombok</groupId>
     <artifactId>lombok</artifactId>
     <version>1.18.26</version>
     <scope>provided</scope>
   </dependency>

   <!-- Adding Jakarta EE dependencies -->
   <dependency>
     <groupId>jakarta.platform</groupId>
     <artifactId>jakarta.jakartaee-api</artifactId>
     <version>10.0.0</version>
     <scope>provided</scope>
```

```
    </dependency>

    <!-- Adding MicroProfie dependency -->
    <dependency>
      <groupId>org.eclipse.microprofile</groupId>
      <artifactId>microprofile</artifactId>
      <version>6.1</version>
      <type>pom</type>
      <scope>provided</scope>
    </dependency>

    <!-- JUnit Jupiter API for writing tests -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.8.2</version>
        <scope>test</scope>
    </dependency>

    <!-- JUnit Jupiter Engine for running tests -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.8.2</version>
        <scope>test</scope>
    </dependency>

  </dependencies>

 ...
```

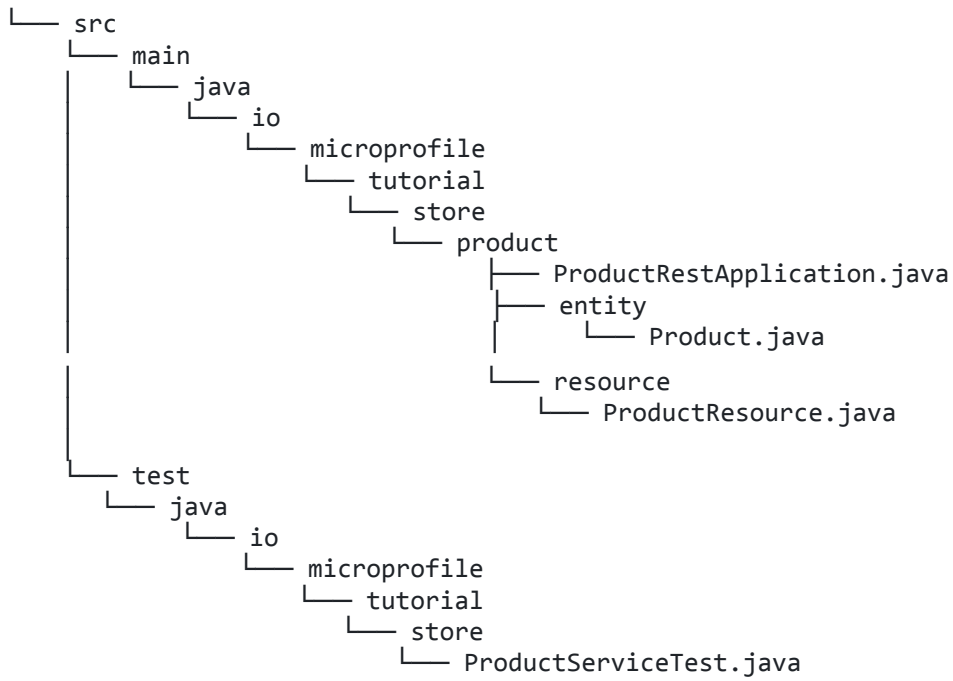
Tip: Execute the `$ mvn validate` command. This checks the `pom.xml` file for correctness, ensuring that all necessary configuration is present and valid.

The heart of your Maven project is `pom.xml`. It defines project metadata, dependencies, and build configurations.

Lombok helps keep your code concise and focused on the domain logic. The Lombok changes are reflected when you compile your project, not directly in the source file.

Next, create the source code and test directory structure as below:

```
mp-ecomm-store
├── pom.xml
├── README.md
```



Standard location of Java source code.

```
src/main/java
```

Standard location of test code.

```
src/test/java
```

You can create these directories from the command line (using `mkdir` command on Unix/Linux or `md` on Windows) or through your file explorer.

Some MicroProfile implementations allow embedding a server, you'll likely want an application server. Consider lightweight servers like OpenLiberty, WildFly, or Payara Micro.

By following these steps, you've created a basic Java project structure using Maven, including MicroProfile, Jakarta EE and Lombok dependencies, ready for further development.

Developing a RESTful Web Service

Web Services are very popular nowadays because they allow for building decoupled systems – services can communicate with each other without the knowledge of each other's implementation details.

There are many different ways to design and implement web services. One popular way is to use the **Representational State Transfer (REST)** architecture.

A Jakarta RESTful Webservice is a web service that uses the Representational State Transfer (REST) architecture. This type of web service makes it easy to build modern, scalable web applications. The REST architecture is based on the principle that all data and functionality should be accessed through a uniform interface. This makes it easy to develop, test, and deploy web applications.

To understand this better, let's create a simple RESTful service to manage a list of products for our sample application, the MicroProfile ecommerce store. This RESTful API will allow client applications to access the product information stored as resources on the server. For example, let's say you have a product catalog that you want to make available as a web service. With REST, you would create a URL that represents the resources (products) in your catalog. When a client (such as a web browser) requests this URL, the server would return a list of products in JSON format.

Creating an Entity class

An Entity class represents a specific object, in our case a product. It contains the product's details id and name, and other properties like price, quantity etc. To implement an entity class first, you need to create a Product class, as below:

```
package io.microprofile.tutorial.store.product.entity;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Product {
   private Long id;
   private String name;
   private String description;
   private Double price;
}
```

Explanation:

- The `Product` class is a Plain Old Java Object (POJO). It has an `id`, `name`, `description` and `price` property. The `id` property is of type `Long`, The `name` and `description` properties are of type `String`. The `price` property is of type `Double`.
- `@Data` annotation will generate constructors, getters, and setters for all fields. By doing this, you enable the Jackson library to convert your Java objects to JSON and vice versa. All properties must be of `Object` type as well. Jackson cannot work with primitive types because they cannot be `null`.
- `@AllArgsConstructor` generates a constructor with one argument for each field in the class. This is useful for instantiating objects with all their fields initialized.
- `@NoArgsConstructor` generates a default constructor for the class.

Creating a Resource class

A resource class represents a collection of related resources. It includes methods for creating, updating, deleting, and retrieving (CRUD) operations on the resources.

Let us now create a `ProductResource` class with a `getProducts()` method to return a list of `Product` objects.

```
// ProductResource.java
package io.microprofile.tutorial.store.product.resource;

import java.util.ArrayList;
import java.util.List;

import io.microprofile.tutorial.store.product.entity.Product;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("/products")
@ApplicationScoped
public class ProductResource {
   private List<Product> products;

   public ProductResource() {
       products = new ArrayList<>();

       products.add(new Product(Long.valueOf(1L), "iPhone", "Apple iPhone 15",
Double.valueOf(999.99)));
       products.add(new Product(Long.valueOf(2L), "MacBook", "Apple MacBook Air",
Double.valueOf(1299.99)));
   }

   @GET
   @Produces(MediaType.APPLICATION_JSON)
   public List<Product> getProducts() {
       // Return a list of products
       return products;
   }
}
```

Explanation:

- The `ProductResource` is annotated with `@ApplicationScoped`. This will ensure that this class is available as long as the application is running.
- The `ProductResource` class has a `getProducts()` method, which returns a list of products. This method is annotated with the `@GET` annotation, which maps this method to the `GET` HTTP method.
- The `@Produces` annotation tells the server that this method produces JSON content. This will return the following JSON response when we make a `GET` request to the `/api/products` endpoint.

- RESTful web services can produce and consume many different media types, including JSON, XML, and HTML.
- Annotations specify the media type that a method can consume or produce. For example, if a method is annotated with `@Produces(MediaType.APPLICATION_JSON)` it can produce JSON.

Creating an Application class

Create a class named ProductRestApplication as per the code below:

```java
// ProductRestApplication.java
package io.microprofile.tutorial.store.product;

import jakarta.ws.rs.ApplicationPath;
import jakarta.ws.rs.core.Application;

@ApplicationPath("/api")
public class ProductRestApplication extends Application{

}
```

Explanation:

- The annotation `@ApplicationPath("/api")` specifies that any RESTful resources registered within this application will be accessed under the base path `/api`. For example, if you have a resource class named `ProductResource` mapped to the path `/products`, it would be accessible at `/api/products`.

Building Your Application

You may build the application using the following commands from your project's root directory:

```
$ mvn compile
```

The above command will compile your project's source code.

```
$ mvn test
```

The above command will run the test using a unit testing framework. These test should not require the code to be packaged and deployed.

```
$mvn package
```

The above command will create a deployment package.

Deploying your microservices

This section guides you through deploying your newly created product microservice to a runtime environment. Below are some of the general considerations:

General Considerations:

- Runtime Compatibility: Ensure your chosen runtime supports the MicroProfile version used in your project.
- Packaging: Decide on a packaging format (e.g., WAR file, Docker image).
- Configuration: Review and adjust any runtime configuration necessary for your service.
- Deployment Tools: Leverage runtime-specific tools or commands for deployment.

Deployment Options

You can then deploy this application on a MicroProfile compatible server and access the web service at http://localhost:<port>/<contextRoot>/api/products. Replace `<port>` with the port number on which the web server or application server is listening. The `<contextRoot>` is a placeholder for the context root of the web application. The context root is part of the URL path that identifies the base path for the application on the web server.

Below are the steps for popular options. Specific steps will depend on your chosen runtime.

Open Liberty

- Package your application as a WAR file using Maven or Gradle by adding the packaging tag in pom.xml

```
 <groupId>io.microprofile.tutorial</groupId>
 <artifactId>mp-ecomm-store</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>
```

- Add a server configuration file at the location `/main/liberty/config/server.xml` with the content as below:

```
<server description="MicroProfile Tutorial Liberty Server">
    <featureManager>
        <feature>restfulWS-3.1</feature>
        <feature>jsonb-3.0</feature>
    </featureManager>

    <httpEndpoint httpPort="${default.http.port}"
httpsPort="${default.https.port}"
                  id="defaultHttpEndpoint" host="*" />
```

```
    <webApplication location="mp-ecomm-store.war"
contextRoot="${app.context.root}"/>
</server>
```

- Add the Open Liberty configuration in the `pom.xml` as below:

```
<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
   <maven.compiler.source>17</maven.compiler.source>
   <maven.compiler.target>17</maven.compiler.target>
   <!-- Liberty configuration -->
   <liberty.var.default.http.port>9080</liberty.var.default.http.port>
   <liberty.var.default.https.port>9443</liberty.var.default.https.port>
   <liberty.var.app.context.root>mp-ecomm-store</liberty.var.app.context.root>
 </properties>
```

- Add the Open Liberty build plugin in the `pom.xml` as below:

```
<build>
    <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.3.2</version>
            </plugin>
            <plugin>
                <groupId>io.openliberty.tools</groupId>
                <artifactId>liberty-maven-plugin</artifactId>
                <version>3.8.2</version>
                <configuration>
                    <serverName>productServer</serverName>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <systemPropertyVariables>
                        <http.port>${liberty.var.default.http.port}</http.port>
                        <war.name>${liberty.var.app.context.root}</war.name>
                    </systemPropertyVariables>
                </configuration>
            </plugin>
        </plugins>
  </build>
```

Run Your Application

Refer to your runtime's documentation for instructions on running your MicroProfile application. For example, Consult the Open Liberty documentation for detailed instructions: [MicroProfile - Open Liberty Docs]

Finally, use the `mvn liberty:run` command from the command line or terminal to run the application on Liberty server. You can also run the following command to start the liberty server in development mode.

```
mvn liberty:dev
```

Assuming your server is running on http://localhost:9080/, you can access your service at:

```
http://localhost:9080/mp-ecomm-store/api/products
```

To call this RESTful web service, you can enter the URL in your browser. The response is an array of JSON objects. Each object has an `id`, `name`, `description` and `price` property. Please note only GET methods can be tested with browsers.

The response should be

```
[{"description":"Apple iPhone
15","id":1,"name":"iPhone","price":999.99},{"description":"Apple MacBook
Air","id":2,"name":"MacBook","price":1299.99}]
```

This uses an in-memory list; In the next chapter, in a real application you should integrate a database (via Jakarta Persistence API). We will be learning about this in the next chapter.

Quarkus

- Build your application as a native executable or Docker image.
- Run the generated executable or deploy the Docker image to a container platform.
- Refer to the Quarkus documentation for deployment guides: Creating your first application - Quarkus

Payara Micro

- Package your application as a WAR file.
- Deploy the WAR to a Payara Micro server instance.
- See the Payara Micro documentation for specific instructions: Getting Started with Payara Micro

WildFly

- Package your application as a WAR file.
- Deploy the WAR to a WildFly server instance.
- Refer to the WildFly documentation for deployment details: WildFly Developer Guide

Helidon

- Choose between Helidon SE (native packaging) or Helidon MP (WAR packaging).
- Build your application using Gradle.
- Follow the relevant Helidon documentation for deployment steps: Helidon - Getting Started

TomEE:

- Package your application as a WAR file.
- Deploy the WAR file to the TomEE server instance.
- Refer to the TomEE documentation for instructions: Serverless TomEE MicroProfile

Additional Considerations:

- Containerization: Consider using containerization technologies like Docker and Kubernetes for portability and scalability.
- Cloud Deployment: Explore cloud platforms like AWS, Azure, or GCP.

Testing your Microservice

Testing your microservice is critical for ensuring the reliability and robustness of your microservice. Maven, being a powerful project build management tool, simplifies this process by automating the test execution.

To create tests for your microservice, start by creating a class called `ProductResourceTest`, which contains unit tests for the `ProductService` class as below:

```
package io.microprofile.tutorial.store.product.resource;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

import java.util.List;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import io.microprofile.tutorial.store.product.entity.Product;

public class ProductResourceTest {
  private ProductResource productResource;

    @BeforeEach
    void setUp() {
        productResource = new ProductResource();
    }

    @AfterEach
    void tearDown() {
        productResource = null;
    }

    @Test
    void testGetProducts() {
        List<Product> products = productResource.getProducts();
```

```
        assertNotNull(products);
        assertEquals(2, products.size());
    }
}
```

Below are the assertions to test the `getProducts()` method of `ProductService`.

- The `assertNotNull(products);` assertion checks that products are not null. It ensures the method returns a list, even if it's empty.
- The `assertEquals(2, products.size());` assertion verifies that the list contains two products.

Next Steps

Now that you have a basic MicroProfile service, consider exploring further:

- Adding configuration with MicroProfile Config
- Implementing health checks using MicroProfile Health
- Enhancing your service with MicroProfile Fault Tolerance

Resources

- MicroProfile Official Website: https://microprofile.io/
- MicroProfile GitHub Repository: https://github.com/eclipse/microprofile
- MicroProfile Documentation and Guides: [Official documentation](https://microprofile.io/documentation/)

After completing this chapter, you should have a basic understanding of MicroProfile and how to start building microservices with it. Continue exploring the specifications and capabilities of MicroProfile to fully leverage its power in your microservices architecture.

Package Structure

The Table below provides an overview of the package structure and their purposes within a typical Java-based microservices architecture.

Package	Description
`dto`	Data Transfer Objects (DTOs) are used to transfer data between processes, such as from your service to a REST endpoint. They often mirror entity classes but can be tailored to the needs of the client to avoid over-fetching or under-fetching data.

`entity`	Entity classes represent the domain model and are typically mapped to database tables. These are the core classes that represent the business data and are often used with ORM tools like JPA.
`repository`	Interfaces in this package abstract the data layer, making it easier to perform CRUD operations without dealing with database intricacies directly. This follows the Repository pattern. Data access layer, interacting with databases or other storage mechanisms (e.g., ProductRepository, CustomerRepository)
`resource`	REST resource classes (sometimes called controllers in other frameworks) are the entry points for HTTP requests. They interact with service classes to process these requests. Interfaces defining endpoints for REST services (e.g., ProductResource, ShoppingCartResource)
`common`	This package contains classes and interfaces that are shared across different microservices, such as utility classes, common configuration, exception handling, and security-related classes.
`client`	For microservices to communicate with each other, they often use HTTP clients. This package contains interfaces or classes annotated for use with MicroProfile Rest Client or similar, facilitating easy communication between your services.
`config`	Configuration classes for MicroProfile Config
`health`	
`exception`	Custom exceptions for error handling (e.g. ProductNotFoundException, PaymentFailedException)
`util`	Helper and utility classes

Base Package: `io.microprofile.tutorial.store`

```
io.microprofile.tutorial.store
├── catalog
│     ├── resource
│     ├── config
│     ├── exception
```

```
│       ├── entity
│       ├── repository
│       └── util
├── cart
│       ├── resource
│       ├── entity
│       ├── repository
│       ├── client
│       ├── exception
│       └── util
├── user
│       ├── resource
│       ├── entity
│       ├── repository
│       ├── exception
│       └── util
├── inventory
│       ├── resource
│       ├── entity
│       ├── repository
│       ├── exception
│       └── util
├── order
│       ├── resource
│       ├── entity
│       ├── repository
│       ├── exception
│       └── util
├── payment
│       ├── resource
│       ├── entity
│       ├── repository
│       ├── exception
│       └── util
└── shipment
        ├── resource
        ├── entity
        ├── repository
        ├── exception
        └── util
```

// TODO: Current package structure is just a proposal will update after completing the source code for all chapters

Glossary

Java Development Kit (JDK)

A software development environment used for developing Java applications. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a

compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

Integrated Development Environment (IDE)

A software application that provides comprehensive facilities to computer programmers for software development. Examples include Eclipse, IntelliJ IDEA, NetBeans, and Visual Studio Code.

RESTful Service

A web service implementing REST (Representational State Transfer) principles, providing interoperability between computer systems on the internet.

Runtime Environment

The environment in which programs are executed. It includes everything your application needs to run in production, such as an operating system, a runtime (like JVM for Java applications), libraries, and environment variables.

JUnit

A unit testing framework for Java, used to write and run repeatable tests.

Containerization

A lightweight alternative to full machine virtualization that involves encapsulating an application in a container with its own operating environment.

Cloud Deployment

Deploying applications in cloud environments, leveraging cloud resources like compute instances, storage, and networking capabilities.