MicroProfile Tutorial

Version 6.1, 2025-07-13

Preface

```
MicroProfile API Tutorial

Version: 6.1

Status: Draft
```

Legal

Preface

About this Tutorial

In this tutorial, you will learn how to use the features of the MicroProfile Platform by building a microservices-based e-commerce application named "MicroProfile e-Commerce". The tutorial will cover using MicroProfile APIs such as Config, REST Client, JWT, Fault Tolerance, and Metrics to build efficient, scalable, and resilient microservices for cloud-native applications. We aim to provide a comprehensive overview and hands-on knowledge about using MicroProfile APIs.

Who is this Tutorial for

This tutorial caters to software professionals, from beginners to senior developers, engineering managers, and architects, to adeptly utilize MicroProfile in real-world projects.

What will be Covered

First, an overview of the MicroProfile project is presented, followed by detailed sections on each specification, complete with thoroughly tested and updated code samples.

Project

In this tutorial, you will learn to build a microservices-based e-commerce application called "MicroProfile e-Commerce". The app will demonstrate the use of MicroProfile APIs for developing an application based on microservices and cloud-native architecture.

It would include multiple microservices, each serving a different purpose and highlighting different aspects of MicroProfile. Java developers can use this adaptation as a practical case study to implement MicroProfile APIs in real-world applications.

The MicroProfile e-Commerce application compirses of multiple microservices, among the key ones are as below:

- **Product Catalog**: This service acts as the central repository for all product-related information, including detailed descriptions, and pricing, and inventory levels. It provides APIs for fetching product details efficiently for the other microservices, such as the Shopping Cart. This microservice is vital for updating product data, ensuring data consistency and accuracy across the e-Commerce platform.
- **The Shopping Cart**: This service allows users to add or remove products from their shopping cart. It communicates with the Product Catalog Microservice to access up-to-date product information. It handles the storage and management of cart items for each user, including the calculation of cart totals with applicable discounts or promotions. This microservice plays interfaces with the Checkout microservice to initiate the order processing.
- **User Management**: This service is responsible for user account management, handles registration, login, and account updates securely using JWT tokens. It is essential for personalizing the user experience and safeguarding user information.
- **Order Processing**: This service manages the entire order process, from collecting shipping information and confirming order details to initiating payment processing. This microservice ensures a seamless transition from shopping to order completion.
- **Payment**: Dedicated to processing payments, this microservice interacts with external payment gateways to securely handle transactions. It receives payment instructions from the Checkout microservice, executes the payment process, and confirms transaction outcomes. This microservice is crucial for ensuring financial transactions are conducted securely and efficiently, maintaining the integrity of the payment process.
- **Inventory**: This service is dedicated to monitoring and managing inventory levels. It tracks product availability, updates inventory in real-time as sales occu, and provides restocking alerts. By integrating with the Product Catalog and Checkout microservices, it ensures that product availability is accurately reflected on the platform and that orders are only placed for in-stock items. This microservice is crucial for maintaining optimal inventory levels and preventing stockouts, thereby enhancing the customer shopping experience.
- **Shipping**: This microservice is responsible for managing the logistics of order delivery. It

receives order details and shipping information from the Order Processing Microservice, ensuring that orders are shipped to customers in a timely and efficient manner. The Shipping Microservice plays a critical role in the post-purchase customer experience, managing expectations and communication regarding order delivery.

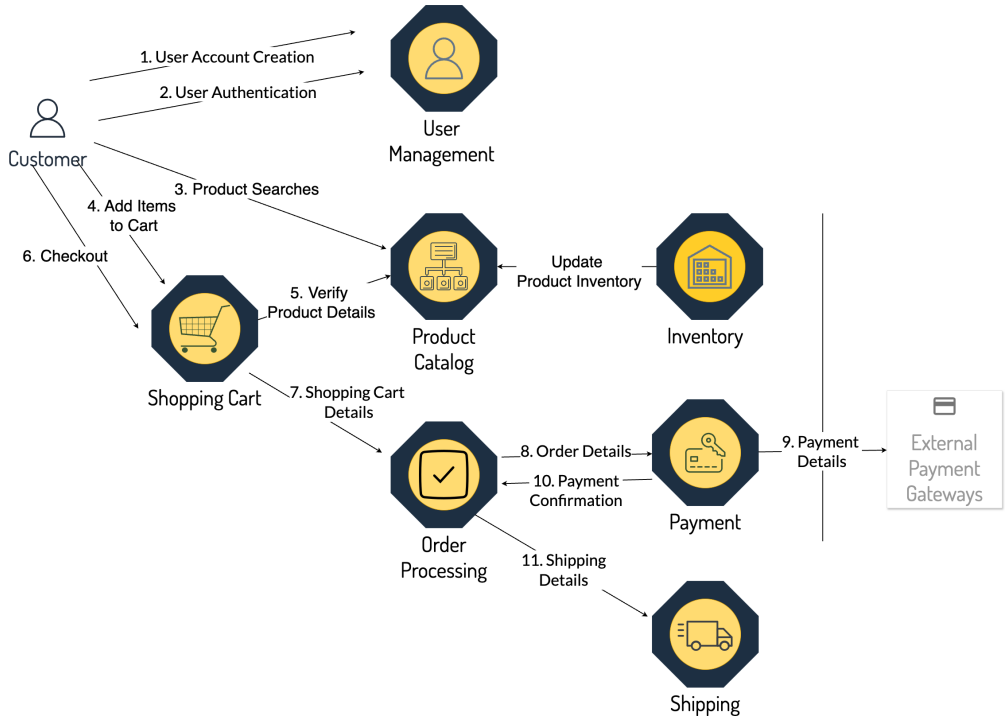


Figure 1. MicroProfile e-Commerce Application

As you can see in the above figure, together these microservices form a robust and flexible e-Commerce application architecture, enabling scalable, efficient, and secure online shopping experiences.

Downloading the Code

The code examples in this tutorial are available at this repo.

Prerequisites

MicroProfile uses the Java Platform, and are usually written in the Java programming language. All the examples in this tutorial are written in Java. If you're new to Java, spend some time getting up to speed on the language and platform; a good place to start is dev.java/learn.

Each topic in this tutorial provides some background information, but in general, we assume you have a basic understanding of RESTful Web Services.

Learning Objectives

- Understanding MicroProfile and Its Ecosystem:
  - Gain a solid understanding of what MicroProfile is and its role in modern cloud-native application development.
  - Learn about the evolution of MicroProfile and its relationship with Jakarta EE.

    - Understand how MicroProfile facilitates building microservices.
- Hands-On Experience with Key MicroProfile APIs:
    - Learn to implement Config, Health, Metrics, JWT Authentication, Fault Tolerance, Rest Client, and other MicroProfile APIs.
    - Understand how to apply these APIs in practical scenarios through the Duke's Forest application case study.
- Building Resilient and Scalable Services:
    - Master techniques for developing resilient services using fault tolerance and health checks.
- Securing Microservices:
    - Learn the intricacies of securing microservices using MicroProfile JWT and Security API.
- Effective Data Management in Microservices:
    - Understand the role of JPA and JSON-B in MicroProfile for handling data operations in microservices.
- Monitoring and Tracing:
    - Implement monitoring strategies using MicroProfile Metrics.
    - Learn to trace microservice interactions with OpenTracing for enhanced observability.
- Collaborative Learning and Community Engagement:
    - Participate in Q&A sessions, forums, and interactive discussions.
    - Engage with the MicroProfile community for continuous learning and staying updated with the latest trends.

By the end of this tutorial readers will gain the knowledge and skills necessary to design, develop, and deploy robust microservices using MicroProfile, preparing them for advanced roles in software development and architecture in cloud-native environments.

Conventions

Convention	Meaning	Example
Boldface	Boldface type indicates a term defined in text or graphical user interface elements associated with an action.	A **cache** is a copy stored locally. From the **File** menu, choose **Open Project**.
`Monospace`	Monospace type indicates the names of files and directories, commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

Convention	Meaning	Example
Italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.	Read Chapter 6 in the *User's Guide*. Do *not* save the file. The command to remove a file is `rm filename`.

Introduction to MicroProfile

Introduction

This introductory chapter provides a comprehensive overview of the MicroProfile platform, setting the stage for subsequent chapters in this tutorial. It aims to familiarize you with the fundamentals of the MicroProfile platform, its need, and benefits. Finally, we will discuss its place in the broader context of enterprise Java development.

Topics to be covered:

- What is MicroProfile
- Need for MicroProfile
- MicroProfile Specifications
- Current MicroProfile Implementations
- Architecture Philosophy
- Benefits of MicroProfile
- Relationship with Jakarta EE specification

What is MicroProfile

MicroProfile is an open-source specification that enhances enterprise Java technologies for microservices development. It provides a set of APIs and specifications for building modern, scalable, resilient, and efficient microservices-based applications. The primary goal of MicroProfile is to simplify the development for Java developers, enabling them to create applications optimized for cloud-native-development.

MicroProfile was initiated in June 2016 by a collaboration of industry leaders, Java community members, and individual contributors. In the following year, the project was transitioned to the Eclipse Foundation to enhance the project's openness and vendor-neutral stance. Now, MicroProfile has become a key framework for extending Java in the cloud-computing domain, offering a comprehensive suite of APIs tailored for developing microservices in a cloud-native ecosystem.

The **MicroProfile Working Group** currently comprises of the following members:

Committer Representative (Year 2024)

- Emerson Castañeda

Java user groups

- Atlanta Java User Group (AJUG)
- Association of the German Java User Groups (iJUG)

Corporate Members

- IBM
- Fujitsu
- Red Hat
- Primeton
- Payara
- Microsoft
- Tomitribe
- Oracle

This collective effort demonstrates MicroProfile's commitment to evolving Java enterprise development for the modern cloud environment, leveraging the expertise of its community.

Need for MicroProfile

The MicroProfile Specification was developed to address the following requirements:

- **Microservices Architecture Adoption**: The industry shift towards microservices architecture has brought several advantages, including improved flexibility, scalability, and speed of deployment. However, it also introduced several new challenges for developers due to the added complexities. These include ensuring seamless integration between microservices, securing each microservice individually as well as interactions between them, managing performance and efficiency, designing microservices to be fault-tolerant and resilient to failures, ensuring data consistency across services, managing configurations across multiple environments and managing various independently deployable components. To address these challenges, MicroProfile provides a simplified and optimized set of APIs designed to build and deploy Java-based microservices applications.
- **Limitations of Traditional Enterprise Java**: Traditional enterprise Java frameworks, like Java EE (now Jakarta EE), were often seen as too monolithic and heavyweight for microservices while evolving too slowly. It led to a demand for a more streamlined and microservices-focused framework. MicroProfile fills this gap by providing a lightweight alternative optimized for microservices development.
- **Cloud-Native Application Development**: The rise of cloud-native applications necessitated new features such as external configuration, health checks, and fault tolerance, which existing Java standards did not adequately address. MicroProfile bridges these gaps left, making it easier for developers to create resilient, scalable, and manageable microservices for cloud-native application development using Java.
- **Community-Driven Innovation**: The rapid pace of technological change in microservices necessitated a collaborative platform for innovation. MicroProfile, backed by community and vendor support, promotes rapid evolution to meet these demands.
- **Vendor Neutrality and Interoperability**: There was a need for a framework that could provide standardization across different implementations and environments, ensuring compatibility and avoiding vendor lock-in.
- **Focus on Simplicity and Productivity**: Developers needed a simple, easy-to-understand

framework that increased productivity by reducing boilerplate code and focusing on essential microservice functionalities. Well-defined standards and patterns eliminate the need to reinvent the wheel, allowing developers to focus on microservices logic.

- **Support for familiar programming model**: MicroProfile was founded with support for Jakarta JSON Processing, Jakarta JSON Binding, Jakarta RESTful Web Services, and Jakarta Contexts and Dependency Injection (CDI) to define the core programming model and accelerate adoption.
- **Lightweight and Resilient Services**: With the microservices architecture, there's a need for frameworks that support the development of lightweight, resilient, and independently deployable services, which are essential for microservices.
- **Rapid Adaptation to New Trends**: The technology landscape, especially around microservices, is constantly evolving. A framework like MicroProfile, which is community-driven and rapidly evolving, can adapt quickly to these changes, continually incorporating new practices and technologies, including:
  - **Streaming APIs and Reactive Programming Model**: To facilitate non-blocking communication and data processing, enhancing system responsiveness and scalability.
  - **API-First Development (Open API)**: Emphasizing the design and documentation of microservices with an API-first approach, promoting interoperability and clear service contracts.
  - **Eventual Consistency and Long Running Actions (LRA)**: Addressing the challenges of data consistency in distributed systems without compromising system performance.
- **Enhanced Observability and Monitoring**: Microservices architectures complicate application monitoring and observability. A framework with built-in support for these capabilities simplifies the management of distributed services.

MicroProfile Specifications

MicroProfile specifications are divided into two main categories: Platform and Standalone.

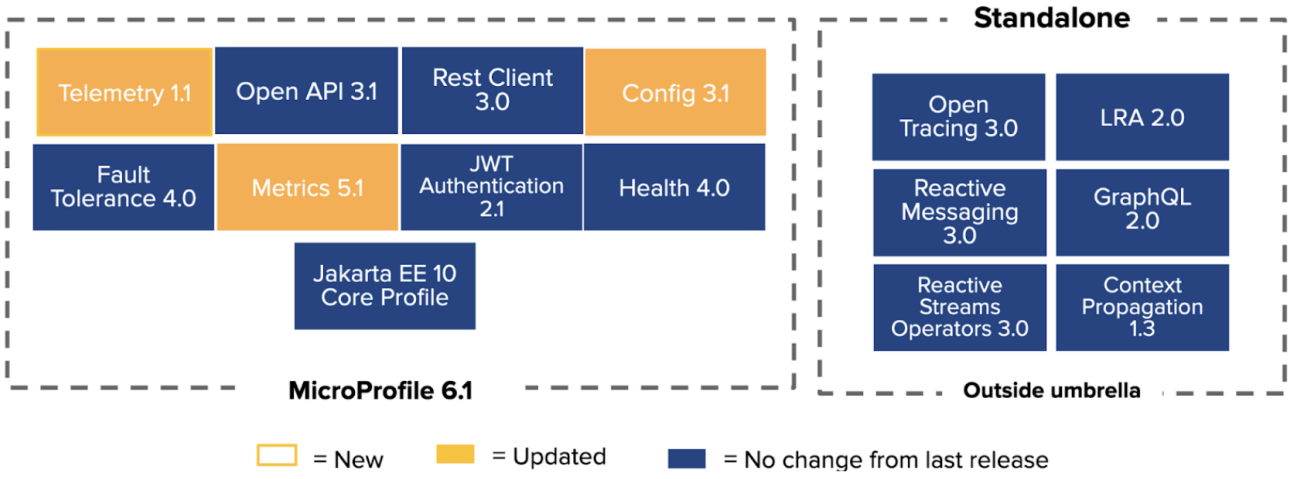


Figure 2. MicroProfile Specifications

MicroProfile Platform Component Specifications

The MicroProfile Platform Specification is the core set of MicroProfile specifications designed to provide the foundational functionalities needed for microservices development. These specifications solve specific microservices challenges, including configuration, fault tolerance, health checks, metrics, and security. The table below provides a list of platform specifications of MicroProfile along with their descriptions:

Specification	Description
Config	Provides an easy-to-use and flexible system for application configuration.
Fault Tolerance	Implements patterns like Circuit Breaker, Bulkhead, Retry, Timeout, and Fallback for building resilient applications.
JWT Authentication	Defines a standard for using OpenID Connect (OIDC) based JSON Web Tokens(JWT) for role-based access control(RBAC) of microservices endpoints for secure communication.
Metrics	Define custom application metrics and expose platform metrics on a standard endpoint using a standard format to external monitoring systems.
Health	Allows applications to expose their health and readiness to perform operations to the underlying platform, which is crucial for automated recovery in cloud environments.
Open API	Facilitates the generation of OpenAPI documentation for RESTful services, making API discovery and understanding easier.
Telemetry	Provides a unified set of APIs, libraries, and tools for collecting, processing, and exporting telemetry data (metrics, traces, and logs) from cloud-native applications and services.
Rest Client	Defines a type-safe approach to invoke RESTful services over HTTP(S), simplifying the development of Rest clients.
Jakarta EE Core Profile 10	An optimized Jakarta EE platform designed specifically for developing microservices and cloud-native Java applications with a reduced set of specifications for a lighter runtime footprint.

Standalone (Outside Umbrella) Specifications

Standalone specifications address more advanced needs that every microservices application may

not require. They allow for innovation and experimentation in areas that are evolving or where there's a need to address niche concerns without burdening the core platform with additional complexity. The table below provides a list of standalone specifications of MicroProfile along with their descriptions:

Specification	Description
Context Propagation	Defines a way to propagate context between threads and managed executor services. Ensure that the context is consistent during executing asynchronous tasks or across different services.
GraphQL	Provides a layer on top of Jakarta EE that allows the creation of GraphQL services. This specification makes it easier to build APIs, enabling clients to request exactly the data they need and nothing more.
Long Running Actions (LRA)	Focuses on providing a model for developing services that participate in long-running processes, ensuring consistency and reliability without necessarily locking data.
Reactive Messaging	Aims to facilitate building applications that communicate via reactive streams, enabling the development of event-driven, responsive, and resilient microservices.
Reactive Streams Operators	Provides a way to process data streams in a reactive manner, allowing for non-blocking system design and improving the efficiency of data processing in microservices.
Open Tracing	Integrates distributed tracing by defining a way for services to trace requests across service boundaries, improving observability.

MicroProfile Implementations

Below is the list of MicroProfile Implementations, each offering a platform for building and running microservices-based applications:

- Payara Micro
- Apache TomEE
- Open Liberty
- Launcher
- Quarkus
- WildFly
- Helidon

Architecture Philosophy

The overall goal of MicroProfile architecture is to provide a lightweight enterprise-grade framework tailored for building cloud-native applications and enabling developers to build and deploy microservices with Java easily:

- **Simplicity**: MicroProfile APIs are designed to be simple and easy to use. They avoid unnecessary complexity and focus on providing the essential functionality for building microservices.
- **Modularity**: Its modular approach allows developers to use only what they need, reducing the overhead typically associated with enterprise frameworks.
- **Standards-based**: MicroProfile is based on open standards and specifications, ensuring compatibility and consistency across different implementations.
- **Community-driven**: It encourages active participation from the Java community for continuous evolution.
- **Vendor-Neutral**: MicroProfile is vendor-neutral. It's supported by several industry players, ensuring that no single company controls its direction.
- **Focus on Cloud-Native Applications**: The architecture is specifically tailored for cloud environments. MicroProfile integrates with a number of cloud-native technologies, such as Kubernetes and Istio. This makes it easy to deploy and manage MicroProfile applications in cloud environments.
- **Reactive programming**: MicroProfile supports reactive programming, which is a style of programming that is well-suited for building microservices. Reactive applications are responsive and scalable, and they can handle high volumes of concurrent requests.

Figure 3. Architecture Philosophy of MicroProfile

Benefits of MicroProfile

MicroProfile offers several benefits, making it a compelling choice for developing microservices, especially in Java-centric environments. These benefits include:

- **Optimized for Microservices**: MicroProfile is designed explicitly for creating microservices, offering APIs that cater to the unique challenges of this architectural style.
- **Cloud-Native Focus**: The framework includes features such as externalized configuration, health checks, and metrics, which are essential for building and operating cloud-native applications effectively. MicroProfile is inherently designed for cloud-native applications.
- **Open Source and Standards-Based**: As an open-source framework based on open standards, MicroProfile facilitates interoperability and reduces the risk of vendor lock-in.
- **Enhanced Productivity, Rapid Development and Deployment**: MicroProfile simplifies microservices development with a set of standard APIs. With its focus on simplicity and productivity, MicroProfile helps speed up the development and deployment of microservices by providing essential functionalities and reducing boilerplate code.
- **Community-Driven Innovation**: Being community-driven, MicroProfile evolves quickly, incorporating new trends and best practices in microservices development. MicroProfile is backed by a strong Java community, ensuring continuous improvement and support.
- **Vendor Neutrality**: Being vendor-neutral, MicroProfile is supported by a wide range of industry players, which ensures a broad choice of tools and platforms for developers.
- **Compatibility with Jakarta EE**: MicroProfile is complementary to Jakarta EE, whether using

MicroProfile implementations that support a small subset of Jakarta EE (such as Core Profile) or implementations that extend the full Jakarta EE Platform implementations with MicroProfile.

- **Lightweight and Modular**: It provides a lightweight model compared to traditional enterprise Java frameworks. Its modularity allows developers to use only the necessary components, reducing the application's footprint and overhead.
- **Scalability**: The framework supports the development of scalable applications, essential for microservices that handle varying loads efficiently.
- **Enhanced Resilience**: MicroProfile includes specifications for fault tolerance patterns like retries, circuit breakers, timeouts, and bulkheads, which are crucial for building resilient services that can withstand network and service failures.
- **Security Features**: MicroProfile's JWT Authentication provides a standardized way to secure microservices, making it easier to implement authentication and authorization.
- **Ease of Testing**: With its lightweight nature and support for advanced features like Rest Client, MicroProfile simplifies the testing of microservices, both in isolation and in integration scenarios.

Relationship with Jakarta EE specification

Jakarta EE is an open specification with more than 40 component specifications to address a wide array of needs of enterprise Java development. MicroProfile complements this by providing a baseline platform definition that optimizes enterprise Java for microservices architecture and delivers application portability across multiple compatible runtimes. Many Jakarta EE implementations that target a broad array of applications supplement Jakarta EE with MicroProfile to better support microservices. Their coexistence allows developers to harness the strength of both platforms, thereby facilitating a more versatile and adaptive approach to modern enterprise and cloud-native application development. MicroProfile strategically leverages Java EE developers' existing skill sets, enabling them to transition and adapt to microservices development with minimal learning curve. This ensures that developers can easily design and implement microservices architecture, enhancing productivity and facilitating the creation of cloud-native applications. Later in this tutorial, we will explore how MicroProfile API extends Jakarta EE's capability to address microservices-specific challenges.

MicroProfile and Jakarta EE are complementary technologies. Both platforms enable developers to stay at the forefront of cloud-native application development.

Conclusion

In this section, we explored the MicroProfile platform in detail, laying the foundation for understanding how it revolutionizes the development of microservices using Java. We started by defining MicroProfile, emphasizing its role as an open-source specification tailored for microservices development. Key contributions from industry leaders and community members have positioned MicroProfile as a pivotal technology in the Java ecosystem, especially for cloud-native application development. We delved into the essential specifications of MicroProfile, each playing a critical role in addressing specific challenges in microservices development, from configuration management to service resilience. As we move forward in this tutorial, we will delve

deeper into each specification and discover how to implement MicroProfile in real-world Java applications effectively.

Glossary

Microservices

An architectural style for building applications as a collection of small, independent services. Each service focuses on a specific business capability and communicates with other services through well-defined APIs.

APIs (Application Programming Interfaces)

A set of definitions and protocols that specify how software components interact with each other.

Cloud-native development

An approach to building and running applications that are specifically designed for the cloud environment. It involves using technologies and practices that leverage the benefits of cloud platforms, such as scalability, elasticity, and pay-as-you-go pricing.

Eclipse Foundation Working Group

A collaborative group of industry leaders and Java community members who actively contribute to the of development of Eclipse projects like MicroProfile within the Eclipse Foundation framework.

Jakarta EE

Jakarta EE (formerly Java Platform, Enterprise Edition, or Java EE) is a set of specifications, extending Java Platform, Standard Edition, or Java SE with specifications for enterprise features such as web services, database persistence, asynchronous messaging and more.

External Configuration

A technique in application development where configuration data is separated from the application code, allowing the application's behavior to be adjusted without changing the code, especially useful in cloud-native and microservices architectures.

Health Checks

Mechanisms used in microservices architectures to continuously check the status of an application or service to ensure it is functioning correctly and available to users.

Fault Tolerance

The ability of a system to continue operating in the event of the failure of some of its components. This feature is critical for maintaining high availability and reliability in microservices architectures.

Vendor Neutrality

The principle of designing software products and standards not controlled by any single vendor, promoting user interoperability and choice.

Interoperability

The ability of a software to exchange and make use of information across different platforms and services.

JSON-P (JSON Processing)

A Jakarta EE (formerly Java EE) API that enables parsing, generating, transforming, and querying JSON data. It facilitates the processing of JSON data within the Java programming environment. Currently it is known as Jakarta JSON Processing.

JSON-B (JSON Binding)

A Jakarta EE (formerly Java EE) API for binding Java objects to JSON messages and vice versa, streamlining the serialization and deserialization process. It allows custom mappings to handle complex conversion scenarios efficiently. Currently it is known as Jakarta JSON Binding.

JAX-RS (Java API for RESTful Web Services)

A Jakarta EE API for creating web services according to the REST architectural pattern in Java, using annotations to simplify development. It enables the easy creation and management of resources via standard HTTP methods. It is currently known as Jakarta RESTful Web Services.

CDI (Contexts and Dependency Injection)

A Jakarta EE API for enterprise-grade dependency injection, offering type-safe mechanisms, context lifecycle management, and a framework for decoupling application components. It enhances modularity and facilitates the development of loosely coupled, easily testable applications.

Boilerplate Code

A piece of code that must be included in many places with little or no alteration.

Lightweight Services

Services designed to consume minimal computing resources, enhancing performance and efficiency, particularly relevant in a microservices architecture.

Resilient Services

Services built to recover quickly from failures and continue operating. It is critical for maintaining the reliability of microservices-based applications.

Observability

The ability to measure the internal state of a system by examining its outputs, crucial for understanding the performance and behavior of microservices.

Monitoring

The practice of tracking and logging the performance and status of applications and infrastructure, essential for maintaining system health in microservices environments.

Circuit Breaker

A fault tolerance mechanism that prevents a failure in one service from causing system-wide failure, by temporarily disabling failing services.

Bulkhead

A pattern that isolates failures in one part of a system from the others, ensuring that parts of an application can continue functioning despite issues elsewhere.

Retry

A simple fault tolerance mechanism where an operation is attempted again if it fails initially, based on predefined criteria.

Timeout

A mechanism to limit the time waiting for a response from a service, helping to avoid resource deadlock situations in distributed systems.

Fallback

A fault tolerance mechanism that provides an alternative solution or response when a primary method fails.

Role-Based Access Control (RBAC)

A method of restricting system access to authorized users based on their roles within an organization.

Kubernetes

An open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts.

Istio

An open platform to connect, manage, and secure microservices, providing an easy way to create a network of deployed services with load balancing, service-to-service authentication, and monitoring.

Reactive Programming

A programming paradigm oriented around data flows and the propagation of change, enabling the development of responsive and resilient systems.

Distributed Tracing

A method for monitoring applications, especially those built using a microservices architecture, by tracking the flow of requests and responses across services.

Long Running Actions (LRA)

A model for managing long-duration, distributed transactions across microservices without locking resources.

Reactive Streams

An initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.

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
- Initializing a New MicroProfile Project
- Choosing Right Modules for your Application
- Building a RESTful service
- Deployment
- 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:

1. **Download**: Visit the official OpenJDK website and download the JDK version compatible with your operating system.
2. **Install**: Follow the installation instructions provided on this official OpenJDK Installation guide.
3. **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.

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, Azul Zulu, OpenJ9 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.

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.

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 boostraping, 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 for Enterprise Java and Web Developers 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 need 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

Apache 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.

- Visit the MicroProfile Starter page - the official website for generating the MicroProfile project templates.
- 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.

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

- Enter the 'artifactID', which is the name of your project (e.g., 'mp-ecomm-store').
- Select the **Java SE version** your project will use.
- 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.
- 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.
- Click the *Download* button.
- 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.

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.

Jakarta EE 10 Core Profile

Introduction

This chapter delves into the **Jakarta EE 10 Core Profile**, a specification designed specifically for microservices and cloud-native apps. **Jakarta EE** is a comprehensive framework within the Java ecosystem for crafting enterprise-grade applications. Complementing this, **MicroProfile** addresses the intricacies of microservices development, such as configuration management, fault tolerance, health checks, and monitoring. The foundation of MicroProfile is built on the strong and established standards of Jakarta EE, which ensures smooth integration of these modern APIs with the enterprise Java landscape.

In this chapter, through practical examples, we will explore the critical features of the Jakarta EE 10 Core Profile that are most relevant to microservices development, including Contexts and Dependency Injection (CDI), Jakarta RESTful Web Services (Jakarta REST, formerly JAX-RS), JSON Binding and JSON Processing.

By the end of this chapter, you will gain a comprehensive understanding of Jakarta EE 10 Core Profile as a foundational platform for developing microservices with MicroProfile. You will be able to appreciate the pivotal role of Jakarta EE in the MicroProfile ecosystem and how its core functionalities develop scalable, resilient, and portable cloud-native applications.

Topics to be covered:

- Understanding the Jakarta EE 10 Core Profile
- Key Specifications in Core Profile
- Managing Component Dependencies
- Handling HTTP Methods and Resources
- Best Practices for Building Robust and Scalable Applications

Understanding the Jakarta EE 10 Core Profile

The Jakarta EE 10 Core Profile is a streamlined subset of the full Jakarta EE platform explicitly designed for building lightweight microservices and cloud-native applications. It provides a standardized foundation for smaller runtime environments, comprising of a curated selection of Jakarta EE specifications:

- **Jakarta Annotations**: Enables developers to decorate their code with metadata to influence system configuration and behavior, making the code concise, readable, and maintainable.
- **Jakarta Contexts and Dependency Injection Lite**: Facilitates the management of lifecycle contexts of stateful components and the injection of dependencies.
- **Jakarta Interceptors**: Offers a means to intercept business method invocations and lifecycle events, ideal for implementing cross-cutting concerns such as logging.
- **Jakarta JSON Processing and Jakarta JSON Binding**: Simplifies the parsing, generation, and

binding of JSON data for Java objects, crucial for RESTful service communication.

- **Jakarta REST**: Provides a framework for creating web services according to the REST architectural pattern, enhancing web API development.

Key Specifications in Jakarta EE 10 Core Profile

Let's delve deeper into some of the specifications included in the Jakarta Core Profile to understand their importance and functionality:

Jakarta Annotations

This specification simplifies the code by reducing the need for external configuration files and making the intentions behind code clear. Annotations are extensively used across various Jakarta EE specifications.

Key Features

- **Simplification of Configuration**: Annotations reduce the need for XML configuration files, making the setup more straightforward and less error-prone.
- **Enhanced Readability and Maintenance**: Code decorated with annotations is easier to read and maintain, as the configuration is co-located with the code it configures.
- **Wide Adoption**: Used across the Jakarta EE platform for a variety of purposes, including dependency injection, defining REST endpoints, and configuring beans.

Jakarta Contexts and Dependency Injection (CDI) - CDI Lite

CDI is the specification that unifies the Jakarta EE platform by providing a consistent way to manage the lifecycle of stateful components and their interactions. The CDI Lite section of the specification is tailored for environments where full CDI support may be too heavyweight, such as microservices and serverless deployments.

Key Features of Contexts and Dependency Injection (CDI) - CDI Lite

- **Type-safe Dependency Injection**: Enables the injection of beans in a type-safe manner, reducing runtime errors and improving developer productivity.
- **Contextual Lifecycle Management**: Manages the lifecycle of beans according to well-defined contexts, simplifying state management across different scopes.
- **Interceptors**: Supports the use of interceptors for adding behavior to beans or for altering their behavior in a non-invasive manner.

The CDI Lite Tutorial is an invaluable resource, if you are looking to gain a solid foundation in CDI Lite and its role within the Jakarta EE ecosystem, especially in the context of building lightweight microservices and cloud-native applications. It will take you through the basics, advanced features, and the practical application of CDI Lite, equipping you with the knowledge to make the most out of this powerful specification.

Jakarta Interceptors

Jakarta Interceptors allow developers to define methods that intercept business method invocations and lifecycle events on Jakarta EE components. This is particularly useful for implementing cross-cutting concerns such as logging, transactions, security, and more, without cluttering business logic.

Key Features of Jakarta Interceptors

- **Separation of Concerns**: Helps in separating cross-cutting concerns (like logging, transaction management, and security) from business logic.
- **Reusability**: Interceptors can be defined once and applied to multiple beans, promoting code reuse.
- **Configurability**: Interceptors can be enabled, disabled, or reordered through configuration, offering flexibility in their application.

For an in-depth understanding of Jakarta Interceptors, We highly recommend you to read the Jakarta Interceptors Tutorial. This tutorial covers everything from basic concepts to advanced usage scenarios, providing a solid foundation for effectively utilizing interceptors in your projects.

Jakarta JSON Processing

Jakarta JSON Processing (JSON-P) is a specification in the Jakarta EE platform that provides a portable API to parse, generate, transform, and query JSON data in a Java application. It is part of the larger ecosystem of Jakarta EE specifications designed to facilitate the development of enterprise applications with support for modern data formats and protocols, including JSON, which is widely used in web services and RESTful APIs.

Key Features of Jakarta JSON Processing

- **Parsing and Generation**: JSON-P allows for both the parsing of JSON data into a Java representation and the generation of JSON data from Java objects. This can be done using either a streaming API for efficiency with large data sets or a more intuitive object model API for ease of use.
- **Object Model API**: This API provides a way to build or manipulate JSON data using a DOM-like tree structure. It enables developers to create, access, and modify JSON data in a flexible manner.
- **Streaming API**: The streaming API (JsonParser and JsonGenerator) offers a lower-level, event-based approach to parsing and generating JSON. It is highly efficient, making it suitable for processing large volumes of JSON data with minimal memory overhead.
- **Data Binding**: While JSON-P itself does not directly support data binding (converting between JSON and Java POJOs), it lays the groundwork for such functionality, which is further extended by Jakarta JSON Binding (JSON-B).

For an in-depth exploration of Jakarta JSON Processing, including understanding

> JSON's syntax, its applications in web services, and the programming models for manipulating JSON data, readers are encouraged to visit the Jakarta EE tutorial. This tutorial offers comprehensive guidance on both the object and streaming models for JSON data handling, suitable for beginners and advanced users alike. Learn more at the Jakarta EE Documentation on JSON Processing.

Jakarta JSON Binding

Jakarta JSON Binding (JSON-B) is a specification within the Jakarta EE platform that provides a high-level API for converting (binding) Java objects to and from JSON documents. It sits on top of Jakarta JSON Processing (JSON-P) and offers a more convenient way to work with JSON data than manually parsing and generating JSON using JSON-P's lower-level APIs. JSON-B is designed to simplify the task of serializing Java objects into JSON and deserializing JSON into Java objects, making it an essential tool for developing modern Java enterprise applications that interact with web services, RESTful APIs, and microservices.

Key Features of Jakarta JSON Binding

- **Automatic Binding**: JSON-B can automatically bind Java objects to JSON and vice versa without requiring manual parsing, significantly simplifying code and reducing boilerplate.
- **Customization**: It provides annotations that allow developers to customize the serialization and deserialization process, such as changing property names in JSON, including or excluding specific fields, and handling custom data types.
- **Support for Java Generics**: JSON-B can handle complex objects, including those that use Java Generics, ensuring type safety during the binding process. Integration with JSON-P: JSON-B is built on top of JSON-P and can seamlessly integrate with it, allowing developers to mix high-level object binding with low-level JSON processing as needed.

> If you are interested in diving deeper into the specifics of JSON Binding, We highly recommend you to visit the Jakarta EE tutorial. It provides detailed insights into how JSON Binding works, including the processes for converting Java objects to JSON and vice versa. This knowledge is crucial for effectively managing JSON data in Java-based enterprise applications. Learn more at the Jakarta EE Documentation on JSON Binding.

Jakarta RESTful Web Services

Jakarta RESTful Web Services(Jakarta REST) is a specification for creating web services according to the Representational State Transfer (REST) architectural pattern. It provides annotations to define resources and operations, making it straightforward to develop APIs for web applications.

Key Features of Jakarta RESTful Web Services

- **Annotation-driven Development**: Simplifies the creation of web services by using annotations to define resources, HTTP methods, and response types.
- **Flexible Data Format Support**: While JSON is commonly used, JAX-RS supports a variety of data formats, providing flexibility in API design.

- **Client API**: Includes a client API for creating HTTP requests to RESTful services, facilitating communication between microservices.

The Jakarta EE 10 Core Profile's focus on these specifications underscores its aim to provide a lightweight, yet comprehensive platform for developing modern Java applications suited for microservices architectures and cloud-native environments.

For those looking to master developing RESTful Web Services, we strongly encourage you to explore Jakarta RESTful Web Services Tutorial. This comprehensive tutorial offers a deep dive into the Jakarta RESTful Web Services specification, demonstrating how to create, deploy, and manage RESTful services efficiently.

Managing Component Dependencies

Jakarta Annotations and CDI plays a central role in integrating different Jakarta EE specifications, such as Jakarta Persistence API (formerly JPA) for database operations and Jakarta RESTful Web Services (formerly JAX-RS) for web services. Let's now enhance the product microservices we developed previously.

Jakarta Annotations is used for defining RESTful services and injecting dependencies. For instance, in our product microservices, we can update the `Product` and `ProductRepository` class to include annotations that facilitate entity management and dependency injection:

Entity class

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

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.GeneratedValue;
import jakarta.validation.constraints.NotNull;

@Entity
@Table(name = "Product")
@NamedQuery(name = "Product.findAllProducts", query = "SELECT p FROM Product p")
@NamedQuery(name = "Product.findProductById", query = "SELECT p FROM Product p WHERE
p.id = :id")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Product {

    @Id
    @GeneratedValue
    private Long id;

    @NotNull
```

```
    private String name;

    @NotNull
    private String description;

    @NotNull
    private Double price;
}
```

Explanation:

- `@Entity` and `@Table(name = "Product")`: These annotations declare the class as a Jakarta Persistence entity and map it to a database table named "Product".
- `@Id` and `@GeneratedValue`: These annotations denote the `id` field as the primary key of the entity and indicate that its value should be generated automatically.
- `@NotNull`: This annotation from Jakarta Bean Validation ensures that the `name`, `description`, and `price` fields cannot be `null`, enforcing data integrity at the application level.
- `@NamedQuery`: These annotations define Jakarta Persistence API named queries for common operations, such as retrieving all products or finding a product by its ids. These can be used throughout the application to interact with the database in a consistent manner.
- `@Data`, `@AllArgsConstructor`, and `@NoArgsConstructor`: These annotations from Project Lombok automatically generate boilerplate code such as getters, setters, a no-arguments constructor, and an all-arguments constructor. This keeps the entity class concise and focused on its fields and annotations related to Jakarta Persistence.

Repository class

The `ProductRepository` class serves as a bridge between the application's business logic layer and the database, performing CRUD (Create, Read, Update, Delete) operations on `Product` entities. It exemplifies the separation of concerns, a fundamental principle in enterprise Java applications, by cleanly segregating the data access logic from the business logic.

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

import java.util.List;

import io.microprofile.tutorial.store.product.entity.Product;
import jakarta.enterprise.context.RequestScoped;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

@RequestScoped
public class ProductRepository {

   @PersistenceContext(unitName = "product-unit")
   private EntityManager em;
```

```
    private List<Product> products = new ArrayList<>();

    public ProductRepository() {
         // Initialize the list with some sample products
         products.add(new Product(1L, "iPhone", "Apple iPhone 15", 999.99));
         products.add(new Product(2L, "MacBook", "Apple MacBook Air", 1299.0));
    }

    public void createProduct(Product product) {
        em.persist(product);
    }

    public Product updateProduct(Product product) {
        return em.merge(product);
    }

    public void deleteProduct(Product product) {
        em.remove(product);
    }

    public List<Product> findAllProducts() {
        return em.createNamedQuery("Product.findAllProducts",
        Product.class).getResultList();
    }

    public Product findProductById(Long id) {
        return em.find(Product.class, id);
    }

    public List<Product> findProduct(String name, String description, Double price) {
        return em.createNamedQuery("Event.findProduct", Product.class)
            .setParameter("name", name)
            .setParameter("description", description)
            .setParameter("price", price).getResultList();
    }

}
```

Explanation:

- `ProductRepository`: This class utilizes Jakarta Persistence API (JPA) for database operations, encapsulating the CRUD (Create, Read, Update, Delete) operations along with methods to find products by various criteria.
- `@RequestScoped`: This CDI annotation for ProductRepository class indicates that an instance of this class is created for each HTTP request to ensure that database operations are handled within the context of a single request.
- `@PersistenceContext`: This annotation injects an entity manager instance, em, specifying the persistence unit product-unit. The entity manager is the primary JPA interface for database interactions.

- The methods createProduct(), updateProduct(), deleteProduct(), findAllProducts(), and findProductById() methods define CRUD operations that might be performed by the repository. These methods utilize the EntityManager instance to persist, merge, remove, and query for product entities.
- The EntityManager is responsible for managing the persistence context and performing CRUD operations on the entities.

The `ProductRepository` serves as a foundational example for developers to understand how to construct a data access layer in a MicroProfile application, emphasizing the significance of CDI in managing component lifecycles and dependencies, as well as showcasing the application of Jakarta Persistence for Object Relational Mapping(ORM) based data access.

Lifecycle Management of Beans in Jakarta EE

CDI defines several built-in scopes to manage the lifecycle of beans, each corresponding to a specific context within the application. When a bean is needed, the CDI container automatically creates it within its defined scope, manages its lifecycle, and destroys it when the context ends. This process is largely transparent to the developer, simplifying development.

To learn more about using built-in scopes in CDI for the lifecycle management of beans, We highly recommend visiting the Using Scopes section of the Jakarta EE Tutorial. This resource provides valuable insights into each scope and how to use them effectively in your applications.

Handling HTTP Methods and Resources

Jakarta RESTful Web Services annotations are utilized to define endpoints for the web services, facilitating the creation and management of RESTful APIs. The `ProductResource` class demonstrates this:

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


import java.util.List;


import io.microprofile.tutorial.store.product.entity.Product;
import io.microprofile.tutorial.store.product.repository.ProductRepository;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;

@Path("/products")
@ApplicationScoped
public class ProductResource {

    private static final Logger LOGGER =
```

```
Logger.getLogger(ProductResource.class.getName());

    // ...

    @Inject
    private ProductRepository productRepository;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAllProducts() {
        LOGGER.info("Fetching all products");
        return Response.ok(products).build();
    }

    // Additional endpoint methods
}
```

The `@ApplicaitonScoped` is an CDI annotation that specifies that the bean is application-scoped, meaning there will be a single instance of `ProductResource` for the entire application, which promotes better resource utilization and performance.

The `@Inject` annotation is commonly used in CDI to inject instances into the application classes without needing to do manual lookups or new instance creations. For example, When `ProductResource` needed a repository to fetch products from a database, we used `@Inject` to incorporate that repository seamlessly.

`@Path` and `@GET`: Defines the URI path and HTTP method for accessing the `getProducts` endpoint.

Defining RESTful APIs

When creating a REST API, you typically start by defining the resources that your API will expose. A unique URI identifies each resource. You then define the operations that can be performed on each resource. These operations are typically CRUD operations: create, read, update, and delete. Let us now create a RESTful API to manage a list of products for a store. This RESTful API allows client applications to access the product stored as resources on the server.

The API is implemented using Jakarta EE and REST architectural style. The API has the following methods:

- `GET /api/products`: Retrieves a list of products
- `POST /api/products`: Creates a new product, the product details are provided as JSON in the request body
- `PUT /api/products`: Updates an existing product, the updated product details are provided as JSON in the request body
- `DELETE /api/products/chapter03`: Deletes a product, the product id is provided in the request URL path

Multiple annotations can be used together in a single method to support multiple media types. For

example, When both `@Consumes(MediaType.APPLICATION_JSON)` and `@Produces(MediaType.APPLICATION_XML)` are used together in a single method, then the method can consume JSON and produce XML.

Table 3-1 shows a list of some of the popular Media types along with their constant fields in `jakarta.ws.rs.core.MediaType` class and corresponding HTTP ContentType:

Media Type	Constant Field	Description
`application/json`	`MediaType.APPLICATION_JSON`	JSON format, used for representing structured data.
`application/xml`	`MediaType.APPLICATION_XML`	XML format, used for representing structured data in XML format.
`text/xml`	`MediaType.TEXT_XML`	XML format, primarily used for XML data that is human-readable.
`text/plain`	`MediaType.TEXT_PLAIN`	Plain text format, used for unstructured text data.
`text/html`	`MediaType.TEXT_HTML`	HTML format, used for markup data that can be rendered by web browsers.
`application/octet-stream`	`MediaType.APPLICATION_OCTET_STREAM`	Binary data stream, used for transmitting files or streaming.
`application/x-www-form-urlencoded`	`MediaType.APPLICATION_FORM_URLENCODED`	Web form format, used for submitting form data in HTTP requests.
`multipart/form-data`	`MediaType.MULTIPART_FORM_DATA`	Multipart format, used for uploading files through web forms.
`application/vnd.api+json`	Custom	JSON API format, a specification for how clients should request and modify resources.
`application/hal+json`	Custom	Hypertext Application Language (HAL) JSON format, used for linking between resources in APIs.

Implementing REST APIs for Managing Products Data

After having successfully performed the development and testing of the GET method of `ProductResource` to fetch the list of product resources. Let's now call the create, update and delete methods for our Products REST API. For this you only need to add additional methods of our `ProductResource` class.

Creating a Product

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Transactional
public Response createProduct(Product product) {
   System.out.println("Creating product");
   productRepository.createProduct(product);
   return Response.status(Response.Status.CREATED)
         .entity("New product created").build();
}
```

Explanation:

The `createProduct()` method is annotated with `@POST`, which means it can be invoked via an HTTP POST request. The `@Consumes(MediaType.APPLICATION_JSON)` annotation says it will consume JSON data. This method takes a single parameter, which is of type `Product`. This parameter will be populated with the data sent in the HTTP POST request. The method creates a new Product object and adds it to the list of products. Finally, the method returns a `Response` object with a status code of 201 (Created) and a message indicating that a new product has been created.

Verifying the POST request

You can use a REST client such as Postman or the cURL command line utility to test the HTTP methods (including PUT, POST, DELETE). To verify the POST request, you can use the following cURL command. This sends a JSON object representing a new product to your microservice.

Command:

```
$ curl -H 'Content-Type: application/json' -d '{ "id": "3", "name":"iPhone 14",
"description":"Apple iPhone 14", "price":"799.99"}' -X POST http://localhost:9080/mp-
ecomm-store/api/products
```

Output:

```
New product created
```

This command specifies the content type as JSON and sends a data payload representing a product with an ID of 3, the name "iPhone 14", a description of "Apple iPhone 14", and a price of 799.99. The `-X POST` parameter indicates that this is a POST request. Upon successful execution, your service should process this data and add the new product to the database.

Next you can verify the addition of the new product, by calling the GET method using cURL or browser as described previously to list all products. This request should now return an updated list of products, including the newly added product.

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

Updating a Product

Updating existing product information is a common operation for RESTful services managing a catalog of items. The `PUT` request method is designed for these scenarios, allowing you to modify an existing product's details. The code snippet below demonstrates updating the product:

```
@PUT
@Consumes(MediaType.APPLICATION_JSON)
@Transactional
public Response updateProduct(Product product) {
   // Update an existing product
   Response response;
   System.out.println("Updating product");
   Product updatedProduct = productRepository.updateProduct(product);

   if (updatedProduct != null) {
       response = Response.status(Response.Status.OK)
                  .entity("Product updated").build();
   } else {
       response = Response.status(Response.Status.NOT_FOUND)
                 .entity("Product not found").build();
   }
   return response;
}
```

Explanation:

- The `@PUT` annotation defines that the method `updateProduct()` can be invoked via an HTTP PUT request.
- As in the POST method, the `@Consumes(MediaType.APPLICATION_JSON)` annotation specifies the method will consume JSON data. This method takes a single parameter, which is of type `Product`. This parameter will be populated with the data sent in the HTTP PUT request. The method updates the product with the same id as the one sent in the request.
- If a product with the same id is not found, the method returns a 404 (Not Found) error. Finally, the method returns a `Response` object with a status code of 204 (No Content) and a message indicating that an existing product has been updated.

Verifying the PUT request

To test the PUT request, you can use the following cURL command.

```
$ curl -H 'Content-Type: application/json' -d '{ "id": "3","name":"iPhone14",
"description":"Apple iPhone 14", "price":"749"}' -X PUT  http://localhost:5050/mp-
ecomm-store/api/products
```

Next you can verify the updation of the new product, by calling the GET method using cURL or browser as described previously to list all products.

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

Deleting a Product

```
@DELETE
@Path("products/{id}")
public Response deleteProduct(@PathParam("id") Long id) {
     // Delete a product
     Response response;
     System.out.println("Deleting product with id: " + id);
     Product product = productRepository.findProductById(id);
     if (product != null) {
         productRepository.deleteProduct(product);
         response = Response.status(Response.Status.OK)
                 .entity("Product deleted").build();
     } else {
        response = Response.status(Response.Status.NOT_FOUND)
                .entity("Product not found").build();
     }
     return response;
}
```

Explanation:

- The `@DELETE` annotation defines that the method `deleteProduct()` can be invoked via an HTTP DELETE request.
- The @Path annotation specifies the id path parameter that will be used to identify which product to delete. This method takes a single parameter of type `Long` and is annotated with the `@PathParam` annotation. This parameter will be populated with the `id` path parameter from the HTTP DELETE request.
- The method deletes the product with the same `id` as the one sent in the request. If a product with the same id is not found, the method returns a 404 (Not Found) error. Finally, the method returns a Response object with a status code of 204 (No Content) and a message indicating that an existing product has been deleted.

The code demonstrated in this chapter is not production quality. It was highly simplified to explain to you the fundamental principles of the REST API. In the upcoming chapters, you will be further building upon this code. By implementing many features from the latest MicroProfile and Jakarta EE standards, you would be making it more a robust microservice that is also cloud-ready. You would also learn to containerize, scale, deploy and manage this application.

Summary

This chapter has laid a solid foundation on the Jakarta EE 10 Core Profile, emphasizing its crucial role in the development of microservices using MicroProfile. By delving into key specifications and through practical implementation examples, you have been equipped with the necessary knowledge to utilize the Jakarta EE 10 Core Profile's features for creating scalable, resilient, and portable cloud-native applications.

Additionally, this chapter guided you through the creation of RESTful web services using Jakarta EE Restful Web Services APIs, providing an overview of REST (Representational State Transfer), it aimed to familiarize you with the basics of REST, enabling you to create and deploy a RESTful web service independently.

As we move forward, the next chapter will delve deeper into the REST architectural pattern, exploring standard conventions, design considerations, and best practices. It will cover many advanced concepts essential for building RESTful web services tailored for cloud-native and microservices-based applications, preparing you for more sophisticated aspects of modern application development.

MicroProfile OpenAPI

Introduction

In the previous chapter, we saw how RESTful APIs facilitate language-agnostic access to web services from diverse environments. However, a clear and comprehensive contract is required to ensure seamless integration between clients and services. This need for a well-defined API contract has led to the adoption of the OpenAPI specification. This chapter will explore the primary features of MicroProfile OpenAPI, demonstrate how to integrate it into your MicroProfile applications, and show you how to annotate your RESTful services to produce rich documentation that adheres to the OpenAPI specification. Furthermore, we will introduce the OpenAPI UI, a visual interface allowing developers and stakeholders to interact with and visualize the documented APIs, enhancing understanding and facilitating integration.

Topics to be covered:

- Introduction to MicroProfile OpenAPI
- API Specification using MicroProfile Open API
- Generating API Documentation
- Documenting Authentication and Authorization Requirements
- Exploring the APIs using Swagger UI

OpenAPI Specification

The Open API Specification (OAS), formerly Swagger specification, is a technical specification that allows REST API providers to describe and publish their APIs using a format that various tools can consume. It defines a standard, language-agnostic interface to RESTful APIs, making it easy for third-party tools to generate documentation, client SDKs, and a range of tools that promote the seamless consumption of RESTful APIs.

The OpenAPI Initiative, a consortium of industry experts committed to standardizing how to describe REST APIs, maintains the OpenAPI Specification. It is a community-driven initiative, and many large organizations use it, including Google, Microsoft, and Amazon.

The OpenAPI specification enables creation of a well-defined, clear and comprehensive API contract. It provides a standardized way to describe the API's structure, expected requests and responses, and authentication mechanisms, making it easier to develop, test, and maintain RESTful APIs.

Introduction to MicroProfile OpenAPI

The MicroProfile OpenAPI specification builds upon the widely recognized OpenAPI Specification (OAS) and leverages annotations from the Jakarta Restful Web Services specification. The primary

focus of MicroProfile OpenAPI is on defining REST APIs that utilize JSON within the context of HTTP.

The specification aims to provide a uniform way of describing APIs so that they are both human-readable and machine-readable. It facilitates the creation of APIs that are consistent, well-documented, and easily consumable by both humans and machines.

Capabilities of MicroProfile OpenAPI Specification

MicroProfile OpenAPI provides a suite of Java APIs that allows developers to define and generate API specifications that adhere to OpenAPI v3 standards. As a result, it simplifies the process of designing, documenting, and publishing RESTful APIs for developers.

Developers can quickly generate documentation for their microservices using MicroProfile OpenAPI. The documentation includes information on what services are provided, how to invoke them, and what data types are used. It generates comprehensive metadata about services, ensuring interoperability across diverse platforms and tools. Also, documentation can generate client code to access the web services.

The OpenAPI Specification fuels a rich ecosystem of tools that automate and support. This specification streamlines the creation of OpenAPI documentation for RESTful services using a unified approach. It generates comprehensive metadata about services, ensuring interoperability across diverse platforms and tools:

- **API Documentation Generation**: Intuitive interactive documentation portals emerge directly from the specification.
- **Client SDK Creation**: Client libraries in various languages can be automatically generated.
- **API Testing**: Testing frameworks can leverage the specification to design robust tests.
- **API Mocking**: Simplifies mocking APIs for testing and development purposes.

Generating OpenAPI documents

There are multiple ways in which you can generate OpenAPI documents. The most common way is to use annotations. This only requires augmenting your Jakarta Restful Web Services annotations with OpenAPI annotations.

Besides annotations, a predefined OpenAPI document may be provided in either YAML or JSON format. This so-called static model will be merged with the model generated by scanning for Jakarta REST endpoints and the combined result will be made available to clients. However, the annotation-based approach is recommended as it is more maintainable and easier to understand. Finally, you can filter out the resources you do not want to document using configuration.

Using MicroProfile Open API in your project

To document Jakarta RESTful Web Services using MicroProfile OpenAPI, we need to annotate the resource classes and methods with the OpenAPI annotation.

To use MicroProfile OpenAPI in your project, you need to add the following maven coordinates to

your project:

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

Below is an illustrative example of how you might annotate a method in the `ProductResource` class to achieve this documentation using MicroProfile OpenAPI annotations:

```
import org.eclipse.microprofile.openapi.annotations.Operation;
import org.eclipse.microprofile.openapi.annotations.media.Content;
import org.eclipse.microprofile.openapi.annotations.media.Schema;
import org.eclipse.microprofile.openapi.annotations.responses.APIResponse;
import org.eclipse.microprofile.openapi.annotations.responses.APIResponses;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@ApplicationScoped
@Path("/products")
@Tag(name = "Product Resource", description = "CRUD operations for products")
public class ProductResource {

    //...

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(summary = "List all products", description = "Retrieves a list of all
available products")
    @APIResponses(value = {
        @APIResponse(
            responseCode = "200",
            description = "Successful, list of products found",
            content = @Content(mediaType = "application/json",
                    schema = @Schema(implementation = Product.class))
        ),
        @APIResponse(
            responseCode = "400",
            description = "Unsuccessful, no products found",
            content = @Content(mediaType = "application/json")
        )
    })
    public List<Product> getAllProducts() {
        // Method implementation
    }
```

```
}
```

Explanation:

- `@Operation`: Provides a summary and description for the `getProducts()` method.
- `@APIResponse`: Describes the possible responses from the `getProducts()` operation. In this case, a successful response (HTTP 200) is described, indicating that the method returns an array of Product entities.
- `@Schema`: Specifies the schema of the response content. Here, it is used to indicate that the method returns an array of Product objects.

These annotations enrich the `ProductResource` class with metadata necessary for generating comprehensive and descriptive OpenAPI documentation automatically.

We have also annotated the `getProducts()` method with the @APIResponse annotation to document the successful response from the operation. The `responseCode` field is used to specify the status code of the response, and the `description` field is used to provide a brief description of the response. There are two possible responses – a successful response containing a list of produdts with a 200 status code, and an unsuccessful response with a 400 status code, if no products are found. The content field is used to specify the schema of the response content. In this example, the response content is a list of `Product`s.

Finally, we need to add the following property to the src/main/resources/META-INF/microprofile-config.properties file:

```
mp.openapi.scan=true
```

This property tells MicroProfile OpenAPI to scan our classes for annotations and generate API documentation for them.

Now that we have configured MicroProfile OpenAPI, we can build and run our application.

How to view the generated documentation

To view the generated documentation, we can use the OpenAPI UI tool. The Open API UI tool is a web-based tool that can be used to view the documentation for a REST API.

The OpenAPI UI tool can be accessed at the following URL:

```
http://localhost:<port>/openapi/
```

Replace `<port>` with the actual port used by your runtime, for e.g. 9080 which is the default port at Open Liberty server.

The `/openapi` endpoint is used to get information about the OpenAPI specification generated from the comments in the source code annotations. It returns information in YAML format.

When we access the `http://localhost:5050/openapi` URL, we should see the API documentation that was generated by MicroProfile OpenAPI:

```
openapi: 3.0.3
info:
  title: Generated API
  version: "1.0"
servers:
- url: http://localhost:9080/catalog
paths:
  /api/products:
    get:
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Product'
    put:
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Product'
      responses:
        "200":
          description: OK
    post:
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Product'
      responses:
        "200":
          description: OK
  /api/products/products/{id}:
    delete:
      parameters:
      - name: id
        in: path
        required: true
        schema:
          format: int64
          type: integer
      responses:
        "200":
```

```
          description: OK
  /api/products/{id}:
    get:
      parameters:
      - name: id
        in: path
        required: true
        schema:
          format: int64
          type: integer
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Product'
components:
  schemas:
    Product:
      required:
      - name
      - description
      - price
      type: object
      properties:
        id:
          format: int64
          type: integer
        name:
          type: string
        description:
          type: string
        price:
          format: double
          type: number
```

As we can see, MicroProfile OpenAPI has generated API documentation for our resource class. We can use this documentation to learn about the API and how to use it.

MicroProfile OpenAPI allows developers to produce these specifications directly from their codebase, leveraging annotations and/or providing OpenAPI documents statically. This direct generation ensures that the API documentation is always up to date with the code.

Exploring the APIs using Swagger UI

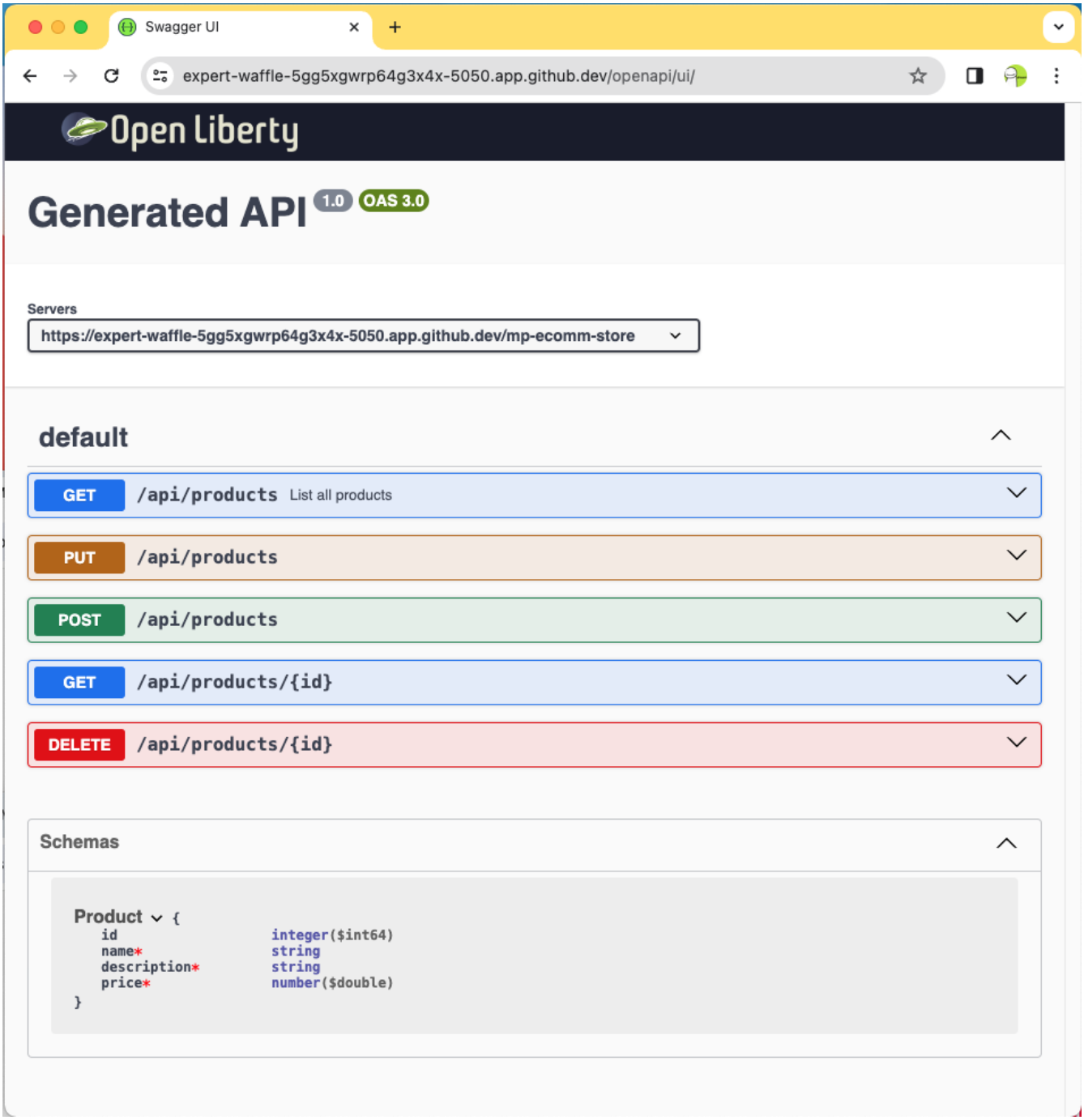
To open Swagger UI for the API documentation generated using MicroProfile OpenAPI, you will need to deploy your application to a server that supports MicroProfile, such as Open Liberty, WildFly, Quarkus, or Payara Micro. These servers automatically generate the OpenAPI

documentation for your RESTful services based on the annotations in your code.

Next, visit the following URL to launch the Swagger UI:

```
http://localhost:9080/openapi/ui
```

Swagger UI is then used to render this documentation in a user-friendly web interface. Below is the screenshot of swagger UI for the Product REST Resource.



Swagger UI 4. Swagger UI

Annotations

The MicroProfile OpenAPI annotations can be used to document any Jakarta Restful Web Services

resource. The annotations can also be used in conjunction with other Jakarta Restful Webservices annotations, such as @Path and @Produces. The most common annotations that are used to document RESTful web services are list in Table 4-1.

Annotations	Details
@OpenAPIDefinition	Provides metadata about the entire API. It can include information such as the title, description, version, terms of service, and contact information.
@Info	Used inside @OpenAPIDefinition to provide API metadata like title, version, description.
@Contact	Specifies contact information for the API, used within @Info.
@License	Defines the license information for the API, also used within @Info.
@Operation	Describes a single API operation on a resource.
@APIResponse	It is used to document a response from an operation.
@APIResponses	A container for multiple @APIResponse annotations, allowing documentation of different responses for a single API operation.
@RequestBody	Describes the request body of an HTTP request, specifying the content of the body and whether it is required.
@Schema	Provides schema details for a response or request body, specifying the data type, format, and constraints.
@Parameter	Provides information on parameters to the operation, including query parameters, header parameters, and path parameters.
@Tag	Adds metadata to a single tag that is used by the Operation. It helps in categorizing operations by resources or any other qualifier.
@Content	Specifies the media type and schema of the operation's request or response body.
@Components	Allows the definition of reusable components such as schemas, responses, parameters, and more, which can be referenced by other annotations.
@SecurityRequirement	Specifies a security requirement for an operation, referencing security schemes defined in the @Components.
@ExternalDocumentation	Provides additional external documentation for an API or operation.
@Callback	Specifies a callback URL for an asynchronous operation.
@Callbacks	Specifies multiple `@Callback` annotations.
@Server	Describes a server that hosts the API, specifying URL and description, which can be global or specific to operations or paths

All of these annotations are defined in the org.eclipse.microprofile.openapi.annotations package.

Summary

By integrating the MicroProfile OpenAPI, developers can generate detailed, OpenAPI-compliant documentation automatically, fostering better understanding and interaction among services. By annotating `ProductResource` class, we generated API documentation as per Open API specification. This will ensure the services are readily discoverable, understandable, and usable, thereby accelerating development cycles and fostering a more robust and collaborative developer ecosystem.

MicroProfile Configuration

This chapter focuses on MicroProfile Configuration, a key feature that allows developers to externalize configuration properties from their code. You can adapt configuration parameters to different environments (development, testing, production) without altering the core code. It provides flexibility and adaptability for microservices in different environments.

Topics to be covered

- Understanding MicroProfile Configuration
- Different environments required for Microservices development
- Working with Various Configuration Sources
- Key capabilities of MicroProfile Configuration
- Implementing Configuration Properties
- Creating a Custom Configuration Source
- Dynamic Updates and Handling Configuration Change Events
- Managing Configuration for Different Environments
- Securing Configuration and Best Practices

Understanding MicroProfile Configuration

MicroProfile Configuration is a specification that allows developers to inject configuration values into applications. The MicroProfile Configuration APIs will enable developers to externalize configuration and access it from within your application. By separating configuration data (like database URLs, API credentials, feature flags) from the codebase, you make it easier to modify these settings without recompiling and redeploying the application.

For instance, with MicroProfile Config, you can configure connection settings for a database enhancing flexibility and adaptability across different environments in our MicroProfile e-commerce application. You can update configurations seamlessly, sometimes even while the application is running (for dynamic config sources), minimizing downtime and streamlining deployment processes. This is essential for microservices that may run in diverse setups.

Different environments required for Microservices development

When developing microservices, it's essential to establish various environments to accommodate different stages of the development lifecycle. Each environment serves distinct purposes, ensuring the application is thoroughly tested, secure, and efficient before its deployment to production. Below are the critical environments typically set up for developing microservices:

- **Development Environment**—Developers write new code, implement features, and perform basic unit testing in this environment, which is where the initial development occurs. This

environment is usually configured to use local or development databases with dummy data for testing. The logging level used in this environment is generally verbose for debugging purposes.

- **Testing or QA Environment**—This environment is dedicated to rigorous testing, including automated tests, integration tests, and manual testing by QA engineers to identify bugs or issues. Configuration here mirrors production settings as closely as possible and connects to a testing database. For error tracing, detailed logging may be enabled in this environment.
- **Staging Environment**—This is a production-like environment for final testing of the changes before deployment to production. It ensures that your microservices perform as expected under production conditions. This environment is configured with settings identical to the production environment. It typically uses a copy of production data that is sanitized of sensitive data.
- **Production Environment**—This is the live environment where the microservice is fully deployed and accessible to end-users. It's optimized for security, performance, and reliability and configured to access actual user data with all security features fully enabled. Performance monitoring tools are also set up here to ensure smooth operations.

Using the above set of environments, development teams can streamline the development process, enhance quality, and ensure the microservices are robust and ready for production use. Your development team may also require additional environments for specific needs like automation, penetration testing, and stress testing, depending on the unique needs of the microservices.

Working with Various Configuration Sources

MicroProfile Config allows applications to retrieve configuration properties from a variety of sources. By default, MicroProfile Config includes various built-in configuration sources, but you can also define custom sources. Below we discuss how to work with these various configuration sources.

Built-in Configuration Sources

MicroProfile Config defines default configuration sources that are automatically enabled:

System Properties: Configuration values defined as system properties can be accessed by MicroProfile Config. These properties can be set at runtime using the `-D` flag when starting the JVM.

Environment Variables: Environment variables available in the system can be used as configuration sources. They are useful for setting configuration properties external to the application, especially in containerized environments.

MicroProfile Config Properties File: A properties file named *microprofile-config.properties* can be placed in the *META-INF* directory of your application. This file is particularly useful for setting default configuration values that ship with the application.

Types of Configuration Sources

A **static configuration source** is the one where the data does not change once the application has started. Examples include the microprofile-config.properties file and most custom implementations that read from a database or a service at startup.

On the other hand, **a dynamic configuration source** is one that can change its data at runtime. System properties and some custom implementations that periodically check for changes in a remote configuration service are examples of dynamic sources.

MicroProfile Config allows applications to read from these dynamic sources as easily as from static ones. However, whether a configuration source supports dynamic behavior depends on its implementation.

Key capabilities of MicroProfile Configuration

The MicroProfile Configuration specifications offer a set of APIs that enable you to handle your application's configuration efficiently. They allow you to easily manage and customize your application's configurations, making it a valuable tool for developers.

The MicroProfile Configuration APIs provide the following capabilities for managing the configuration settings of your application:

- It allows reading configuration values.
- It allows applications to retrieve configuration values reliably, supporting various sources, such as property files, system properties, environment variables, and more.

The MicroProfile Configuration API provides several classes, allowing easy integration of configuration values. Below is the list of key classes and interfaces included in the MicroProfile Configuration API:

- `Config` - the class that is the main entry point to the configuration API and provides access to configuration data. The Config class provides static methods that can be used to access configuration properties.
- `ConfigProvider` - a utility class for getting the Config instance. It allows retrieving the static instance of the Config object.
- `ConfigBuilder` - An interface used to create a Config instance manually. It can add default sources, converters, and configuration sources.
- `ConfigSource` - This class represents a source of configuration values. It reads configuration data from a specific source, such as system properties, environment variables, files, or data stores.
- `Property` - It represents a key/value pair in the configuration data.
- `Converter<T>` - This interface implements custom converters that convert configuration values from String to any desired type.

These classes and interfaces provide a robust configuration mechanism that is easy to use and extend. Developers can leverage these APIs to externalize configuration from their applications, making them more flexible and more accessible to run in different environments.

Implementing Configuration Properties

The Config API allows you to define configuration properties in many ways, including property files, environment variables, and system properties. To use the Config API, we'll need to include the

following dependency in our *pom.xml* file:

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

For Gradle, modify your *build.gradle* file with the following dependency:

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

Let's now modify the `getProducts()` method to return a `jakarta.ws.rs.core.Response` object instead of a list of Product entities directly, we can utilize the `Response` class to build our response. This approach allows for a more standardized and flexible API response handling, including the ability to set HTTP status codes and headers.

Lets create a configuration file with the name *microprofile-config.properties* and the content as below:

```
# microprofile-config.properties
product.maintenanceMode=false
```

This configuration file should be placed in the *src/main/resources/META-INF/* directory of your application.

Reading Configuration Properties

Next inject this configuration value to a private variable in the ProductResource and consume this within all the operations of this service.

MicroProfile Config will automatically detect and use the properties defined in this file, allowing you to externalize configuration and easily adjust the behavior of your application based on the environment in which it is deployed.

Below is the updated `ProductResource` class and `getProducts()` method:

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

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

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.transaction.Transactional;
import jakarta.ws.rs.*;
```

```
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.eclipse.microprofile.openapi.annotations.Operation;
import org.eclipse.microprofile.openapi.annotations.media.Content;
import org.eclipse.microprofile.openapi.annotations.media.Schema;

import org.eclipse.microprofile.openapi.annotations.responses.APIResponse;
import org.eclipse.microprofile.openapi.annotations.responses.APIResponses;

import java.util.List;

@Path("/products")
@ApplicationScoped
public class ProductResource {

   @Inject
   @ConfigProperty(name="product.maintenanceMode", defaultValue="false")
   private boolean maintenanceMode;

   @Inject
   private ProductRepository productRepository;

   @GET
   @Produces(MediaType.APPLICATION_JSON)
   @Transactional

   // OpenAPI code
   // ...

   public Response getProducts() {

       List<Product> products = productRepository.findAllProducts();

       // If in maintenance mode, return Service Unavailable status
       if (maintenanceMode) {
          return Response
                  .status(Response.Status.SERVICE_UNAVAILABLE)
                  .entity("The product catalog service is currently in maintenance
mode. Please try again later.")
                  .build();

       // If products found, return products and OK status
       } else if (products != null && !products.isEmpty()) {
           return Response
                   .status(Response.Status.OK)
                   .entity(products).build();

      // If products not found, return Not Found status and message
      } else {
```

```
            return Response
                    .status(Response.Status.NOT_FOUND)
                    .entity("No products found")
                    .build();
        }
    }
}
```

Explanation:

- `@Inject`: This CDI annotation enables dependency injection. It tells the container to inject an instance of a specified bean at runtime. As we have learnt previously, dependency injection enables loose coupling between classes and their dependencies, making the code more modular, easier to test, and maintain.
- `@ConfigProperty(name="product.maintenanceMode", defaultValue="false")`: This MicroProfile Config annotation used along with `@Inject` to inject configuration property values into beans. It allows developers to externalize configuration from the application code, making applications more flexible and environment-agnostic. The `name` parameter specifies the key of the configuration property to be injected. In this case, `product.maintenanceMode` is the key for a property that controls whether this service is in maintenance mode. The `defaultValue` provides a default value to be used if the specified configuration property is not found in any of the configured sources. Here, the default value is `false`, indicating that, by default, the service is not in maintenance mode unless explicitly configured otherwise.
- `private boolean maintenanceMode`: This field is set to the value of the `product.maintenanceMode` configuration property. Due to the `defaultValue ="false"`, if the configuration is not specified elsewhere, `maintenanceMode` will be `false`, meaning the service operates normally.
- `private ProductRepository productRepository`: This field is injected with an instance of ProductRepository. This class abstracts the data access operations for products. This injection decouples the class from the specific implementation of the repository, making the code more modular and easier to adapt or replace parts of it in the future.
- The `getProducts()` method retrieves all products from the repository by calling `productRepository.findAllProducts()`, which queries the database to retrieve a list of all available products. Before proceeding to return the list of products, the method checks the `maintenanceMode` flag. If `maintenanceMode` is `true`, the service is currently undergoing maintenance, and thus, it is not appropriate to perform regular operations. The method constructs and returns a Response with a `503 Service Unavailable` HTTP status code, along with a message indicating that the product catalog service is in maintenance mode.
- If the service is not in maintenance mode, then the method checks if the list of retrieved products is not `null` and not empty.
- If products are found, it constructs a Response with a status of `200 OK` and includes the list of products as the response entity. This indicates a successful operation where product data is found and returned.
- If the products list is `null` or empty, indicating no products were found, the method constructs and returns a `Response` with a `404 Not Found` status code and a message stating that no products were found.

When we deploy the application and invoke the `/api/products` endpoint, we should see the list of products as below:

```
[{"description":"Apple iPhone 15 Pro","id":1,"name":"iPhone 15 Pro","price":999.0}]
```

Specifying Default Values for a `ConfigProperty`

For non-critical properties, providing a default value using the defaultValue attribute of the @ConfigProperty annotation ensures that your application has a fallback option. We can specify a default value to be used if the property does not exist as below:

```
public class ProductResource {

   @Inject
   @ConfigProperty(name="product.maintenanceMode", defaultValue="false")
   private boolean maintenanceMode;
   ...
```

In the example above , the `false` default value will be used if the `product.maintenanceMode` property does not exist.

Type Conversion in `ConfigProperty`

`ConfigProperty` also supports type conversion, so we can inject our configuration data into fields of any type:

```
   @Inject
   @ConfigProperty(name="product.maintenanceMode", defaultValue="false")
   private boolean maintenanceMode;
```

In this example, the `product.maintenanceMode` property will be converted to an `Boolean` before it is injected into the `maintenanceMode` field.

Converting Configuration data to a POJO

We can also use the Config API to convert our configuration data to a POJO:

```
import org.eclipse.microprofile.config.inject.ConfigProperty;


public class MyApplication {
    @Inject
    private MaintenanceMessage message;
}
```

```
public class MaintenanceMessage {
    @ConfigProperty(name="product.maintenanceMessage")
    private String message;
}
```

In this example, we're injecting a property named "product.maintenanceMessage" into the message field of our MaintenanceMessage class.

Creating a Custom `ConfigSource`

As we saw, the Config API makes it easy to inject configuration properties into an application. The Config API defines a contract for config implementations. A ConfigSource is used to read configuration data from a particular source. For example, we could create a ConfigSource that reads configuration data from a file.

`ConfigSource` interface has the following methods:

- `String getName()` : Returns the name of the ConfigSource.
- `int getOrdinal()` : Returns the ordinal of the ConfigSource. Ordinals are used to determine the precedence of ConfigSources. A higher ordinal means a higher precedence.
- `Map<String, String> getProperties()` : Returns a map of the properties in this ConfigSource. The keys in the map are the property names, and the values are the property values.
- `getValue(String propertyName)` : Returns the value of the given property. If the property is not found, this method returns null.
- `Set getPropertyNames()` : Returns a Set of the property names in this ConfigSource.

Let's implement a feature in our MicroProfile e-Commerce application to integrate payment gateway configuration dynamically by creating a PaymentServiceConfigSource (a custom ConfigSource) which could fetch API keys and endpoints. This would ensure that payment service configurations are up-to-date and can be changed without redeploying the application.

The following is an implementation of a `ConfigSource` that reads configuration data from a file:

```
package io.microprofile.tutorial.store.payment.config;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import org.eclipse.microprofile.config.spi.ConfigSource;

public class PaymentServiceConfigSource implements ConfigSource{

   private Map<String, String> properties = new HashMap<>();

   public PaymentServiceConfigSource() {
       // Load payment service configurations dynamically
       // This example uses hardcoded values for demonstration
```

```
        properties.put("payment.gateway.apiKey", "secret_api_key");
        properties.put("payment.gateway.endpoint", "https://api.paymentgateway.com");
    }

    @Override
    public Map<String, String> getProperties() {
        return properties;
    }

    @Override
    public String getValue(String propertyName) {
        return properties.get(propertyName);
    }

    @Override
    public String getName() {
        return "PaymentServiceConfigSource";
    }

    @Override
    public int getOrdinal() {
        // Ensuring high priority to override default configurations if necessary
        return 600;
    }

    @Override
    public Set<String> getPropertyNames() {
        // Return the set of all property names available in this config source
        return properties.keySet();}
}
```

The above code snippet demonstrates MicroProfile Config's flexibility in integrating with various external configuration providers. This enables applications to load configurations from sources beyond the default system properties, environment variables, and microprofile-config.properties files. This capability is crucial for modern applications that may need to pull configuration from dynamic sources like cloud services, databases, or custom APIs.

When integrating with external configuration providers, it's essential to consider security aspects, especially when dealing with sensitive configuration data. Use secure communication channels (e.g., HTTPS) to retrieve configuration from external services. Manage access control meticulously to prevent unauthorized access to sensitive configuration. Consider encrypting sensitive configuration values and decrypting them within your ConfigSource or application logic.

Registering a `ConfigSource`

To register a custom `ConfigSource` implementation with MicroProfile Config, you need to include the fully qualified class name of your custom `ConfigSource` in this resource file */META-INF/services/org.eclipse.microprofile.config.spi.ConfigSource*.

This `PaymentService` would be a part of the e-Commerce application, handling payment transactions by utilizing configurations that determine which payment gateway to use and how to authenticate with it. By externalizing these configurations, the e-Commerce platform can easily switch payment providers or update API keys without needing to adjust the codebase, providing flexibility and enhancing security.

Accessing the Configuration Data

First, create a class to represent the payment information sent by clients as below:

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

import java.math.BigDecimal;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class PaymentDetails {
   private String cardNumber;
   private String cardHolderName;
   private String expirationDate; // Format MM/YY
   private String securityCode;
   private BigDecimal amount;
}
```

The `PaymentDetails` class succinctly encapsulates the necessary attributes for processing payments. This class can be used to pass payment details for processing payments, validating card details, and logging transaction information.

Next, implement the `PaymentService` class, which utilizes MicroProfile Config to inject the necessary configurations. It represents a simple service that could call a payment gateway API using the configurations provided by the custom `ConfigSource`.

```
import org.eclipse.microprofile.config.inject.ConfigProperty;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;

@Path("/authorize")
@RequestScoped
```

```
public class PaymentService {

    @Inject
    @ConfigProperty(name = "payment.gateway.apiKey")
    private String apiKey;

    @Inject
    @ConfigProperty(name = "payment.gateway.endpoint")
    private String endpoint;

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response processPayment(PaymentDetails paymentDetails) {
        // Example logic to call the payment gateway API
        System.out.println("Processing payment with details: " +
paymentDetails.toString());
        System.out.println("Calling payment gateway API at: " + endpoint);
        // Assuming a successful payment operation for demonstration purposes
        // Actual implementation would involve calling the payment gateway and
handling the response

        // Dummy response for successful payment processing
        String result = "{\"status\":\"success\", \"message\":\"Payment processed
successfully.\"}";
        return Response.ok(result, MediaType.APPLICATION_JSON).build();
    }
}
```

Explanation:

- `@Path("/authorize")`: Defines the base URI for the RESTful service. This class will handle requests made to URIs that start with `/payment/api/authorize`.
- `@RequestScoped`: Indicates that a new instance of `PaymentService` is created for each HTTP request.
- `@POST`: Specifies that the `processPayment` method will respond to HTTP POST requests, which is appropriate for operations that change server state (in this case, processing a payment).
- `@Consumes(MediaType.APPLICATION_JSON)`: Indicates that the method expects requests to have a payload formatted as JSON, aligning with how payment details might be sent.
- `@Produces(MediaType.APPLICATION_JSON)`: Specifies that the method produces JSON-formatted responses, useful for indicating the result of the payment processing operation.
- `Response processPayment(PaymentDetails paymentDetails)`: The method now returns a Response object, allowing for more flexible HTTP response handling. The PaymentDetails parameter would be a POJO (Plain Old Java Object) representing the payment information sent by the client.

The clients can call to process payments through the e-Commerce application using this RESTful web service endpoint. The actual logic for calling the payment gateway API and handling the

response would be implemented within this method, utilizing the injected configuration properties for authentication and endpoint URL.

`ConfigSources` are hierarchical, which means that we can override properties from one `ConfigSource` with another `ConfigSource`. For example, we could create a `ConfigSource` that reads configuration data from a file, and another `ConfigSource` that reads configuration data from system properties. The system properties would take precedence over the file-based `ConfigSource`, which would take precedence over the default `ConfigSource`.

- `Property getProperty(String propertyName)` : Returns information about the given property. If the property is not found, this method returns `null`.

Enabling MicroProfile Config in Open Liberty

Open Liberty requires a `server.xml` file for server configuration. This file should be located at `/src/main/liberty/config/server.xml` within your project. To enable MicroProfile Config, you need to include the `mpConfig` feature in the `<featureManager>` section.

```
<server description="MicroProfile Tutorial Server">
  <featureManager>
    <feature>mpConfig-3.1</feature> <!-- Use the appropriate version -->
    <!-- Include other features as needed -->
  </featureManager>
```

Managing Configuration for Different Environments

Managing configurations for different environments is a crucial aspect of modern application development, especially in microservices architectures where applications may run in development, testing, staging, and production environments with varying configurations. MicroProfile Config provides the flexibility to handle environment-specific configurations efficiently. Here's how to manage configurations for different environments using MicroProfile Config:

Use of Profiles: MicroProfile Config does not explicitly define the concept of profiles for managing environment-specific configurations. However, developers can implement a profile-like mechanism using custom ConfigSource implementations or by organizing configuration properties in a way that differentiates them by environment. For instance, you could prefix configuration keys with the environment name:

- `dev.database.url`
- `test.database.url`
- `prod.database.url`

Then, you can programmatically or conditionally load configurations based on the active environment.

Environment Variables and System Properties: Leveraging environment variables and system properties is a common and effective way to provide environment-specific configurations.

MicroProfile Config automatically includes ConfigSources for both system properties and environment variables, allowing for easy overrides of configurations per environment:

```
String databaseUrl = ConfigProvider.getConfig().getValue("database.url",
String.class);
```

Custom ConfigSources: For more complex scenarios or to integrate with external configuration management systems (e.g., Consul, Etcd, AWS Parameter Store), you can implement custom ConfigSources. These sources can dynamically load configurations based on the environment, either by connecting to external services or by loading environment-specific files:

```
public class MyEnvironmentConfigSource implements ConfigSource {
    // Implementation that loads configurations based on the detected environment
}
```

Configuration Isolation: It's essential to isolate configurations for different environments to prevent accidental leaks of sensitive information (e.g., production database credentials). This can be achieved by using: - separate configuration files for each environment, stored securely and only accessible by the application running in that environment. - Utilizing external secrets management tools to store sensitive configurations, with access controlled by the environment.

CI/CD Integration: Integrate environment-specific configuration management into your CI/CD pipelines. Ensure that the correct configurations are applied automatically as part of the deployment process for each environment.

Strategies for Handling Configuration Change Events

Although direct support for configuration change events is not provided by MicroProfile Config specification itself, applications can implement their mechanisms or use external libraries to achieve this functionality. To implement dynamic updates in your MicroProfile Config usage, you might need to adopt one of the following approaches:

- **Manual Refresh:** Provide a mechanism (e.g., an admin-restricted endpoint) to manually trigger a refresh of the configuration. This approach gives control over when changes are applied but requires manual intervention.
- **Polling**: Implement a scheduler that periodically checks certain configuration properties for changes. This approach is straightforward but might introduce latency between the actual change and its detection.
- **Event-driven Updates**: If your configuration source supports event notifications (for example, a database trigger or a cloud service event), you can set up listeners that update your application's configuration in response to these events.
- **Application-level Event Handling**: Design your application components to subscribe to a custom event bus or notification system. When a configuration change is detected (via polling or custom ConfigSource), publish an event to this bus, triggering subscribed components to update their configurations.

- **Custom Configuration Source**: Develop a custom ConfigSource that includes logic to listen for changes in the underlying configuration store (such as a database, filesystem, or cloud service). This ConfigSource can then notify the application of changes, prompting it to refresh configuration properties.
- **Runtime Extensions**: Some MicroProfile runtimes may offer extensions that support dynamic configuration and change event handling. Check the documentation of your runtime environment for such features and best practices for their usage.
- **Framework/Library Support**: Use a third-party library or framework that extends MicroProfile Config with change event support. These libraries might offer annotations or listener interfaces to react to configuration changes automatically.
- **External Configuration Management Tools**: Utilize configuration management tools or services that offer webhook or messaging functionalities to notify your application of configuration changes. Upon receiving a notification, the application can reload its configuration context.

While MicroProfile Config provides the mechanisms to read from dynamic configuration sources, it does not specify a standard way to listen for changes in configuration properties directly within its API as of version 3.1. Applications need to implement their logic or use additional libraries/frameworks to detect changes in configuration sources and react accordingly.

However, some implementations of MicroProfile Config might offer extensions or additional functionalities to support configuration change events. For example, an application can poll a configuration source at intervals to detect changes or use a notification system that triggers configuration reloads.

Best Practices and Securing Configuration in MicroProfile Config

Here are some recommended practices for using MicroProfile Config:

Graceful Configuration Reloads: Ensure that your application can gracefully handle configuration reloads, especially in critical components that depend on configuration properties for their operation.

Minimize Performance Impact: Design your dynamic configuration update mechanism to minimize performance impacts, especially if using polling mechanisms.

Secure Configuration Management: When implementing custom solutions for dynamic configuration, pay attention to security aspects, particularly if configurations include sensitive information. Securing sensitive configuration properties is crucial for maintaining the security and integrity of applications.

Encrypt Sensitive Configuration Values: Sensitive information, such as passwords, tokens, and API keys, should be encrypted in the configuration source. Decryption can be handled programmatically within the application or through integration with external secrets management systems.

Use Environment-Specific Configuration Files: Separate configuration files for different environments (development, testing, production) can help minimize the risk of exposing sensitive data. For instance, development configurations might use placeholder values, whereas production configurations access secrets from a secure vault or environment variables.

Leverage External Secrets Management: Integrating with external secrets management tools (like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault) ensures that sensitive configurations are stored securely and accessed dynamically at runtime. These tools provide mechanisms to control access to secrets and often include auditing capabilities.

Use Environment Variables for Sensitive Values: Environment variables can be a secure way to provide configuration to applications, especially for containerized or cloud-native applications. This approach leverages the underlying platform's security model to protect sensitive information.

Implement Access Control: Ensure that only authorized personnel have access to configuration files, especially those containing sensitive information. Use file permissions, access control lists (ACLs), or similar mechanisms provided by the operating system or hosting environment.

Audit and Monitor Configuration Access: Regularly audit access to configuration files and monitor for unauthorized access attempts. This can help detect potential security breaches and ensure that only authorized changes are made to the configuration.

Configuration Validation: Validate configuration data at startup to ensure that it meets the application's expected format and values. This step can prevent configuration errors and detect tampering or unauthorized changes.

Keep Configuration Data Updated: Regularly review and update configuration data to ensure that it reflects the current operational and security needs. Remove unused properties and update secrets periodically to reduce the risk of compromise.

Summary

Dynamic configuration management is essential for modern applications, providing the flexibility to adapt to changing environments without downtime. Although MicroProfile Config as of version 3.1 does not define a standard for handling configuration change events directly, applications can still achieve this by combining MicroProfile Config with custom logic or additional tools designed for dynamic configuration management. Always consult the documentation of your MicroProfile implementation to learn about supported features and extensions related to dynamic configuration and change events.

While the MicroProfile Config specification provides a powerful and flexible framework for configuration management, handling dynamic updates and configuration change events may require additional custom development or the use of external tools. By considering the strategies mentioned above, developers can effectively manage configuration changes, ensuring their microservices remain responsive and resilient in dynamic environments.

The MicroProfile Config specification offers a robust and adaptable framework for managing application configurations. By implementing MicroProfile Config, developers can effectively manage configuration changes, ensuring their microservices remain responsive and resilient in

dynamic environments.

Integrating external configuration providers with MicroProfile Config extends the flexibility and dynamism of configuration management in microservices architectures. By implementing custom ConfigSources, applications can seamlessly adapt to various environments and configuration paradigms, pulling configuration data from virtually any source.

Handling missing or invalid configurations in MicroProfile Config involves using default values, optional properties, custom ConfigSource implementations, and appropriate exception handling. By following these practices, you can ensure that your application remains robust and flexible, even in the face of configuration challenges.

MicroProfile Health

Introduction

This chapter provides an in-depth exploration of MicroProfile Health, a critical component for ensuring the reliability and availability of microservices. This specification aims to enhance the observability of microservices in a cloud environment where automatic scaling, failover, and recovery are essential for maintaining service availability and reliability. In this chapter, we will learn about different types of health checks and standard health indicators provided by MicroProfile.

Topics to be covered:

- Overview of MicroProfile Health
- Key Concepts
- Types of Health Checks
- Exposing Health Checks
- Steps for Implementing Health Checks
- Integration with CDI
- Accessing Health Checks
- Kubernetes Probe Configuration
- Best Practices for Effective Health Checks

Overview of MicroProfile Health

The MicroProfile Health specification offers a standardized mechanism for microservices to report their health status. In the context of microservices, "health" refers to the ability of a microservice to perform its functions correctly and efficiently. The health check mechanism is crucial for operating microservices in a cloud or containerized environment where automated processes need to make decisions about whether to restart a failing service, reroute traffic away from an unhealthy service, or take other actions to maintain overall system reliability.

Let’s delve into the essentials of MicroProfile Health, its importance, and how it works.

Key Concepts

At its core, the MicroProfile Health specification defines a mechanism for microservices to report their health status via HTTP. These health checks can be used by external systems to verify the operational status of the services. This is crucial in modern cloud environments where automated processes continuously monitor service health, initiate failover procedures, and manage load balancing to ensure high availability and reliability.

Health Check

A **health check** is a test that can be used to determine the health of an application or service. This mechanism is implemented via standard HTTP endpoints that respond with the health status of the service. These endpoints are typically exposed at predefined paths, such as `/health`, `/health/live` (for liveness), `/health/ready` (for readiness), and `/health/started` (for startup). Health status is communicated through a simple JSON format, which can be easily interpreted by humans and machines. Applications servers that support MicroProfile may offer built-in mechanisms or simplified configurations to define such health checks.

Types of Health Checks

MicroProfile Health Check defines three main types of health checks, each with its own annotation to indicate the MicroProfile Health runtime about the type of check being performed, allowing it to execute and report health check responses appropriately. These are:

Liveness Checks

Liveness checks help to determine if a microservice is in a state where it can perform its functions correctly. A failing liveness check suggests that the microservice is in a broken state, and the only way to recover might be to restart the microservice. This type of health check is crucial for detecting deadlocks, infinite loops, or any conditions that render the microservice unresponsive or dysfunctional. Liveness checks are annotated with `@Liveness`.

Readiness Checks

Readiness checks are used to determine if a microservice is ready to process requests. If a readiness check fails, it indicates that the microservice should not receive any inbound requests because it's not ready to handle them properly. This can be due to the application still initializing, waiting for dependencies, or any other condition that would prevent it from correctly processing incoming requests. Readiness checks are annotated with `@Readiness`.

Startup Checks

Startup checks are designed for verifying the microservice's health immediately after it has started. This type of check is useful for applications that require additional initialization time or need to perform certain actions before they are ready to serve requests. Including startup checks in the health checking mechanism is crucial because if we hit the liveness probe before the application is fully initialized, it could cause a continuous restart loop. Startup checks provide a mechanism to postpone other health checks until certain startup conditions are fulfilled. This ensures that readiness and liveness probes are not prematurely activated, allowing the microservice adequate time to complete its initialization processes, such as loading configurations, establishing database connections, or performing necessary pre-service tasks.These checks are annotated with `@Startup`.

Exposing Health Checks

Health checks are exposed via HTTP endpoints automatically without additional configuration needed from the developer's side. The runtime environment provides these endpoints:

- `/health`: Aggregates all health check responses.
- `/health/live`: Returns responses from liveness checks.
- `/health/ready`: Returns responses from readiness checks.
- `/health/started`: Returns responses from startup checks.

These endpoints return a JSON object containing the overall status (UP or DOWN) and individual health check responses, including their names, statuses, and optional data.

Example JSON Response

For example a `LivenessCheck`, if accessed via `/health/live`, the JSON response might look something like this when the service is healthy:

```
{
  "status": "UP",
  "checks": [
    {
      "name": "LivenessCheck",
      "status": "UP"
    }
  ]
}
```

If the service is unhealthy, the "status" field would be "DOWN", and additional data might be provided to indicate the cause of the health check failure. Each type of health check is implemented as a procedure annotated with the respective annotation. Each procedure returns a `HealthCheckResponse` indicating the health status (UP or DOWN) and optionally includes additional details. Implementing these health check types in microservices architecture ensures that services are only used when they are in a healthy state and can correctly process requests. This enhances the overall reliability and maintainability of applications.

Standard Health Check

Applications can implement multiple health checks of each kind. The overall health status reported by the application is a logical AND of all individual health checks. A special endpoint `/health` aggregates the results from all three types of checks.

Implementing and Exposing Health Check

To implement health checks for microservices using MicroProfile Health, you would generally follow a pattern to define health check procedures that align with the services' operational characteristics. The Health Check API allows us to expose information about the health of our application. This information can be used by load balancers and other tools to determine if an application is healthy.

The `HealthCheck` interface

The `HealthCheck` functional interface uses CDI beans with annotations (`@Liveness`, `@Readiness`, and, `@Startup`) to mark a class as a health checker for liveness, readiness and startup. They are automatically discovered and registered by the runtime. Implementations of this interface are expected to be provided by applications.

The Health Check API defines a contract for health check implementations. A health check is a Java class that implements the HealthCheck functional interface:

```
package org.eclipse.microprofile.health;

@FunctionalInterface
public interface HealthCheck {
  HealthCheckResponse call();
}
```

You can check out the actual code here - https://github.com/eclipse/microprofile-health/blob/main/api/src/main/java/org/eclipse/microprofile/health/HealthCheck.java

The `HealthCheckResponse` class

The `HealthCheckResponse` class is used to represent the result of a health check invocation. It contains information about the health check, such as name, state (up or down), and data that can be used for troubleshooting.

The `call()` method of `HealthCheck` interface is used to perform the actual health check and return a `HealthCheckResponse` object:

```
package org.eclipse.microprofile.health;

public class HealthCheckResponse {

   private static final Logger LOGGER =
Logger.getLogger(HealthCheckResponse.class.getName());

    // the name of the health check.
    private final String name;

    // the outcome of the health check
    private final Status status;

    // information about the health check.
    private final Optional<Map<String, Object>> data;

    // Status enum definition
    public enum Status {
        UP, DOWN
    }
```

```
   // Getters
    public String getName() {
        return name;
    }

    public Status getStatus() {
        return status;
    }

    public Optional<Map<String, Object>> getData() {
        return data;
    }

}
```

The provided code snippet offers a conceptual and simplified implementation of the `HealthCheckResponse` class to illustrate how health check responses can be structured within the MicroProfile Health framework. To view the actual `HealthCheckResponse` class source code, please visit: https://github.com/eclipse/microprofile-health/blob/main/api/src/main/java/org/eclipse/microprofile/health/HealthCheckResponse.java

The `HealthCheckResponseBuilder` class

The `HealthCheckResponseBuilder` abstract class provides a fluent API for constructing instances of `HealthCheckResponse`. This means you can chain method calls to set various properties of the response in a single statement, improving code readability and maintainability.

```
package org.eclipse.microprofile.health;

public abstract class HealthCheckResponseBuilder {

    // Sets the name of the health check response.
    public abstract HealthCheckResponseBuilder name(String name) {
        this.name = name;
    }

    // Sets the status of the health check to UP
    public abstract HealthCheckResponseBuilder up();

    // Sets the status of the health check to DOWN
    public abstract HealthCheckResponseBuilder down();

    // Adds additional string data to the health check response
    public HealthCheckResponseBuilder withData(String key, String value);

    // Adds additional numeric data to the health check response
    public HealthCheckResponseBuilder withData(String key, long value);
```

```
    // Sets the status of the health check response
    public abstract HealthCheckResponseBuilder status(boolean up);

    // Builds and returns the HealthCheckResponse instance
    public abstract HealthCheckResponse build();

}
```

The above code snippet offers a conceptual and simplified definition of the `HealthCheckResponseBuilder` abstract class to illustrate how health check responses can be structured within the MicroProfile Health framework. For the actual `HealthCheckResponseBuilder` abstract class source code, please visit: https://github.com/eclipse/microprofile-health/blob/main/api/src/main/java/org/eclipse/microprofile/health/HealthCheckResponseBuilder.java

Steps for Implementing Health Checks

Below are the steps for implementing Health Checks for each of the microservices:

Add MicroProfile Health Dependency: To utilize MicroProfile Health in a Java project, include the MicroProfile Health API dependency in your *pom.xml* or *build.gradle* file.

For maven, add:

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

For gradle, add:

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

When implementing MicroProfile Health checks, including the MicroProfile Health API dependency in your project is not enough. You need an actual implementation on the classpath. This could be a MicroProfile-compatible server runtime such as Open Liberty, Quarkus, Payara Micro, or WildFly. Without an implementation present at runtime, the application will not be able to execute health checks.

The health information can be used by other tools to help keep our application running well.

Implementing Health Checks

Health checks in MicroProfile are implemented as CDI beans that implement the `HealthCheck` interface. Each health check procedure is a method that returns a `HealthCheckResponse`. You can define different types of health checks (readiness, liveness, and startup) depending on the type of

check by annotating the health check class with `@Readiness`, `@Liveness`, or `@Startup`. These methods return a `HealthCheckResponse` object, which includes the health check status (UP or DOWN) and additional metadata about the health check.

Readiness Check:

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

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Readiness;

import io.microprofile.tutorial.store.product.entity.Product;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

@Readiness
@ApplicationScoped
public class ProductServiceHealthCheck implements HealthCheck {

    @PersistenceContext
    EntityManager entityManager;

    @Override
    public HealthCheckResponse call() {
        if (isDatabaseConnectionHealthy()) {
            return HealthCheckResponse.named("ProductServiceReadinessCheck")
                    .up()
                    .build();
        } else {
            return HealthCheckResponse.named("ProductServiceReadinessCheck")
                    .down()
                    .build();
        }
    }

    private boolean isDatabaseConnectionHealthy(){
        try {
            // Perform a lightweight query to check the database connection
            entityManager.find(Product.class, 1L);
            return true;
        } catch (Exception e) {
            System.err.println("Database connection is not healthy: " +
e.getMessage());
            return false;
        }
    }
}
```

Liveness Check:

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

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.HealthCheckResponseBuilder;
import org.eclipse.microprofile.health.Liveness;

import jakarta.enterprise.context.ApplicationScoped;

@Liveness
@ApplicationScoped
public class ProductServiceLivenessCheck implements HealthCheck {

   @Override
   public HealthCheckResponse call() {
       Runtime runtime = Runtime.getRuntime();
       long maxMemory = runtime.maxMemory(); // Maximum amount of memory the JVM will
attempt to use
       long allocatedMemory = runtime.totalMemory(); // Total memory currently
allocated to the JVM
       long freeMemory = runtime.freeMemory(); // Amount of free memory within the
allocated memory
       long usedMemory = allocatedMemory - freeMemory; // Actual memory used
       long availableMemory = maxMemory - usedMemory; // Total available memory

       long threshold = 100 * 1024 * 1024; // threshold: 100MB

         // Including diagnostic data in the response
       HealthCheckResponseBuilder responseBuilder =
HealthCheckResponse.named("systemResourcesLiveness")
            .withData("FreeMemory", freeMemory)
            .withData("MaxMemory", maxMemory)
            .withData("AllocatedMemory", allocatedMemory)
            .withData("UsedMemory", usedMemory)
            .withData("AvailableMemory", availableMemory);

        if (availableMemory > threshold) {
            // The system is considered live
            responseBuilder = responseBuilder.up();
        } else {
            // The system is not live.
            responseBuilder = responseBuilder.down();
        }

        return responseBuilder.build();
    }
}
```

The above code uses the HealthCheckResponseBuilder to construct the response. Depending on the outcome of `checkDatabaseConnection()`, the health check response is marked either "up" or "down", and relevant data is added to the response using `.withData(key, value)`. This approach allows for rich, descriptive health check responses that can convey detailed status information, not just binary up/down states.

Startup Check:

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

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;

import jakarta.ejb.Startup;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.PersistenceUnit;

@Startup
@ApplicationScoped
public class ProductServiceStartupCheck implements HealthCheck{

    @PersistenceUnit
    private EntityManagerFactory emf;

    @Override
    public HealthCheckResponse call() {
        if (emf != null && emf.isOpen()) {
            return HealthCheckResponse.up("ProductServiceStartupCheck");
        } else {
            return HealthCheckResponse.down("ProductServiceStartupCheck");
        }
    }
}
```

Integration with CDI

The specification also emphasizes the importance of integrating health checks with the application's Context and Dependency Injection (CDI) context, enabling health check procedures to be automatically discovered and invoked by the runtime. MicroProfile Health thus provides a robust and standardized way to implement health checks, facilitating the management and orchestration of microservices in a cloud environment.

Accessing Health Checks

Once defined, these health check procedures are automatically discovered and invoked by the MicroProfile Health runtime. They are accessible through standardized HTTP endpoints provided by MicroProfile Health (`/health`, `/health/live`, `/health/ready`, `/health/started`) and can be used by

orchestration tools (like Kubernetes) or monitoring systems to manage and monitor the health of your microservices.

This approach allows you to tailor health checks to the operational specifics of each microservice, providing a robust mechanism for observing and managing your application's health in a cloud-native environment.

Kubernetes Probe Configuration

Integrating MicroProfile Health checks with Kubernetes probes allows you to leverage Kubernetes' native capabilities to manage the lifecycle of your containers based on their current health status. Specifically, you can map Liveness, Readiness, and Startup probes in Kubernetes to the corresponding health check types defined by the MicroProfile Health specification.

Here's a basic overview of how each type of MicroProfile Health check maps to Kubernetes probes:

- **Liveness Probes:** Determine if a container is running and healthy. If a liveness probe fails, Kubernetes will kill the container and create a new one based on the restart policy.
- *Readiness Probes: Determine if a container is ready to serve traffic. If a readiness probe fails, Kubernetes will stop sending traffic to that container until it passes again.
- **Startup Probes:** Determine if a container application has started. These are useful for applications that have a long startup time to prevent them from being killed by Kubernetes before they are up and running.

To configure these probes in your Kubernetes pod, you can use the livenessProbe, readinessProbe, and startupProbe fields in your container specification. Here's an example of how you might define a readiness probe in your Kubernetes pod configuration, that utilizes a MicroProfile Health endpoint:

```
apiVersion: v1
kind: Pod
metadata:
  name: mp-pod
spec:
  containers:
  - name: my-mp-app
    image: myimage:v1
ports:
- containerPort: 8080
readinessProbe:
  httpGet:
    path: /health/ready
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 2
  periodSeconds: 5
  failureThreshold: 3
```

In the above example, the readinessProbe is configured to make an HTTP GET request to the `/health/ready` endpoint, which is the default endpoint provided by MicroProfile Health for readiness checks. Similarly, you can configure livenessProbe and startupProbe by specifying `/health/live` and `/health/startup` endpoints respectively.

It's important to adjust the `initialDelaySeconds`, `timeoutSeconds`, `periodSeconds`, and `failureThreshold` according to the specifics of your application to ensure that Kubernetes accurately reflects the state of your containers based on its health checks.

Best Practices for Effective Health Checks

Here are some best practices for implementing and utilizing health checks effectively:

- **Clearly Define Health Check Types**: Use readiness, liveness, and startup checks appropriately to reflect the state of your microservices. This helps in accurately signaling the service's ability to handle traffic and its current operational state.
- **Implement Meaningful Health Checks**: Ensure that your health checks meaningfully reflect the operational aspects they are intended to monitor. Avoid trivial checks that do not accurately represent the service's health.
- **Utilize Health Check Responses**: Make effective use of the health check responses, including the UP/DOWN status and additional metadata. This information can be valuable for logging and reporting on the health state of your services.
- **Secure Health Check Endpoints**: Consider the security of your health check endpoints, especially if they expose sensitive details about the application's state.
- **Monitor Health Check Performance**: Health checks should be lightweight and not introduce significant overhead. Monitor the performance of your health checks and optimize as needed to prevent impacting the application's performance.
- **Logging Health Check Results**: Implementing logging within your health check procedures can provide insights into the health status over time. Log entries can be made when health check statuses change or when significant health-related events occur.
- **Reporting and Alerting**: Based on logged health check results, implement reporting mechanisms to visualize the health over time and set up alerting for when health checks fail. This could be integrated with existing monitoring and alerting tools.

By following these best practices, you can effectively implement and expose health checks in your MicroProfile applications, improving observability and reliability, especially in cloud-native environments.

Summary

This chapter provided a comprehensive overview of MicroProfile Health, emphasizing its critical role in enhancing the observability and reliability of microservices within cloud environments. Key topics included an introduction to the MicroProfile Health specification, detailed explanations of health check types (liveness, readiness, and startup checks), and guidance on implementing, exposing, and effectively utilizing these health checks.

The essence of MicroProfile Health lies in its standardized mechanism for microservices to report health status via HTTP endpoints, facilitating automated decision-making processes like scaling, failover, and recovery in cloud or containerized environments. The specification defines three primary types of health checks: liveness, readiness, and startup checks, each designed to assess different aspects of a microservice's operational status.

Implementing health checks involves creating procedures annotated with the respective health check annotations. These procedures return a HealthCheckResponse indicating the service's health status (UP or DOWN). These checks are automatically exposed via predefined HTTP endpoints, allowing easy integration with orchestration tools like Kubernetes.

The chapter also touched on best practices for effective health checks, including defining meaningful checks, utilizing health check responses, handling failures gracefully, and securing health check endpoints. In conclusion, MicroProfile Health offers a robust framework for monitoring and managing the health of microservices, ensuring that services remain reliable and available in dynamic cloud environments. By following the guidelines and best practices outlined in this chapter, developers can effectively implement and leverage health checks to maintain the overall health of their applications.

MicroProfile Metrics

Introduction

This chapter provides a comprehensive and detailed overview of MicroProfile Metrics, a widely used specification for monitoring microservices. You will gain an understanding of the various types of metrics and how you can use them to monitor microservices effectively. Additionally, this chapter covers the standard metrics provided by MicroProfile and how you can leverage them to monitor various aspects of microservices.

Furthermore, this chapter discusses the process of instrumenting microservices, which involves adding code to the application to collect metrics. You will learn how to expose endpoints to access metric data and interpret the data generated by these metrics.

This chapter also highlights the importance of integrating monitoring solutions with MicroProfile Metrics. You will learn how to incorporate monitoring solutions and choose the right monitoring solution for your needs.

By the end of this chapter, you will have a deep understanding of MicroProfile Metrics and the various techniques for monitoring microservices. This chapter will equip you with the knowledge and skills to effectively monitor your microservices and ensure they perform optimally.

Topics to be covered:

- Introduction to MicroProfile Metrics
- Need for Metrics in Microservices
- Types of Metrics
- MicroProfile Metrics Dependency
- Metrics Annotations
- Categories of Metrics
- Metric Registry
- Instrumenting Microservices with Metrics
- Creating Custom Metrics

Introduction to MicroProfile Metrics

It is essential to monitor your microservices to ensure smooth operations. You can monitor a microservice using two different techniques: Metrics and health checking. Health checks provide information on the health status of a service, such as whether it is up and running, while Metrics offer more detailed information on its performance, such as response times, throughput, and error rates. In the previous chapter, we discussed health checks and their importance. This chapter will cover the MicroProfile Metrics specification, which provides a standardized way of collecting and exposing performance data for Java microservices.

MicroProfile Metrics is a specification for developers who want to measure their applications' performance more thoroughly. It provides a set of annotations and APIs to track various metrics related to the application's health and performance. For instance, developers can use these APIs to track metrics such as the number of requests processed, the response time of each request, and the size of the response sent back to the client.

This specification defines a standardized format for exposing metrics, which other tools and frameworks can easily collect and track. By using this specification, developers can monitor the performance of their applications in real time and identify any issues that may impact the user experience.

Moreover, this specification defines a set of standard metrics that we can expose in Prometheus. With the help of this tool, developers can optimize their applications for better performance, ensuring that they meet the requirements of their consumers while delivering a seamless experience.

Prometheus is a powerful tool designed to monitor and collect metrics from your services. It provides a highly efficient time-series database system that securely stores your data for long-term analysis. With Prometheus, you can easily visualize and gain insights into your system's performance, allowing you to make informed decisions and optimize your services for better efficiency and reliability.

Need for Metrics in Microservices

Metrics, enables developers and operators to monitor and measure the behavior of microservices at runtime. This observability is crucial for:

- **Performance Tuning**: Identifying bottlenecks and optimizing resource utilization to ensure services are running efficiently.
- **Scalability**: Making informed decisions on when to scale services up or down based on real-time data on load and performance.
- **Troubleshooting**: Quickly pinpointing issues by analyzing trends in performance metrics, leading to reduced downtime.
- **Service Health Monitoring**: Complementing the MicroProfile Health checks by providing deeper insights into the internal state of a service, beyond simple up/down statuses.

Types of Metrics

MicroProfile Metrics offers a range of customizable metrics that can be used to measure and monitor microservices' performance. It allows developing microservices that are observable, manageable, and which provide insights into their behavior.

The MicroProfile Metrics specification includes four different types of metrics that serve specific monitoring purposes: Counter, Gauge, Histogram, and Timer. Each of these types offers unique insights into different aspects of application behavior and performance. Below is the breakdown of available metric types:

1. **Counter**: It is a simple metric type that represents a single numerical value that can only increase over time. This metric is typically used to count occurrences of certain events, such as the number of requests processed, items created, or tasks completed. Monitoring tools like Prometheus are commonly used to analyze changes in the Counter's value over specific intervals. These tools can track the differences in the Counter's value across time periods, providing insights into the rate of occurrences and trends.
2. **Gauge**: It is a metric type that measures an instantaneous value of something , which can arbitrarily go up or down. It's used to capture the value of a metric at a particular point in time like the size of a queue, memory usage, or current number of active user sessions. Gauges are typically used for values that change over time, providing a current "gauge" of the system's state.
3. **Histograms**: They provide a distribution of values for a given metric, which are useful for identifying performance outliers. It measures the frequency of values in different ranges (or "buckets") and is useful for tracking the distribution of values, such as response times or data sizes. Histograms can give insights into the average, percentiles, and trends of the measured data over time.
4. **Timer**: It is a specialized metric type that aggregates timing durations and provides data such as the count, total time, mean, and maximum duration. It can also report the duration distribution. Timers are invaluable for tracking the duration of certain activities or operations within your application, such as processing time or method execution time.

Note: Counters, Histograms, and Timers, which are updated synchronously when annotations or API calls are made to update them, Gauges are registered as callbacks. These callbacks are invoked to retrieve their value at the moment the list of metrics is requested, typically by a monitoring tool calling the /metrics endpoint. This allows Gauges to provide a real-time snapshot of dynamic values as they fluctuate.

By leveraging these metrics, developers and operators can gain a deeper understanding of how their microservices are performing. They can use this information to identify areas where improvements can be made and optimize their microservices' performance.

MicroProfile Metrics Dependency

If you're using Maven, add the following dependency to your pom.xml file located in the root folder of your project:

```
<dependency>
  <groupId>org.eclipse.microprofile.metrics</groupId>
  <artifactId>microprofile-metrics-api</artifactId>
  <version>5.1.1</version>
</dependency>
```

For Gradle, add the corresponding dependency to your build.gradle file located within the root folder of your project:

```
dependencies {
```

```
    providedCompile 'org.eclipse.microprofile.metrics:microprofile-metrics-api:5.1.1'
}
```

Metrics Annotations

MicroProfile Metrics defines a set of annotations to be used for exposing metrics. These annotations can be used on classes, methods, or fields. Table 7-1 shows the list of Metrics Annotation along with their descriptions.

Annotation	Description
`@Timed`	Times how long a method takes to execute and exposes this information as a metric.
`@Counted`	Tracks how many times a method is invoked and exposes this information as a metric.
`@Gauge`	Enables you to expose a custom metric that can be any value. It is useful for exposing application-specific metrics.

Besides annotations, MicroProfile Metrics also defines a set of programmatic APIs for working with metrics. These APIs can be used to register custom metrics or access existing metrics.

Categories of Metrics

In MicroProfile Metrics, metrics are organized into three distinct scopes: Base, Vendor, and Application. This categorization is designed to clearly separate metrics by their origin and relevance, making it easier for developers and operators to monitor and manage the performance of their microservices. Each scope serves a specific purpose and contains a different set of metrics:

- **Base Metrics** are common to all applications, such as the number of CPUs or the amount of free memory. These metrics provide essential information about the underlying Java Virtual Machine (JVM) and the core libraries that are common across all MicroProfile applications. Base metrics typically include JVM-specific metrics such as memory usage, CPU load, thread counts, and garbage collection statistics. The intention behind base metrics is to offer a consistent set of low-level metrics that are universally applicable and useful for monitoring the health and performance of the JVM itself, which is the foundation upon which all MicroProfile applications run. Base metrics are exposed under the path `/metrics?scope=base`.
- **Application Metrics** are specific to an application, they are defined by the developers of the MicroProfile applications themselves. These are custom metrics that are specific to the business logic or operational aspects of the application. Developers use annotations or programmatic APIs to create and register these metrics, tailoring them to monitor the performance and behavior of their application's unique functionalities. Application metrics enable developers to gain insights into the runtime characteristics of their application, such as the number of transactions processed, response times for specific endpoints, or the rate of specific business events. Application metrics are exposed under the path `/metrics?scope=application`.
- **Vendor Metrics** are specific to a particular vendor or technology. These metrics provide insights into the performance and behavior of the runtime's internal components and

extensions. Since different MicroProfile implementations may offer additional features or optimize certain areas differently, vendor metrics can vary widely between runtimes. They allow runtime vendors to expose unique metrics that are relevant to their implementation, offering users the ability to monitor vendor-specific aspects of their applications. Application metrics are exposed under the path `/metrics?scope=vendor`.

Besides the standard scopes above, MicroProfile Metrics also supports custom scopes. You can use custom scopes to group sets of metrics that you frequently expect to view together.

Note: In version 5.x, base metrics have become optional. This allows for flexibility in environments where these metrics may not be necessary or where they can be sourced from alternative monitoring tools.

Metric Registry

The **MetricRegistry** component acts as a container for storing and managing metrics within an application. It provides a structured way to collect, organize, and access various types of metrics (e.g., counters, gauges, histograms, and timers) for monitoring the behavior and performance of applications. It offers a centralized repository where metrics can be created and retrieved. This allows applications to consistently monitor critical operational and performance statistics.

Types of Metric Registries

MicroProfile Metrics creates metric registries for each scope:

- Application Scope (`MetricRegistry.Type.APPLICATION`): Contains custom metrics that are specific to the application. These are typically the metrics that developers explicitly create and register to monitor application-specific behaviors.
- Base Scope (`MetricRegistry.Type.BASE`): Contains metrics that are fundamental and common across all MicroProfile applications. These metrics provide basic information about the underlying JVM and application server.
- Vendor Scope (`MetricRegistry.Type.VENDOR`): Contains metrics that are specific to the implementation of the MicroProfile platform being used. These metrics offer insights into vendor-specific features and optimizations.

A metric registry is created as per the above scopes to enable the organization of metrics based on their origin and relevance.

Instrumenting Microservices with MicroProfile Metrics

Instrumenting microservices with MicroProfile Metrics enables developers to gain detailed insights into their application's operational health and performance. This level of observability is essential for maintaining scalable and resilient microservice architectures in dynamic environments.

Tracking response time using `@Timed`

MicroProfile Metrics also allows you to track a method's response time as a timed metric. The code example below shows how to use the @Timed annotation to track the response time.

```
import org.eclipse.microprofile.metrics.annotation.Timed;
// ...

public class ProductResource {

    // ...
    // Expose the response time as a timer metric
    @Timed(name = "productLookupTime",
            tags = {"method=getProduct"},
            absolute = true,
            description = "Time spent looking up products")
    public Response getProductById(@PathParam("id") Long id) {
        return productService.getProduct(productId);
    }

    // ...
```

It will expose a metric called `productLookupTime`, which will track the amount of time spent in the `getProduct()` method in seconds. You can visit the following URL `https://<hostname>:<port>/metrics?scope=application` (Replace `<hostname>` and `<port>` with the actual hostname and port where the server is running) to see the response time of this method as below:

```
...
# HELP productLookupTime_seconds_max Time spent looking up products
# TYPE productLookupTime_seconds_max gauge
productLookupTime_seconds_max{method="getProduct",mp_scope="application",} 0.002270643
...
```

Tracking number of invocations using `@Counted`

MicroProfile Metrics also allows you to track the number of invocations of a method as a counter metric. The code example below shows how to use the `@Counted` annotation to track the invocation count.

```
import org.eclipse.microprofile.metrics.Metrics;

public class ProductResource {

    // Expose the invocation count as a counter metric
    @Counted(name = "productAccessCount",
        absolute = true,
        description = "Number of times the list of products is requested")
    public Response getAllProducts() {
       // Method implementation
       // ....
    }
}
```

In the example above, the `@Counted` annotation tells MicroProfile Metrics to track the number of invocations of the `getProducts()` method and expose this metric as a counter. The name, and description of the metric can also be specified. You can visit the following URL `https://localhost:<port>/metrics?scope=application` (Replace `<port>` with the actual port where the server is running) to see the number of times this method is called as below:

```
...
# HELP productAccessCount_total Number of times the list of products is requested
# TYPE productAccessCount_total counter
productAccessCount_total{mp_scope="application",} 3.0
...
```

Creating a Custom Metric

Creating a custom metric to track the number of products in a catalog involves using the MicroProfile Metrics API. This custom metric can be implemented as a gauge, which measures an instantaneous value (in this case, the current number of products in the catalog).

```
import org.eclipse.microprofile.metrics.annotation.Gauge;
...

@Path("/products")
@ApplicationScoped
public class ProductResource {
  // ...

  @GET
  @Path("/count")
  @Produces(MediaType.APPLICATION_JSON)
  @Gauge(name = "productCatalogSize",
    unit = "none",
  description = "Current number of products in the catalog")
  public long getProductCount() {
     return productCatalogSize;
  }
}
```

The gauge metric `productCatalogSize` can be accessed through the following endpoint:

`/metrics?name=io_microprofile_tutorial_store_product_resource_ProductResource_productCatalogSize`

This custom metric implementation provides a real-time insight into the size of your product catalog, which can be invaluable for monitoring the scale of your service's data and understanding its behavior over time.

```
# HELP
```

```
io_microprofile_tutorial_store_product_resource_ProductResource_productCatalogSize
Current number of products in the catalog
# TYPE
io_microprofile_tutorial_store_product_resource_ProductResource_productCatalogSize
gauge
io_microprofile_tutorial_store_product_resource_ProductResource_productCatalogSize{mp_
scope="application",} 8.0
```

Vendors may, by their own implementation, support `/metrics?name=<name>` to directly retrieve that metric from all scopes. However, the specification itself only illustrates `/metrics?scope=<scope>&name=<name>`.

Summary

This Chapter delved into the intricacies of MicroProfile Metrics, illuminating its role as a pivotal specification for efficiently monitoring microservices. Now you are equipped with a thorough understanding of diverse metric types and their application for monitoring microservice performance. This chapter highlighted the need for regular microservice monitoring via metrics and health checks, emphasizing metrics for detailed performance insights such as response times and throughput. Through practical examples, this chapter showcases how to instrument microservices with MicroProfile Metrics, leveraging standard metrics, and creating custom metrics to monitor microservices comprehensively.

MicroProfile Fault Tolerance

In a Microservices architecture, an application consists of multiple smaller, autonomous services. This architecture enhances development flexibility, agility, and scalability but introduces new challenges, particularly in ensuring the application's reliability and managing failures. Unlike monolithic applications, where defects are localized, a single failure in one microservice can propagate across the entire application, potentially causing widespread outages. Therefore, fault tolerance is critical in a microservices architecture to ensure that failures are seamlessly isolated, managed, and recovered.

MicroProfile Fault Tolerance offers strategies for building resilient and reliable microservices, ensuring service continuity and stability even during unexpected failures.

This chapter explains how to enhance your microservices' resilience and reliability using MicroProfile Fault Tolerance capabilities and annotations. We will also demonstrate how to implement key strategies such as timeouts, retries, fallbacks, circuit breakers, and bulkheads to handle faults. By the end of the chapter, you will understand how to use these strategies to enhance the resilience of your microservices.

Topics to be Covered

- What is Fault Tolerance?
- Key Strategies for Enhancing Fault Tolerance
- Implementing Retry Policies and Configuration
- Avoiding and Managing Cascading Failures
- Configuring Circuit Breaker
- Using `@Asynchronous` Annotation
- Setting Timeouts
- Implementing Fallback Logic
- Isolating Resources for Fault Tolerance

What is Fault Tolerance?

Fault tolerance is a system's ability to continue working correctly even in case of unexpected failures. A fault-tolerant system should be able to detect, isolate, and recover from errors without human intervention. It is critical in applications based on modern microservices architectures where individual component failures are inevitable due to network issues, resource limitations, or transient errors.

Key Strategies for Enhancing Fault Tolerance

Some of the key strategies for enhancing the fault tolerance of a microservices-based application include:

Asynchronous Execution

Asynchronous execution allows operations to run in a separate thread. It means the caller does not have to wait for the operation to finish, making the application more responsive. For example, when a user searches for products in the product catalog service, the service can asynchronously fetch product recommendations from an external API while immediately returning the main search results to the user, ensuring a fast and responsive experience.

When applied individually or in combination, these strategies form the foundation of a fault-tolerant microservices architecture. The following sections delve deeper into their implementation and best practices.

Timeout

A timeout sets a time limit for operations, preventing indefinite waits and freeing up system resources for other tasks. For instance, a timeout in payment service ensures that the application can recover gracefully if the payment processing is taking too long to respond.

Retry

A retry allows the system to automatically retry failed operations, particularly useful for handling transient errors like temporary network glitches. You can customize the retry policy with parameters such as the delay between retries and maximum retries. Adding jitter prevents synchronized retries across services.

For example, a payment service can retry a failed payment authorization request with an external payment gateway to ensure successful transaction processing.

Bulkhead

A bulkhead isolates failures in one part of a system from other parts by segregating resources, such as thread pools, connection pools, or memory, among different microservices interactions.

For example, in an e-commerce application, the catalog service can implement bulkheads using separate thread pools or connection pools for different upstream dependencies, such as the product database and the pricing service. If the pricing service becomes slow or unresponsive, a bulkhead prevents it from consuming all the resources of the catalog service, ensuring that requests to fetch product details from the database continue to work unaffected.

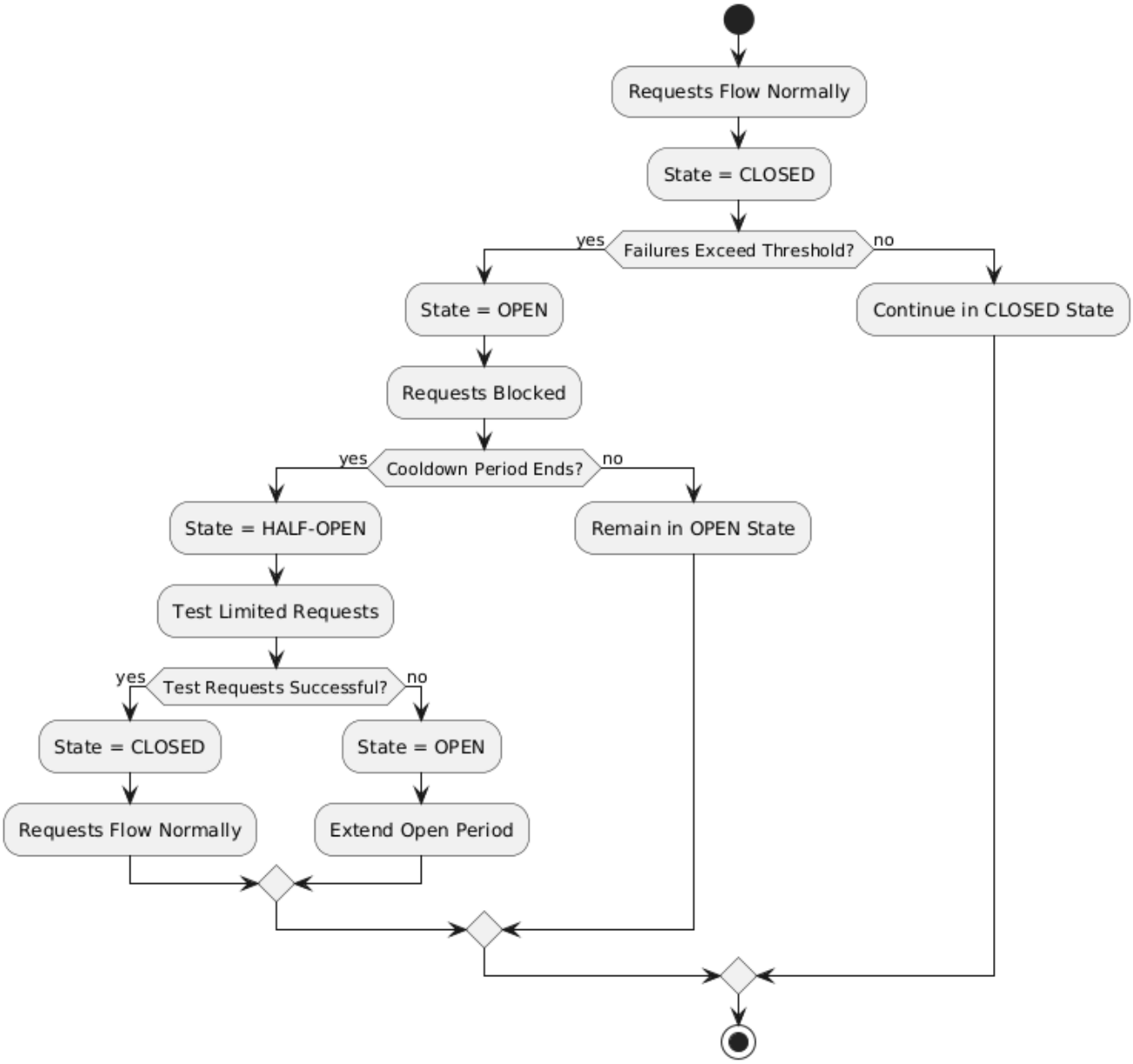
Fallback

A fallback provides a default response if an operation fails. It ensures the system continues providing a meaningful response instead of completely failing. For example, if the database fails or becomes slow in the product catalog service, the system can fetch cached product data to continue serving user requests for product listings.

Circuit Breaker

A circuit breaker stops an application from making too many unsuccessful requests to another

system. If the number of failures exceeds a threshold, the circuit breaker will `open`, causing all subsequent requests to fail immediately. After a configured delay, the circuit breaker will `half-open` and allow limited requests. If those requests succeed, the circuit breaker will `close` and let all requests go through.



MicroProfile Fault Tolerance 5. MicroProfile Fault Tolerance

For example, a circuit breaker can be applied to calls to an external inventory service in the Product Catalog Microservice. If the inventory service starts failing or becomes unresponsive, the circuit breaker will `open`, preventing repeated requests and reducing load. After a configured delay, the circuit breaker will `half-open` to test the availability of the inventory service with a few requests. If those succeed, the circuit breaker will `close`, resuming normal operations.

Fault Tolerance API

The Fault Tolerance API equips developers with annotations to enhance the resilience of microservices against failures. It integrates seamlessly with the MicroProfile Config API, enabling the dynamic configuration of fault tolerance behaviors without modifying the application code.

This section will explore using the Fault Tolerance API to build a robust, fault-tolerant microservice.

Adding Dependency for Fault Tolerance API

To use the Fault Tolerance API in your project, include the following dependency in your `pom.xml` file. Ensure you specify the version (e.g., 4.1.1) compatible with your MicroProfile runtime.

```
<dependency>
  <groupId>org.eclipse.microprofile.fault-tolerance</groupId>
  <artifactId>microprofile-fault-tolerance-api</artifactId>
  <version>4.1.1</version>
</dependency>
```

The Fault Tolerance API defines a contract for fault tolerance implementations.

MicroProfile Fault Tolerance Annotations

The MicroProfile Fault Tolerance annotations provide a declarative way to implement fault-tolerant behavior in Java methods, allowing developers to handle failures gracefully with minimal code changes.

List of Annotations

Annotation	Description
`@Asynchronous`	Ensures that the annotated method executes in a separate thread, allowing non-blocking execution. This is useful for improving responsiveness and handling long-running tasks asynchronously.
`@Retry`	Specifies that the annotated method should automatically retry on failure. Parameters such as `maxRetries`, `delay`, `maxDuration`, and `jitter` control retry behavior. Configurations can be externalized using MicroProfile Config.
`@Timeout`	Specifies the maximum duration (in milliseconds) the method can execute before being aborted. If the timeout is exceeded, a `FaultToleranceException` is thrown.
`@CircuitBreaker`	Defines a circuit breaker mechanism to prevent repeated calls to a failing method. Includes parameters like `failureRatio`, `delay`, and `requestVolumeThreshold`.
`@Fallback`	Specifies alternative logic to execute when the primary method fails. This ensures meaningful responses and graceful degradation.

`@Bulkhead`	Limits the number of concurrent method executions to isolate system resources and prevent cascading failures.

Implementing Retry Policies and Configuration

Retries are a fundamental fault tolerance strategy for managing transient failures such as temporary network outages or intermittent service unavailability. The `@Retry` annotation in the MicroProfile Fault Tolerance API provides a simple and effective way to implement retry policies. By customizing parameters such as the number of retries, delay between attempts, and conditions for retries, you can ensure your application responds to failures gracefully and minimizes downtime.

Applying `@Retry` in `PaymentService` class

Below is an example of applying the `@Retry` annotation in a `processPayment` method within a `PaymentService` class of the MicroProfile e-commerce project:

```
package io.microprofile.tutorial.store.payment.service;

import org.eclipse.microprofile.faulttolerance.Retry;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.core.MediaType;

public class PaymentService {

    @Retry(
        maxRetries = 3,
        delay = 2000,
        jitter = 500,
        retryOn = PaymentProcessingException.class,
        abortOn = CriticalPaymentException.class
    )
    public Response processPayment(PaymentDetails paymentDetails) throws
PaymentProcessingException {
        System.out.println("Processing payment for amount: " +
paymentDetails.getAmount());

        // Simulating a transient failure
        if (Math.random() > 0.7) {
            throw new PaymentProcessingException("Temporary payment processing
failure");
        }

        return Response.ok("{\"status\":\"success\"}",
MediaType.APPLICATION_JSON).build();
    }
}
```

Defining the PaymentDetails Class

To store the necessary payment information, the following `PaymentDetails` class is used. This class acts as a simple data container for payment-related details.

```
public class PaymentDetails {
    private double amount;

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```

Creating Custom Exception Classes for Handling Failures

The `PaymentProcessingException` class represents a recoverable error, which triggers retries when thrown.

```
package io.microprofile.tutorial.store.payment.exception;

public class PaymentProcessingException extends Exception {
   public PaymentProcessingException(String message) {
       super(message);
   }
}
```

The `CriticalPaymentException` is considered a non-recoverable failure. If this exception occurs, the retry process is aborted.

```
package io.microprofile.tutorial.store.payment.exception;

public class CriticalPaymentException extends Exception {
    public CriticalPaymentException(String message) {
        super(message);
    }
}
```

In this example, the `processPayment` method attempts to process a payment. If a transient failure occurs (e.g., `PaymentProcessingException`), the method retries up to three times (`maxRetries = 3`), and there is a delay of 2000 milliseconds between retries (`delay = 2000`), with a random variation of up to 500 milliseconds added to the delay (`jitter = 500`) to avoid synchronized retries (e.g. thundering herd problem). The retries are attempted only for the exception `PaymentProcessingException` (`retryOn = PaymentProcessingException.class`) and are aborted if a `CriticalPaymentException` is

encountered (`abortOn = CriticalPaymentException.class`).

This approach helps maintain application resilience while preventing unnecessary retries that could worsen critical failures.

Understanding the `@Retry` Parameters

A retry policy specifies the conditions under which an operation should be retried. The key attributes of the `@Retry` annotation include:

Parameter	Description
`maxRetries`	Specifies the maximum number of retries.
`delay`	Sets the time (in milliseconds) to wait between retry attempts.
`jitter`	Adds a random variation (in milliseconds) to the delay to avoid synchronized retries.
`retryOn`	Defines the exception(s) that should trigger a retry. Defaults to all exceptions if not specified.
`abortOn`	Specifies the exception(s) that should not trigger a retry, overriding the default retry behavior.
`maxDuration`	Limits the total time (in milliseconds) that retries can be attempted.

Externalizing Configuration with MicroProfile Config

Retry policies can be externalized using the MicroProfile Config API. This allows you to modify the retry behavior without changing the application code. Here's how to externalize the configuration:

- Add the `@Retry` annotation with minimal attributes:

```
package io.microprofile.tutorial.store.payment.service;

import org.eclipse.microprofile.faulttolerance.Retry;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.core.MediaType;

public class PaymentService {

    @Retry
    public Response processPayment(PaymentDetails paymentDetails) throws
PaymentProcessingException {
        System.out.println("Processing payment for amount: " +
paymentDetails.getAmount());

        // Simulating a transient failure
        if (Math.random() > 0.7) {
            throw new PaymentProcessingException("Temporary payment processing
```

```
failure");
        }

        return Response.ok("{\"status\":\"success\"}",
MediaType.APPLICATION_JSON).build();
    }
}
```

- Define the retry policy in a configuration file (e.g., microprofile-config.properties):

```
io.microprofile.tutorial.store.payment.service.PaymentService/processPayment/Retry/max
Retries=3
io.microprofile.tutorial.store.payment.service.PaymentService/processPayment/Retry/del
ay=2000
io.microprofile.tutorial.store.payment.service.PaymentService/processPayment/Retry/jit
ter=500
```

In this approach, you gain flexibility to adapt retry policies based on the environment, such as increasing retry attempts in production or reducing delays during testing.

Best Practices for Retry Policies

- **Limit Retries:** Avoid setting `maxRetries` too high, as excessive retries can overwhelm the system or cause cascading failures.
- **Use Jitter:** Always configure jitter to reduce the risk of synchronized retry attempts by multiple services.
- **Abort Non-Recoverable Errors:** Use the `abortOn` parameter to exclude critical exceptions that retries cannot resolve.
- **Monitor Metrics:** Integrate with MicroProfile Metrics to track retry patterns and adjust configurations dynamically based on real-world performance.
- **Combine Strategies:** For robust error handling, use retries alongside other fault tolerance mechanisms, such as timeouts and circuit breakers.

Avoiding and Managing Cascading Failures

In a distributed microservices architecture, cascading failures occur when the failure of one service propagates to others, potentially causing widespread system outages. Such failures often result from tightly coupled services, unbounded retries, or resource exhaustion.

Causes of Cascading Failures

- **Tight Coupling:** Dependencies between services without sufficient isolation mechanisms.
- **Unbounded Retries:** Excessive retries on failing services, overwhelming resources.
- **Resource Contention:** Exhaustion of critical resources such as thread pools, memory, or database connections.

- **Lack of Fail-Safe Mechanisms:** Missing circuit breakers, bulkheads, or fallback logic.

Strategies to Prevent Cascading Failures

- Use **circuit breakers** to isolate failing services.
- Apply **bulkheads** to limit the scope of failures and resource usage.
- Set **timeouts** to prevent long-running operations from blocking resources.
- Design retries with care to avoid overwhelming the system.

Configuring Circuit Breaker

A circuit breaker is a critical fault tolerance mechanism that protects a system from repeated failures of a dependent service. It stops repeated calls to a failing service, allowing it to recover.

Circuit Breaker Parameters

Parameter	Description
`failureRatio`	Specifies the proportion of failed requests required to open the circuit breaker.
`requestVolumeThreshold`	The minimum number of requests made in a rolling time window before the failure ratio is evaluated.
`delay`	The time (in milliseconds) the circuit breaker remains open before transitioning to the "half-open" state.
`successThreshold`	The number of consecutive successful test requests required in the "half-open" state to close the circuit breaker.
`failOn`	Specifies the exception(s) considered failures contributing to the failure ratio.

Below is an example of configuring a circuit breaker for a service method using the `@CircuitBreaker` annotation:

```
@CircuitBreaker(
    requestVolumeThreshold = 10,
    failureRatio = 0.5,
    delay = 5000,
    successThreshold = 2,
    failOn = RuntimeException.class
)
public String getProduct(Long id) {
    // Logic to call the product details service
    if (Math.random() > 0.7) {
        throw new RuntimeException("Simulated service failure");
    }
```

```
    return repository.findProductById(id);
}
```

In the above code, the circuit breaker opens if 50% of requests fail (`failureRatio = 0.5`) after at least 10 requests (`requestVolumeThreshold = 10`). It remains open for 5 seconds (`delay = 5000`) and transitions to the "half-open" state to test recovery. Two consecutive successful requests (`successThreshold = 2`) in the "half-open" state close the circuit breaker.

Externalizing Circuit Breaker Configuration

Using MicroProfile Config, you can externalize circuit breaker parameters to make them adjustable without code changes as below:

- Update the `@CircuitBreaker` annotation:

```
@CircuitBreaker (failOn = RuntimeException.class)
public String getProduct(Long id) {
    // Logic to call the product details service
    if (Math.random() > 0.7) {
        throw new RuntimeException("Simulated service failure");
    }
    return productRepository.findProductById(id);
}
```

- Define the configuration in **microprofile-config.properties**:

```
io.microprofile.tutorial.store.payment.service.ProductService/fetchProductDetails/Circ
uitBreaker/requestVolumeThreshold=10
io.microprofile.tutorial.store.payment.service.ProductService/fetchProductDetails/Circ
uitBreaker/failureRatio=0.5
io.microprofile.tutorial.store.payment.service.ProductService/fetchProductDetails/Circ
uitBreaker/delay=5000
io.microprofile.tutorial.store.payment.service.ProductService/fetchProductDetails/Circ
uitBreaker/successThreshold=2
```

Best Practices for Circuit Breaker

- **Set Realistic Failure Ratios and Thresholds:** Tailor parameters to your services' expected load and failure behavior.
- **Monitor Metrics:** Use MicroProfile Metrics to monitor circuit breaker state transitions.
- **Combine with Other Strategies:** Use circuit breakers alongside retries and timeouts for a robust fault tolerance setup.

Using `@Asynchronous` Annotation

The `@Asynchronous` annotation in MicroProfile Fault Tolerance is used to enable asynchronous execution of methods. It allows operations to run in a separate thread, freeing up the main thread

for other tasks. This approach enhances the application's responsiveness and scalability, particularly in high-concurrency or latency-sensitive scenarios.

Why Use `@Asynchronous`?

1. **Improved Responsiveness**: The caller does not need to wait for the method execution to complete, allowing the application to remain interactive.
2. **Non-Blocking Execution**: Long-running operations are offloaded to a separate thread, preventing bottlenecks.
3. **Scalability**: By decoupling method execution from the calling thread, you can handle higher loads without increasing thread contention.

Implementation

Below is an example of using the `@Asynchronous` annotation with MicroProfile Fault Tolerance:

```
package io.microprofile.tutorial.store.payment.service;

import org.eclipse.microprofile.faulttolerance.Bulkhead;
import jakarta.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.faulttolerance.Asynchronous;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

@ApplicationScoped
public class PaymentService {

    private static final int SIMULATED_DELAY_MS = 2000;

    /**
     * Processes payments asynchronously
     *
     * @return A CompletionStage with the result of the operation.
     */
    @Asynchronous
    public CompletionStage<String> processPayment() {
        simulateDelay();
        return CompletableFuture.completedFuture("Payment processed asynchronously.");
    }

    /**
     * Simulates a delay in processing
     */
    private void simulateDelay() {
        try {
            Thread.sleep(SIMULATED_DELAY_MS); // Simulating delay
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Error during simulated delay", e);
```

```
        }
    }
}
```

Externalizing Timeout Configuration

Timeout values can be externalized using the MicroProfile Config API, allowing flexibility to adjust values without modifying code. Here's how: * Define the @Timeout annotation without specifying the value:

```
@Timeout
public String fetchData() {
    // Logic
}
```

- Configure the timeout in **microprofile-config.properties**:

```
io.microprofile.tutorial.store.payment.service.ProductService/fetchData/Timeout/value=
1500
```

Best Practices for Using @Asynchronous

- **Use `CompletableStage` or `Future`**: Return types like `CompletableStage` allow asynchronous methods to integrate seamlessly with other asynchronous workflows.

Asynchronous Execution in Fault Tolerance Strategies

When used with other fault tolerance strategies, `@Asynchronous` provides a powerful mechanism to handle faults without impacting the system's responsiveness:

1. **Asynchronous with Bulkhead**:
    - Isolates resources while maintaining non-blocking execution.
    - Handles concurrent requests efficiently using thread pools.
2. **Asynchronous with Circuit Breaker**:
    - Prevents system overload during failures by breaking the circuit for failing asynchronous methods.
    - The circuit breaker's delay allows recovery while new threads are available for other tasks.

Setting Timeouts

Timeouts are an essential fault tolerance strategy to prevent long-running operations from consuming resources indefinitely. Slow or unresponsive services can degrade overall system performance and reliability in a microservices architecture. The `@Timeout` annotation provided by MicroProfile Fault Tolerance allows you to define a maximum duration for a method to complete, ensuring that system resources remain available for other tasks.

Why Use Timeouts?

In distributed systems, slow responses from downstream services can cascade through the system, leading to resource contention and degraded performance. Timeouts allow you to: - Abort operations that exceed acceptable time limits. - Free system resources for other operations. - Trigger alternative strategies, such as fallbacks, to maintain functionality.

```
package io.microprofile.tutorial.store.payment.service;

import io.microprofile.tutorial.store.payment.entity.PaymentDetails;
import io.microprofile.tutorial.store.payment.exception.PaymentProcessingException;

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

import java.util.concurrent.CompletionStage;
import java.util.concurrent.CompletableFuture;
import java.util.logging.Logger;

import org.eclipse.microprofile.faulttolerance.Asynchronous;
import org.eclipse.microprofile.faulttolerance.Timeout;

@ApplicationScoped
public class PaymentService {

    private static final int TIMEOUT_MS = 1000;
    private static final double FAILURE_THRESHOLD = 0.7;

    @Inject
    private Logger logger;

    /**
     * Processes payments asynchronously with a timeout.
     *
     * @param paymentDetails the payment details
     * @return a CompletionStage with the result of the operation
     */
    @Asynchronous
    @Timeout(TIMEOUT_MS)
    public CompletionStage<String> processPayment(PaymentDetails paymentDetails) {
        return CompletableFuture.supplyAsync(() -> {
            simulateDelay();
            logger.info("Processing payment for amount: " +
paymentDetails.getAmount());

            if (Math.random() > FAILURE_THRESHOLD) {
                throw new PaymentProcessingException("Temporary payment processing
failure");
            }
```

```
            return "{\"status\":\"success\", \"message\":\"Payment processed
successfully.\"}";
        }).exceptionally(ex -> {
            logger.warning("Payment processing failed: " + ex.getMessage());
            return "{\"status\":\"failure\", \"message\":\"Payment failed due to a
temporary issue.\"}";
        });
    }

    /**
     * Simulates a delay in processing.
     */
    private void simulateDelay() {
        try {
            Thread.sleep(2000); // Simulating delay
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            logger.severe("Error during simulated delay: " + e.getMessage());
            throw new RuntimeException("Error during simulated delay", e);
        }
    }
}
```

In this example, the `@Timeout(1000)` annotation specifies that the `processPayment` method must complete within 1000 milliseconds (1 second). If the execution exceeds this time, a `TimeoutException` will be thrown, and the process will terminate. `@Asynchronous` ensures non-blocking execution by making the method run in a separate thread. To explore the benefits of asynchronous programming with MicroProfile Fault Tolerance, the following resources provide valuable insights and real-world examples:

- Asynchronous Programming with MicroProfile Fault Tolerance (Part 1)
- Asynchronous Programming with MicroProfile Fault Tolerance (Part 2)

These articles explain how asynchronous execution enhances system responsiveness, reduces blocking, and ensures better resource utilization in MicroProfile applications.

Best Practices for Timeouts

- **Align Timeouts with SLAs:** Ensure timeout values align with service-level agreements and user expectations.
- **Monitor Performance:** Use MicroProfile Metrics to monitor execution times and identify operations requiring optimized timeout values.
- **Combine with Fallbacks:** Always pair timeouts with fallback logic to provide a reliable response in case of delays.
- **Avoid Overly Short Timeouts:** Overly aggressive timeout settings may cause unnecessary failures, particularly in high-latency environments.
- **Combine Timeout with Asynchronous:** Use timeout together with asynchronous to improve

responsiveness and prevent blocking the calling thread. This approach ensures better resource utilization and system scalability during long-running operations.

Implementing Fallbacks

Fallbacks provide a default response when an operation fails. They ensure the system continues to function, even if the primary operation cannot complete successfully. The `@Fallback` annotation in MicroProfile Fault Tolerance allows developers to define fallback logic for a method, ensuring graceful degradation.

Why Use Fallbacks?

Fallbacks help to: - Maintain system availability during failures. - Provide a meaningful response to users instead of complete failure. - Improve user experience by minimizing disruptions.

```
import org.eclipse.microprofile.faulttolerance.Fallback;
import jakarta.ws.rs.core.Response;

public class PaymentService {

    @Fallback(fallbackMethod = "fallbackProcessPayment")
    public Response processPayment(PaymentDetails paymentDetails) {
        // Simulate a failure
        throw new RuntimeException("Service Unavailable");
    }

    public Response fallbackProcessPayment(PaymentDetails paymentDetails) {
        return Response.ok("{\"status\":\"failed\", \"message\":\"Payment service is
currently unavailable.\"}").build();
    }
}
```

In this example: - The `@Fallback` annotation specifies that if the `processPayment` method fails, the `fallbackProcessPayment` method will be executed. - The fallback method provides a meaningful response, ensuring the user is informed of the service unavailability.

Using Fallback Handlers

A fallback handler class can implement the `FallbackHandler<T>` interface, allowing for reusable fallback logic across multiple methods.

```
import org.eclipse.microprofile.faulttolerance.Fallback;
import org.eclipse.microprofile.faulttolerance.FallbackHandler;
import org.eclipse.microprofile.faulttolerance.ExecutionContext;


public class ProductService {

    @Fallback(FallbackHandlerImpl.class)
```

```
    public Product getProduct(Long id) {
        // Logic to call the product details service
        if (Math.random() > 0.7) {
            throw new RuntimeException("Simulated service failure");
        }

        return productRepository.findProductById(id);
    }
}

public class FallbackHandlerImpl implements FallbackHandler<String> {
    @Override
    public String handle(ExecutionContext context) {
        return "Fallback response for product details.";
    }
}
```

Combining Fallbacks with Other Fault Tolerance Strategies

Fallback logic can be combined with other fault tolerance mechanisms to create a robust strategy: - **Timeout with Fallback:** Ensure operations terminate within a specific time and provide a fallback if they fail.

Example:

```
import org.eclipse.microprofile.faulttolerance.Fallback;
import org.eclipse.microprofile.faulttolerance.Timeout;

import jakarta.enterprise.context.RequestScoped;

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

@RequestScoped
public class ProductService {

    @Inject
    private ProductRepository productRepository; // Access to the database

    @Inject
    private ProductCache productCache; // Cache mechanism

    /**
     * Retrieves a list of products. If the operation takes longer than 2 seconds,
     * fallback to cached data.
     */
    @Timeout(2000) // Set timeout to 2 seconds
    @Fallback(fallbackMethod = "getProductsFromCache") // Fallback method
    public List<Product> getProducts() {
        if (Math.random() > 0.7) {
```

```
            throw new RuntimeException("Simulated service failure");
        }
        // database call
        return productRepository.findAllProducts();
    }

    /**
     * Fallback method to retrieve products from the cache.
     */
    public List<Product> getProductsFromCache() {
        System.out.println("Fetching products from cache...");
        return productCache.getAll().stream()
                .map(obj -> (Product) obj)
                .collect(Collectors.toList());
    }
}
```

This example demonstrates the use of MicroProfile Fault Tolerance annotations `@Timeout` and `@Fallback` to enhance the resilience of the `ProductService`. When `getProducts()` method is invoked, the application tries to retrieve product data from the database using `productRepository.findAllProducts()`. The `@Timeout(2000)` annotation ensures that this operation completes within 2 seconds. If the query executes successfully within this time, the method returns the product list as expected. However, if the execution time exceeds the timeout limit, a `TimeoutException` is triggered. Additionally, if an exception occurs within the time limit, the method also fails. To handle such failures gracefully, the `@Fallback` annotation specifies `getProductsFromCache()` as an alternative method. When a timeout or exception occurs, the fallback method is invoked, fetching product data from the cache instead of the database. This approach guarantees service availability and ensures a seamless user experience, even in scenarios where the database is slow or temporarily unavailable. For improved scalability and performance, `@Asynchronous` can be combined with `@Timeout` and `@Fallback`. This allows the method to execute in a non-blocking manner, freeing up system resources and enabling parallel processing of multiple requests. By utilizing asynchronous execution, the application can handle high loads efficiently while maintaining fault tolerance.

Externalizing `@Timeout` Configuration using MicroProfile Config

To externalize the @Timeout configuration using MicroProfile Config, you can replace the hardcoded timeout value with a configurable property. This allows us to modify the timeout dynamically without changing the source code.

- Define a Configurable Property: Use `@ConfigProperty` to inject the timeout value.

```
// ...
@RequestScoped
public class ProductService {

    @Inject
    private ProductRepository productRepository; // Access to the database
```

```
    @Inject
    private ProductCache productCache; // Cache mechanism

    // Inject the timeout value from MicroProfile Config
    @Inject
    @ConfigProperty(name = "product.service.timeout", defaultValue = "2000")
    private long timeoutValue;

    // ...
```

- Use the Configured Value in @Timeout Annotation: Define a getter method and using it in the annotation.

```
    ...
    /**
     * Provide the timeout value dynamically using a method reference.
     */
    @Timeout(value = getTimeout()) // Use method reference to fetch dynamic value
    public long getTimeout() {
        return timeoutValue;
    }
```

- Define the Configuration Property: Configure the timeout in **microprofile-config.properties**:

```
io.microprofile.tutorial.store.product.service.ProductService.timeout=3000
```

This sets the timeout to 3000 milliseconds (3 seconds) instead of the default 2000 making your application more configurable and adaptable without code changes.

Best Practices for Fallbacks

- **Keep Fallbacks Lightweight:** Ensure fallback logic is simple and reliable, avoiding dependencies on other potentially failing services.
- **Provide Meaningful Responses:** The fallback response should maintain a reasonable user experience, even if it cannot replicate full functionality.
- **Monitor Fallback Usage:** Use metrics to track the frequency of fallback execution, which can indicate service health and the need for improvements.
- **Plan for Degraded Functionality:** Ensure the fallback behavior aligns with business priorities and provides the most critical features.

Combining Fault Tolerance Strategies

Combining fault tolerance strategies, such as `@Timeout`, `@Fallback`, `@CircuitBreaker`, and `@Retry`, ensures resilience and efficient resource usage. Externalize configurations with MicroProfile Config for flexibility across environments.

Isolating Resources for Fault Tolerance

Resource isolation is a key principle in building resilient microservices. By isolating resources, you prevent failures in one part of the system from spreading and affecting others. MicroProfile Fault Tolerance provides features like bulkheads to achieve resource isolation and ensure critical components remain functional, even when others fail.

Why Resource Isolation Matters

In a distributed system, shared resources like thread pools, database connections, and network bandwidth can quickly become bottlenecks if not adequately managed. Resource isolation ensures: - Failures in one service do not deplete resources for other services. - Critical operations remain functional even under load or failure conditions. - Better predictability and control over system behavior.

Using Bulkheads to Isolate Resources

Bulkheads are a common pattern for isolating resources by dividing a system into separate pools or partitions. This ensures that a failure in one area does not impact others. The MicroProfile Fault Tolerance standard provides the `@Bulkhead` annotation to implement this pattern.

Bulkhead Types

MicroProfile supports two types of bulkheads:

- **Semaphore-Style Bulkhead:** Limits the number of concurrent requests.
- **Thread Pool-Style Bulkhead:** Runs a maximum number of requests on a thread pool to isolate operations.

Semaphore-Style Bulkhead

The semaphore-style bulkhead pattern limits the number of concurrent requests that can be processed by a service or method at any given time. Any additional requests are immediately rejected when the specified concurrency limit is reached. This approach prevents resource contention and protects the system from being overwhelmed during high traffic or failure scenarios.

```
package io.microprofile.tutorial.store.payment.service;

import org.eclipse.microprofile.faulttolerance.Bulkhead;
import org.eclipse.microprofile.faulttolerance.Asynchronous;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import java.util.logging.Logger;

@ApplicationScoped
public class PaymentService {
```

```
    @Inject
    private Logger logger;

    @Inject
    @ConfigProperty(name = "payment.simulatedDelay", defaultValue = "1000")
    private int simulatedDelay;

    @Inject
    @ConfigProperty(name = "payment.bulkhead.value", defaultValue = "5")
    private int bulkheadValue;

    /**
     * Processes payment transactions with limited concurrency to prevent
     * system overload and ensure stability during high traffic.
     *
     * The @Bulkhead annotation ensures that only a limited number of
     * concurrent requests can access this method.
     * The @Asynchronous annotation enables the use of the thread pool
     * style bulkhead for non-blocking execution.
     *
     * @return A success message indicating the processing status.
     */
    @Asynchronous
    @Bulkhead(value = bulkheadValue)
    public CompletionStage<String> processPayment() {
        logger.info("Starting payment processing...");
        simulateDelay();
        logger.info("Payment processing completed.");
        return CompletableFuture.completedFuture("Payment processed asynchronously.");
    }

    /**
     * Simulates a delay in processing.
     */
    private void simulateDelay() {
        try {
            Thread.sleep(simulatedDelay); // Simulating delay
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            logger.severe("Error during simulated delay: " + e.getMessage());
            throw new RuntimeException("Error during simulated delay", e);
        }
    }
}
```

In this example: - The method allows up to 5 concurrent invocations (`value = 5`). - Any additional requests are rejected to prevent overload, ensuring system stability.

Thread Pool-Style Bulkhead

The thread-pool-style bulkhead pattern leverages a thread pool to achieve resource isolation. Incoming requests are placed into a queue when the maximum allowed number of threads are in use. Queued requests are executed as threads become available. This design helps manage resource contention effectively.

```
package io.microprofile.tutorial.store.payment.service;

import org.eclipse.microprofile.faulttolerance.Bulkhead;
import jakarta.enterprise.context.ApplicationScoped;

import org.eclipse.microprofile.faulttolerance.Asynchronous;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

@ApplicationScoped
public class PaymentService {

    private static final Logger logger =
LoggerFactory.getLogger(PaymentService.class);

    /**
     * Processes payment transactions with limited concurrency using a thread pool
     * to prevent system overload and ensure stability during high traffic.
     *
     * The @Bulkhead annotation ensures that only a limited number of concurrent
     * requests (5 in this case) can access this method, and the @Asynchronous
     * annotation allows the use of the thread pool style bulkhead.
     */
    @Bulkhead(value = 5, waitingTaskQueue = 10)
    @Asynchronous
    public CompletionStage<Void> processPayment() {
        return CompletableFuture.runAsync(() -> {
            simulateDelay();
            System.out.println("Payment processed with limited concurrency.");
        }).thenRun(() -> logger.info("Payment processed with limited concurrency."));
    }

    private void simulateDelay() {
        try {
            Thread.sleep(1000); // Simulating a delay
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Error during payment processing simulation",
e);
        }
    }
}
```

In this example, The method uses up to 5 concurrent threads (`value = 5`) from a thread pool and a queue of up to 10 tasks (`waitingTaskQueue = 10`).This configuration prevents failures in one operation from depleting shared resources.

Externalizing Bulkhead Configuration

Bulkhead resource limits can be externalized using MicroProfile Config to allow runtime adjustments. For example:

Annotate the method without specific values:

```
    @Asynchronous
    @Bulkhead
    public CompletionStage<String> processPayment() {
    logger.info("Starting payment processing...");
    simulatePaymentProcessing();
    logger.info("Payment processing completed.");
    return CompletableFuture.completedFuture("Payment processed successfully with an
isolated thread pool.");
    }
```

Define bulkhead parameters in `microprofile-config.properties`:

```
com.example.Service/dynamicBulkheadOperation/Bulkhead/value=5
com.example.Service/dynamicBulkheadOperation/Bulkhead/waitingTaskQueue=10
```

Best Practices for Resource Isolation

- **Isolate Critical Resources:** Use bulkheads for high-priority operations, such as authentication, to ensure they are not impacted by failures elsewhere.
- **Monitor Usage:** Track bulkhead metrics using MicroProfile Metrics to identify bottlenecks and adjust limits.
- **Plan for Scaling:** Test bulkhead configurations under various load conditions to ensure scalability.
- **Combine with Graceful Degradation:** Pair bulkheads with fallbacks to handle rejected requests gracefully.

By effectively isolating resources, you can ensure that your microservices remain reliable and resilient, even in the face of unexpected failures or high demand. This approach not only protects critical operations but also improves overall system stability.

Summary

This chapter explored the MicroProfile Fault Tolerance API and essential fault tolerance strategies:

- **Retries:** Automatically reattempt failed operations for transient errors.

- **Timeouts:** Define maximum execution times for operations to avoid resource blocking.
- **Circuit Breakers:** Prevent repeated calls to failing services and allow graceful recovery.
- **Bulkheads:** Limit concurrent operations and isolate resource usage.
- **Fallbacks:** Provide meaningful responses during failures.

By leveraging these strategies and combining them effectively, you can design resilient microservices that gracefully handle failures, minimize disruptions, and ensure a seamless user experience.

MicroProfile Telemetry

Microservices-based applications have better scalability, flexibility, and resilience, but they suffer from additional challenges regarding availability and performance monitoring. This makes observability critical to ensure these distributed systems operate reliably.

MicroProfile Telemetry specification provides a set of vendor-neutral APIs for instrumenting, collecting, and exporting telemetry data such as traces, metrics, and logs. It is built on the foundation of OpenTelemetry from the Cloud Native Computing Foundation (CNCF) project, an open-source observability framework.

In this chapter, we will explore the fundamentals of MicroProfile Telemetry, covering topics such as tracing concepts, instrumenting Telemetry, setting up tracing providers, context propagation and correlation, analyzing traces, security considerations for tracing, and more. By the end of this chapter, you will learn how to effectively leverage distributed tracing for debugging, performance monitoring, and system optimization.

Topics to be covered

- Introduction to MicroProfile Telemetry
- Tracing Concepts
  - Spans
  - Traces
  - Context Propagation
  - Correlation
- Instrumenting OpenTelemetry
- Tools for Trace Analysis
- Exporting the Traces
- Types of Telemetry
- Agent Instrumentation
- Analyzing Traces
- Security Considerations for Tracing

Introduction to MicroProfile Telemetry

MicroProfile Telemetry addresses the operational challenges inherent in modern microservices architectures. Without proper observability, debugging, performance monitoring, and ensuring system reliability become complex and time-consuming.

Some of the key challenges in microservices-based applications include:

- **Complexity due to Distributed Architecture**: Microservices are often deployed across multiple nodes, containers, or cloud environments, making it challenging to track requests as

they move through the system. This lack of visibility increases debugging complexity, making it harder to identify bottlenecks and analyze system behavior.

- **Polyglot Architecture**: Microservices are developed using multiple programming languages (e.g., Java, Python, and Go) and frameworks, resulting in inconsistent telemetry data and a lack of standardization in observability. This fragmentation makes correlating logs, traces, and metrics across services difficult.
- **Latency**: Communication between Microservices involves latency, and all of this adds up as requests traverse several services. This makes it difficult to identify the root causes of issues. Ensuring High Availability: Failures in one microservice can affect the entire system, impacting multiple dependent microservices. This can lead to downtime or degraded performance, resulting in lost revenue and diminished user trust.

To address these challenges, MicroProfile Telemetry specification provides a standardized set of APIs for capturing telemetry data, including trace information and context propagation, to improve observability in distributed systems. By enabling seamless tracing, developers can analyze system behavior, troubleshoot service interactions, and ensure application reliability.

MicroProfile Telemetry is vendor-neutral. It allows developers to switch between different OpenTelemetry implementations without modifying their application code. This flexibility ensures that MicroProfile applications can easily integrate with various observability platforms, making it easier to adopt, scale, and maintain Telemetry in modern cloud-native environments.

Tracing Concepts

Tracing is critical for observability. It allows developers to inspect the flow of requests as they traverse through distributed systems. Tracing provides visibility into the interactions and dependencies within a system by breaking down a request into multiple spans, and connecting them into traces with context propagated across services.

Spans

A **span** is the basic unit of work in tracing. It represents a single operation or task a service performs, such as an HTTP request, a database query, or a computation. Each span contains metadata, including:

- **Operation Name**: Describes the activity (e.g., HTTP GET /products).
- **Start Time and Duration**: Captures when the operation started and how long it took.
- **Attributes**: Key-value pairs providing context (e.g., user IDs, resource names, HTTP status codes).
- **Parent Span ID**: Indicates the parent span, forming a relationship within a trace.

Spans may also include additional data like logs and events, which help provide a detailed view of the operation’s lifecycle. Spans are connected to form a trace, which helps identify bottlenecks and performance issues.

Traces

A **trace** is a collection of related spans representing the end-to-end execution of a request or transaction. It provides a holistic view of how a single request flows through the system, including service interactions. Traces often form a tree structure, where the root span represents the entry point (e.g., a user request), and child spans represent subsequent operations.

For example:

```
API Gateway (Root Span) +
 |
 ├──── Order Service (Child Span) +
 |     |
 |     ├──── Database Query (Another Child Span) +
 |     |     ├──── Fetch Order Details +
 |     |     ├──── Process Order Data +
 |     |     └──── Return Data to Order Service +
 |     |
 |     └──── Return Response to API Gateway +
 |
 └──── API Gateway Sends Final Response to User
```

Context Propagation

Context propagation refers to the mechanism of carrying trace-related metadata, such as **trace IDs** and **span IDs**, across service and thread boundaries. This ensures that all spans created during a request can be linked together to form a complete trace.

Correlation

Context propagation is vital for connecting distributed spans and understanding their relationship ensuring trace metadata remains correlated as it travels with requests across service boundaries. **Correlation** is the process of associating related spans and traces across multiple services and threads to form a cohesive view of a transaction. Correlation enables developers to:

- Identify the source of bottlenecks or errors in distributed systems.
- Understand the dependencies and interactions between services.

When viewing logs, the traceId and spanId allow you to link specific log entries to the corresponding spans in your tracing system.

- **Trace ID**: A unique identifier shared across all spans in a single trace.
- **Span ID**: A unique identifier for a single span. It is linked to a parent span, forming a hierarchy.

Together, these concepts form the foundation of distributed tracing, enabling developers to monitor, analyze, and optimize the performance of their microservices effectively.

Instrumenting Telemetry

MicroProfile Telemetry simplifies instrumentation by integrating OpenTelemetry for distributed tracing. The following steps outline how to instrument telemetry in a MicroProfile E-Commerce application.

Step 1: Add the MicroProfile Telemetry Dependency

To enable tracing and exporting of telemetry data, include the MicroProfile Telemetry API dependency in your `pom.xml` file.

```
<!-- Adding MicroProfile Telemetry dependency -->
<dependency>
   <groupId>org.eclipse.microprofile.telemetry</groupId>
   <artifactId>microprofile-telemetry-api</artifactId>
   <version>1.1</version>
   <scope>provided</scope>
</dependency>
```

Step 2: Create a Tracer

MicroProfile automatically traces requests, but you can manually instrument your code using OpenTelementry APIs.

A **Tracer** is a core component of OpenTelemetry, responsible for **creating spans** and **managing trace data** within the application. To use it, inject a Tracer instance into your MicroProfile service:

```
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.api.trace.Span;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

@ApplicationScoped
public class PaymentService {

    @Inject
    Tracer tracer;

    public void processPayment(String orderId, double amount) {
        // Create a custom span for tracing the payment process
        Span span = tracer.spanBuilder("payment.process").startSpan();

        try {
            span.setAttribute("order.id", orderId);
            span.setAttribute("payment.amount", amount);
            span.setAttribute("payment.status", "IN_PROGRESS");

            // Business logic for processing the payment
```

```
            executePayment(orderId, amount);

            span.setAttribute("payment.status", "SUCCESS");
        } catch (Exception e) {
            span.setAttribute("payment.status", "FAILED");
            span.recordException(e);
        } finally {
            span.end();
        }
    }

    private void executePayment(String orderId, double amount) {
        System.out.println("Processing payment for Order ID: " + orderId + ", Amount:
" + amount);
    }
}
```

The implementation injects a `Tracer`, which enables manual span creation and precise trace management within the application. By creating a custom span (payment.process), it captures detailed telemetry data related to the payment process. Additionally, custom attributes such as `order.id`, `payment.amount`, and `payment.status` are attached to the span, providing valuable metadata for trace analysis. The implementation also includes exception handling, ensuring that any failures encountered during payment processing are properly recorded in the trace. Finally, the span is explicitly ended, marking the completion of tracing for this method.

This setup ensures that each payment transaction is fully traceable, allowing developers to monitor execution flow, debug issues, and optimize application performance effectively.

Step 3: Create a Span

Use the Tracer to create a span that represents a specific operation or activity in your application:

```
Span span = tracer.spanBuilder("my-span").startSpan();
```

The method `spanBuilder("my-span")` creates a new named span, which represents a specific operation within the application's execution flow. This helps in tracing and monitoring the operation as part of a distributed system. Calling `startSpan()` marks the beginning of the span lifecycle, ensuring that the span is actively recorded until it is explicitly ended. This allows telemetry data to be captured for performance analysis, debugging, and observability.

Step 4: Add Attributes to the Span

Attributes enhance trace context by attaching key-value pairs to a span, providing additional metadata that helps filter and analyze traces in observability tools. This helps in contextualizing the trace data:

```
span.setAttribute("http.method", "GET");
span.setAttribute("http.url", "/products/12345");
```

```
span.setAttribute("user.id", "98765");
```

The above statements allow the tracing system to capture essential details about an HTTP request.

Step 5: End the Span

When the operation completes, end the span to capture the telemetry data:

```
Span span = tracer.spanBuilder("payment.process").startSpan();

try {
    // Business logic execution
} catch (Exception e) {
    span.recordException(e);
    span.setAttribute("error", true);
} finally {
    span.end();
}
```

Tools for Trace Analysis

The following tools are commonly used for trace collection, visualization, and analysis in MicroProfile applications:

OpenTelemetry Collector

The OpenTelemetry Collector is an open-source telemetry processing system that acts as an intermediary between instrumented applications and observability backends such as Jaeger, Zipkin, and Prometheus. It is designed to receive, process, and export tracing data, making it a powerful tool for managing distributed traces in MicroProfile applications.

It is vendor-agnostic, which allows for seamless integration with multiple tracing backends without requiring any changes to application instrumentation. It supports multiple data formats, enabling the ingestion of traces through several protocols, ensuring compatibility across different telemetry sources. Additionally, it offers processing pipelines that let developers filter, batch, and transform trace data before exporting it, optimizing observability workflows.

Designed for scalability, the OpenTelemetry Collector can be deployed as a standalone instance or distributed across multiple nodes, making it suitable for both small-scale applications and large enterprise-grade distributed systems.

Jaeger

Jaeger is an open-source distributed tracing system developed by Uber, widely used for monitoring microservices and visualizing request flows in cloud-native applications. It provides a powerful visualization interface that enables developers to inspect traces, analyze dependencies between services, and examine execution timelines, making it an essential tool for debugging performance

bottlenecks.

One of Jaeger's key capabilities is service dependency analysis, which helps identify how microservices interact, providing insights into latency, failures, and request propagation. It also supports adaptive sampling strategies, allowing developers to control the volume of traces collected to optimize performance without overwhelming storage and processing resources. Additionally, Jaeger offers built-in storage options, allowing trace data to be stored in Elasticsearch, Cassandra, or Kafka, making it scalable and flexible for various deployment environments.

Zipkin

Zipkin is a distributed tracing system designed to help developers visualize and diagnose latency issues in microservices-based applications. It provides a lightweight and fast tracing solution, making it ideal for quick deployment with minimal resource usage. Its simplicity and efficiency make it a popular choice for teams looking to implement tracing without significant infrastructure overhead.

One of Zipkin's core strengths is its tag-based searching, which allows developers to filter traces based on metadata such as service name, request ID, or other custom attributes, enabling quick identification of relevant traces. It also offers dependency graph visualization, helping to uncover bottlenecks and inefficiencies in microservices interactions. To accommodate different storage needs, Zipkin supports multiple storage backends, including Elasticsearch, MySQL, and Cassandra, providing flexibility for various deployment scenarios.

Grafana Tempo

Grafana Tempo is a distributed tracing backend. Unlike Jaeger and Zipkin, Tempo does not require indexing as it only requires object storage, making it highly scalable and cost-efficient for handling large volumes of trace data. This unique approach allows Tempo to store traces efficiently without increasing storage and query overhead, making it an ideal choice for high-performance microservices environments. One of Tempo's key advantages is its tight integration with Grafana dashboards, enabling developers to correlate logs, metrics, and traces within a unified observability platform. Additionally, Tempo offers multi-backend support, meaning it can ingest and process trace data from OpenTelemetry, Jaeger, and Zipkin sources, ensuring compatibility with existing tracing setups. Its scalability makes it well-suited for large-scale microservices architectures, where efficiently managing distributed tracing data is crucial.

Exporting the Traces

To export the traces we need to configure the exporter type and endpoint in the `src/main/resources/META-INF/microprofile-config.properties`. For using OTLP (OpenTelemetry Protocol) export, you need to add the following configuration in:

```
# Enable OpenTelemetry
otel.traces.exporter=otlp

# Set the OTLP exporter endpoint
otel.exporter.otlp.endpoint=http://localhost:4317
```

```
# Define the service name
otel.service.name=payment-service

# Sampling rate: (1.0 = always, 0.5 = 50%, 0.0 = never)
otel.traces.sampler=parentbased_always_on
```

This sends traces directly to a observability tool, enabling real-time distributed tracing and performance monitoring. To ensure proper tracing, your observability tool (for e.g. Jaeger) must be running to receive trace data.

Using OTLP is advantageous because it is the native standard for OpenTelemetry, ensuring seamless integration with a wide range of observability tools. One of its key benefits is that it allows developers to use multiple observability platforms without changing instrumentation, providing a unified and vendor-neutral tracing solution.

Verify the Traces

Once tracing is enabled and the appropriate exporter is configured, the next step is to verify that traces are being captured and sent to the observability backend. This ensures that the MicroProfile Telemetry setup is functioning correctly and that distributed tracing data is available for monitoring and debugging.

Run Jaeger

The simplest way to run Jaeger is with Docker using the command as below:

```
docker run -d --name jaeger \
  -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  -p 14250:14250 \
  -p 9411:9411 \
  jaegertracing/all-in-one:latest
```

The above command runs the **all-in-one** Jaeger container, which includes the agent, collector, query service, and UI.

The Jaeger UI can be accessed at: `https://<hostname>:16686`.

Ensure all the services of our MicroProfile E-commerce applications are running.

Search using parameters like operation name, time range, or service for the traces associated with different microservices and confirm that the telemetry data is visible. View a detailed breakdown of each span within the trace, including timing and attributes.

Types of Telemetry

MicroProfile Telemetry supports multiple approaches to instrumentation and tracing, ensuring flexibility for developers based on their observability needs. The three primary types of telemetry in MicroProfile Telemetry are:

Automatic Instrumentation

Automatic Instrumentation enables distributed tracing without requiring any modifications to the application code. This is particularly beneficial for Jakarta RESTful Web Services and MicroProfile REST Clients, as it enables seamless integration into distributed tracing systems following the semantic conventions of OpenTelemetry. This ensures compatibility across different tracing tools.

For example, in the ProductService, which exposes a RESTful endpoint, automatic instrumentation ensures that incoming and outgoing HTTP requests are traced with minimal configuration, without requiring any additional code changes.

By default, MicroProfile Telemetry tracing is disabled. To activate it, set the following property in `microprofile-config.properties`:

```
otel.sdk.disabled=false
```

This ensures that OpenTelemetry's tracing capabilities are enabled for the application.

Manual Instrumentation

Manual Instrumentation provides developers with fine-grained control over how telemetry data is collected and structured within a MicroProfile application. By explicitly defining spans, attributes, and trace propagation, developers can gain greater insight into application behavior beyond what automatic instrumentation provides.

Using the @WithSpan Annotation

The `@WithSpan` annotation provides a simple way to create custom spans within a trace. By annotating a method with `@WithSpan`, a new span is automatically generated whenever the method is invoked. This span is linked to the current trace context, allowing developers to track key operations without manually managing span lifecycle.

```
import io.opentelemetry.instrumentation.annotations.WithSpan;
import jakarta.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class PaymentService {

    @WithSpan
    public void processPayment(String orderId) {
        // Business logic here
    }
```

```
}
```

Every time processPayment is called, a new span is created. The span is automatically linked to the current trace context. No need for explicit span creation or lifecycle management. You can use `@WithSpan` for tracing key business operations, such as order processing, payment handling, or API requests.

Using `SpanBuilder` for Custom Spans

For greater flexibility, developers can manually create spans using the OpenTelemetry API. The `SpanBuilder` class provides the ability to define custom span names, making trace analysis more meaningful and structured. Additionally, developers can attach custom attributes to spans, enriching trace data with relevant metadata for deeper insights. This method also offers explicit control over the span lifecycle, allowing spans to be started and ended manually, ensuring they accurately represent specific business operations or execution flows within the application.

```
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.api.trace.Span;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

@Path("/trace")
public class TraceResource {

    @Inject
    Tracer tracer;

    @GET
    @Path("/custom")
    public String customTrace() {
        Span span = tracer.spanBuilder("custom-span").startSpan();
        span.setAttribute("custom.key", "customValue");
        span.end();
        return "Trace recorded";
    }
}
```

The method `tracer.spanBuilder("custom-span").startSpan()` creates a span with a specific name allowing developers to define meaningful trace segments for better observability. Using `span.setAttribute("custom.key", "customValue")`, custom metadata can be attached to the span, enriching trace data with relevant contextual information. Finally, calling `span.end()` explicitly marks the completion of the span, ensuring accurate tracking of execution duration. The `SpanBuilder` approach is particularly useful when developers require fine-grained control over when spans start and end, as well as the ability to include detailed metadata for enhanced trace analysis.

Manual Tracing in `PaymentService`

To manually instrument the processPayment method in the PaymentService, we use OpenTelemetry's API to create a custom span, add attributes, and control the span lifecycle.

```
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.Tracer;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

@ApplicationScoped
public class PaymentService {

    @Inject
    Tracer tracer;

    public void processPayment(String orderId, double amount, String paymentMethod) {
        // Create a custom span for tracing the payment process
        Span span = tracer.spanBuilder("payment.process").startSpan();

        try {
            // Add attributes to enrich the trace
            span.setAttribute("order.id", orderId);
            span.setAttribute("payment.amount", amount);
            span.setAttribute("payment.method", paymentMethod);
            span.setAttribute("payment.status", "IN_PROGRESS");

            // Business logic for processing the payment
            System.out.println(Processing Payment…);

            // Update span attribute on successful completion
            span.setAttribute("payment.status", "SUCCESS");
        } catch (Exception e) {
            // Capture error in tracing
            span.setAttribute("payment.status", "FAILED");
            span.recordException(e);
        } finally {
            // End the span to complete the tracing
            span.end();
        }
    }
}
```

The `payment.process` span is manually created using `tracer.spanBuilder()`, allowing explicit control over the tracing of the payment process. To enhance trace visibility, custom attributes such as the order ID, payment amount, and payment method are attached to the span, providing valuable context for analysis. Additionally, the payment status is recorded as `IN_PROGRESS` when processing starts and updated to `SUCCESS` or `FAILED` based on the outcome.

In the event of an error, the span captures and records the exception, ensuring failure details are

logged for debugging. The span lifecycle is carefully managed, starting before the business logic executes and ending only after the process is completed in the `finally` block. This structured approach guarantees accurate performance monitoring and trace completeness, improving visibility into how payments are processed in a distributed system.

Agent Instrumentation

Agent Instrumentation enables telemetry data collection without modifying application code by attaching a Java agent at runtime. This approach is particularly useful for legacy applications or scenarios where modifying source code is not feasible. The OpenTelemetry Java Agent dynamically instruments applications, automatically detecting and tracing interactions within commonly used frameworks such as Jakarta RESTful Web Services, database connections, and messaging systems.

One of the key advantages of agent-based instrumentation is that it requires no changes to the application's source code and eliminates the need for recompilation or redeployment. Instead, it can be activated by attaching the agent at application startup.

Refer to the OpenTelemetry Java Agent Getting Started page for step-by-step instructions on enabling it for your application. Once enabled, the agent automatically instruments the application, seamlessly integrating with distributed tracing systems without requiring developer intervention. This makes it an efficient and non-intrusive way to implement observability in MicroProfile applications.

Once enabled, the agent automatically instruments the application, seamlessly integrating with distributed tracing systems without requiring developer intervention. This makes it an efficient and non-intrusive way to implement observability in MicroProfile applications.

Analyzing Traces

Once trace data is collected and exported to a backend system, analyzing these traces becomes a crucial step in understanding the behavior of your distributed microservices architecture. By examining traces, you can gain insights into system performance, identify bottlenecks, and detect failures or anomalies.

Visualizing Traces

Tracing backends like **Jaeger**, **Zipkin**, or **Graphana Tempo** provide visual interfaces to explore and analyze traces. These tools display traces as timelines or dependency graphs, making it easier to:

- Understand the sequence of operations.
- Identify the services and components involved in a request.
- Observe how requests propagate through the system.

Identifying Bottlenecks

Traces highlight spans with long durations or repeated retries, which often point to bottlenecks or inefficiencies. Pay close attention to:

- **Critical Path**: The longest path in a trace that determines the total response time.
- **Service Dependencies**: Examine how upstream and downstream services interact to find slow components.
- **Retries and Failures**: Repeated spans or high failure rates indicate problematic dependencies or transient errors.

Diagnosing Failures

Traces provide valuable information for diagnosing failures, including:

- **Error Codes**: Look for spans with error attributes, such as `http.status_code=500`.
- **Exception Details**: Many tracing systems capture stack traces or error messages in spans.
- **Service Impact**: Identify which upstream and downstream services are affected by the failure.

Understanding Service Dependencies

Dependency graphs generated from traces show the interactions between services. These graphs help:

- Visualize which services depend on each other.
- Detects circular dependencies or excessive coupling.
- Plan optimizations by focusing on critical services.

Correlating Traces with Logs and Metrics

Traces, when combined with logs and metrics, provide a comprehensive picture of the system:

- **Logs**: Use trace IDs and span IDs in logs to correlate application logs with specific spans.
- **Metrics**: Correlate trace performance data with system metrics like CPU usage, memory consumption, or request rates. Example: If a span indicates high latency, check corresponding logs and metrics to identify the underlying cause, such as a resource constraint or network delay.

Best Practices for Analyzing Traces

1. **Establish Baselines**: Use traces to establish performance baselines for services.
2. **Monitor Critical Paths**: Focus on traces that traverse critical services or user-facing operations.
3. **Use Sampling Strategically**: Balance trace volume and storage costs by sampling traces intelligently.
4. **Automate Alerts**: Set up alerts for abnormal patterns in traces, such as increased latency or failure rates.
5. **Collaborate Across Teams**: Share trace insights with development, operations, and QA teams to improve system reliability.

By analyzing traces effectively, you can identify opportunities to optimize your microservices,

ensure smoother operations, and enhance the overall user experience. Tracing tools provide a powerful way to visualize and understand the intricate dynamics of distributed systems.
When analyzing traces, developers should look for the following:

- **Long spans:** Spans that take a long time to complete may indicate a performance issue.
- **Missing spans:** Missing spans can make it difficult to understand the flow of a request.
- **Errors:** Errors can indicate problems with a service or a request.
- **High latency:** High latency can indicate a problem with the network or a service.

By analyzing traces, developers can identify and troubleshoot problems with their microservices applications. This can help developers improve the performance and reliability of their applications.

Here are some tips for analyzing traces:

- **Use a trace viewer:** A trace viewer is a tool that can help you visualize and analyze traces.
- **Look for patterns:** Look for patterns in the traces that may indicate a problem.
- **Correlate traces with metrics:** Correlate traces with metrics to get a better understanding of the performance of your application.
- **Use sampling:** Use sampling to reduce the number of traces that are collected. This can improve the performance of your tracing system.

By following these tips, developers can effectively analyze traces to improve the performance and reliability of their microservices applications.

Security Considerations for Tracing

When implementing tracing in your applications, it is crucial to be mindful of security implications. Tracing involves collecting and storing data about application behavior, which can potentially expose sensitive information if not handled properly.

- **Data Sensitivity:** Be cautious about the data included in traces. Avoid logging sensitive information such as passwords, API keys, or personally identifiable information (PII).
- **Access Control:** Implement strict access controls to limit who can view and manage trace data.
- **Encryption:** Consider encrypting trace data at rest and in transit to protect it from unauthorized access.
- **Storage:** Carefully manage the storage of trace data. Avoid storing traces indefinitely and implement data retention policies.
- **Third-Party Services:** If using third-party tracing services, ensure they have robust security measures in place to protect your data.

Avoid Capturing Sensitive Data

Traces often include attributes and metadata that can contain sensitive information. Avoid storing or transmitting sensitive details, such as:

- Personally Identifiable Information (PII) (e.g., names, addresses, social security numbers).
- Payment information (e.g., credit card numbers).
- Authentication credentials (e.g., passwords, API keys, tokens).

Best Practice:

Sanitize attributes before adding them to spans:

```
span.setAttribute("user.id", "anonymized-user-id");
span.setAttribute("credit.card.last4", "****1234");
```

Encrypt Trace Data

To prevent unauthorized access during transmission, ensure that telemetry data is encrypted. Use secure protocols such as HTTPS or TLS for exporting trace data to a backend.

```
*Example:*
```

- Configure the tracing provider to use encrypted connections:

```
otel.exporter.jaeger.endpoint=https://secure-jaeger-collector.example.com
otel.exporter.otlp.endpoint=https://secure-collector.example.com
```

Limit Trace Retention

Trace data can grow rapidly in distributed systems. Retaining it indefinitely increases the risk of exposing sensitive information. Implement retention policies to:

- Retain traces only for the necessary duration for debugging or performance analysis.
- Periodically purge older traces from storage.

Access Control and Auditing

Restrict access to trace data to authorized personnel only. Ensure that your tracing backend implements robust authentication and authorization mechanisms.

Best Practice:

- Use role-based access control (RBAC) to define permissions for viewing and managing traces.
- Audit access to trace data regularly to identify potential misuse or breaches.

Sampling Strategies to Minimize Exposure

Sampling reduces the volume of traces collected and limits the exposure of sensitive data by capturing only a subset of requests. Common strategies include:

- Random Sampling: Captures a fixed percentage of traces.
- Rate-Limiting Sampling: Limits the number of traces per second.
- Key-Based Sampling: Samples traces based on specific attributes (e.g., user ID).

Example:

Random sampling to limiting the amount of trace data collected:

```
otel.traces.sampler=traceidratio
otel.traces.sampler.traceidratio=0.1
```

Compliance with Regulations

Ensure that your tracing practices comply with data protection and privacy regulations such as GDPR, CCPA, or HIPAA. Key considerations include:

- Anonymizing sensitive data before tracing.
- Informing users about telemetry collection in your privacy policy.
- Providing mechanisms to opt out of tracing where required.

Isolate Tracing Infrastructure

The tracing infrastructure, such as Jaeger or OpenTelemetry Collector, should be isolated from the public internet and accessible only within secure networks.

Best Practice:

- Deploy tracing backends in private subnets or behind firewalls.
- Use VPNs or dedicated connections for remote access to tracing dashboards.

Monitor and Alert on Trace Anomalies

Tracing can help detect potential security incidents. Monitor traces for unusual patterns, such as:

- Unexpected spikes in requests.
- Requests from unknown or unauthorized sources.
- Abnormal response times indicating possible exploits. Set up alerts for these anomalies to investigate and mitigate potential issues.
  By following these security considerations, you can leverage the benefits of distributed tracing without compromising the security of your system or the privacy of your users. Careful handling of trace data, coupled with robust encryption, access controls, and compliance practices, ensures that tracing remains a valuable yet secure component of your observability strategy.

Conclusion

MicroProfile Telemetry provides a robust foundation for observability in Java-based microservices, enabling developers to implement distributed tracing seamlessly. By leveraging this specification, you can gain deep insights into the flow of requests, identify bottlenecks, and enhance the reliability and performance of your applications. The integration of standardized tracing concepts like spans, traces, and context propagation ensures that developers can maintain a cohesive understanding of their system's behavior across service boundaries.

Through instrumentation, context propagation, and effective trace analysis, MicroProfile Telemetry simplifies the complexities of monitoring and debugging distributed systems. It empowers teams to proactively address issues, optimize performance, and improve the user experience. Moreover, by adhering to security best practices, developers can ensure that telemetry data is protected, compliant with regulations, and free of sensitive information.

In this chapter, we explored the critical security considerations surrounding tracing within the MicroProfile Telemetry framework. We emphasized the importance of safeguarding sensitive data by avoiding the inclusion of Personally Identifiable Information (PII) in trace spans. Additionally, we discussed the potential security risks associated with tracing in production environments and the significance of carefully managing sampling rates and data retention policies. By adhering to these security best practices, developers can harness the power of tracing for observability while ensuring the confidentiality and integrity of their applications.

As microservices architectures continue to evolve, the ability to observe and trace system interactions will remain a critical factor in maintaining resilient and efficient applications. MicroProfile Telemetry stands as a valuable tool in achieving these goals, providing developers with the observability they need to deliver reliable, high-performance microservices in modern cloud-native environments.

MicroProfile JWT

In modern microservices architectures, where services are distributed and stateless, securing communications between clients and services and between individual services is critical. **JSON Web Token (JWT)** provides a lightweight, self-contained, and efficient user authentication and authorization mechanism, enabling scalable and secure identity propagation across distributed systems.

MicroProfile JWT is a specification that standardizes JWT-based authentication and authorization for Java microservices. Leveraging the JWT open standard RFC 7519 enables services to securely extract and validate claims such as identity, roles, and permissions.

MicroProfile JWT allows developers to build secure, interoperable, and portable microservices. It supports **role-based access control (RBAC)**, simplifies identity management in stateless services, and avoids vendor lock-in by adhering to open specifications.

Topics to be covered:

- Introduction to JWT Authentication
- Structure of JWT
- Use cases for JSON Web Tokens
- Benefits of JWT in Microservices
- Setting up MicroProfile JWT
- Configuring MicroProfile JWT Validation
- Request Flow in MicroProfile JWT
- Role-Based Access Control (RBAC)
- Setting Token Expiry Times for Security
- Error Handling
- Best Practices for JWT Authentication
- Security Best Practices for Microservices
- Conclusion

Introduction to JWT Authentication

This section will explore JSON Web Tokens, how they work, and why they are foundational to implementing stateless authentication and authorization in microservices-based systems.

What is a JSON Web Token (JWT)?

A **JSON Web Token (JWT)** (see JWT.io), as defined in RFC 7519, is an open standard for securely transmitting information (claims) between parties as a JSON object. JWTs are digitally signed, ensuring their integrity and authenticity.

Structure of a JWT

A JWT consists of three Base64 encoded parts, separated by dots (.):

```
<Header>.<Payload>.<Signature>
```

- **Header** - It contains metadata about the token, such as token type (type: “JWT”) and signing algorithm (alg: “RS256” for RSA-SHA256).

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

- **Payload** - It contains claims which are key-value pairs representing data about the user, such as roles and expiration. Example of claims in a JWT payload:

```
{
  "iss": "https://io.microprofile.com/issuer",
  "sub": "user1",
  "exp": 1735689600,
  "iat": 1735686000,
  "aud": "my-audience",
  "groups": ["user", "admin"]
}
```

- **Signature** — A digital signature that verifies the token’s integrity by combining the encoded header, payload, and private key.

Example JWT Token:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
QR1Owv2ug2WyPBnbQrRARTeEk9kDO2w8qDcjiHnSJflSdv1iNqhWXaKH4MqAkQtM
oNfABIPJaZm0HaA415sv3aeuBWnD8J-Ui7Ah6cWafs3ZwwFKDFUUsWHSK-IPKxLG
TkND09XyjORj_CHAgOPJ-Sd8ONQRnJvWn_hXV1BNMHzUjPyYwEsRhDhzjAD26ima
sOTsgruobpYGoQcXUwFDn7moXPRfDE8-NoQX7N7ZYMmpUDkR-Cx9obNGwJQ3nM52
YCitxoQVPzjbl7WBuB7AohdBoZOdZ24WlN1lVIeh8v1K4krB8xgKvRU8kgFrEn_a
1rZgN5TiysnmzTROF869lQ.
AxY8DCtDaGlsbGljb3RoZQ.
MKOle7UQrG6nSxTLX6Mqwt0orbHvAKeWnDYvpIAeZ72deHxz3roJDXQyhxx0wKaM
HDjUEOKIwrtkHthpqEanSBNYHZgmNOV7sln1Eu9g3J8.
fiK51VwhsxJ-siBMR-YFiA
```

Types of Claims in MicroProfile JWT

Claims in JWTs can be categorized into two types:

Standard Claims

These are predefined claims with specific meanings, as defined by the JWT specification. Some commonly used standard claims include:

Claim	Description	Example
`iss`	Issuer, the entity that issued the JWT (e.g., an authentication server)	`"iss": "https://io.microprofile.com/issuer"`
`sub`	Subject, the principal (user or service) that the JWT is about.	`"sub": "user1"`
`aud`	Audience, the intended recipients of the token (e.g., specific microservices).	`"aud": "order-service"`
`exp`	Expiration time	`"exp": 1735689600`
`nbf`	not before	`"nbf": 1735686000`
`iat`	issued at, when that token was issued.	`"iat": 1735686000`
`jti`	Unique JWT token identifier	`"jti": "a1b2c3d4"`
`groups`	Groups, list of roles or users allowed to access the resource	`["user", "admin"]`

Custom Claims

These are application-specific claims that provide additional information about the user or entity. They can extend authorization logic with application-specific claims (e.g., `department`, `region`). Custom claims are not part of the JWT specification but often include domain-specific data, such as user preferences, tenant IDs, or other metadata. MicroProfile JWT allows developers to access these claims programmatically.

Use cases for JSON Web Tokens

JWTs are versatile tokens commonly used in modern applications for authentication, where they verify the identity of a user or service; for authorization, where they grant access to resources based on roles or permissions; and for information exchange, where they securely transmit data between parties.

Below are key scenarios where JWTs shine in microservices environments:

Authentication

JWTs enable stateless authentication in distributed systems. When a user logs in, an authentication service issues a JWT containing claims like sub (user ID) and exp (expiration time). The client sends this token in the `Authorization: Bearer` header of subsequent requests, allowing microservices to verify the user's identity without requiring repeated authentication.

For example, a user authenticates with an Auth Service and receives a JWT. This JWT token grants access to other services, such as a product catalog or order management system, without re-authentication.

Authorization (Role-Based Access Control)

JWTs are also used for authorization, enabling fine-grained access control based on user roles or permissions. The JWT payload typically includes a group or role claim specifying the user's roles or permissions. For example, a user with the admin role might be allowed to access all resources while a user with the user role might only have access to specific resources.

MicroProfile JWT integrates seamlessly with Jakarta EE's `@RolesAllowed` annotation, making it easy to enforce role-based access control (RBAC) in microservices. Role mapping can be configured in *microprofile-config.properties*:

```
mp.jwt.verify.roles=groups
```

Claims-based identity

JWTs are often used to represent claims-based identity, where the JWT contains claims representing the user's identity, such as their name, email address, or other attributes. Applications can use these claims to identify the user and personalize their experience.

For example, an application might use the email claim to look up the user's profile information in a database or display the user's name on a welcome page using the name claim.

Information Exchange

JWTs can securely exchange information between parties. The token payload can include custom claims representing the data being exchanged, such as an order ID or user ID. This makes JWTs useful in scenarios like Single Sign-On (SSO) systems, where information needs to be shared across multiple services.

For example, a JWT might contain an `order_id` claim and a `user_id` claim, which an order management service can use to retrieve and display the user's order details.

Federation & Single Sign-On (SSO)

JWTs facilitate identity federation by allowing integration of multiple trusted identity providers (e.g., Active Directory, LDAP) to provide a single sign-on (SSO) experience. In this case, the JWT contains claims representing the user's identity, which applications can use to identify the user and retrieve their profile information.

For example, an enterprise SSO system can issue a JWT that grants access to HR, Payroll, and CRM microservices. MicroProfile JWT validates the token's iss (issuer) and aud (audience) to enforce trust boundaries.

Benefits of using JWT in Microservices

JWTs are widely used in microservices for the following reasons:

Statelessness & Scalability

JWTs eliminate the need for centralized session storage. Each token is self-contained, embedding all necessary user claims (e.g., roles, permissions) in its payload.

Independent Validation: Microservices validate JWTs locally using public keys, avoiding calls to a central authority. This reduces latency and scales horizontally.

Example: A payment service validates a JWT's signature without querying an authentication server.

Interoperability

Open Standards: JWTs adhere to RFC 7519, ensuring compatibility across platforms (Java, .NET, Node.js) and frameworks (Spring Boot, Quarkus).

MicroProfile Integration: MicroProfile JWT standardizes validation and claim extraction, enabling seamless interoperability across Java microservices.

Fine-Grained Authorization

Role-Based Access Control (RBAC): Map JWT claims (e.g., groups) to Jakarta EE roles using @RolesAllowed.

Decentralized Security

Propagation Across Services: A JWT issued by an authentication service is propagated across microservices (e.g., the Order Service and the Inventory Service). Each service independently verifies the token and enforces access control.

Reduced Central Dependency: No need for a central authorization server, simplifying architecture and improving fault tolerance.

Example:

- Authentication Service: Issues a JWT with `sub: "user1"` and `groups: ["user"]`.
- Order Service: Validates the JWT and processes requests if groups include `users`.
- Inventory Service: Revalidates the same JWT without contacting the auth service.

Setting Up MicroProfile JWT

To use MicroProfile JWT in your project, add the following dependency to your *pom.xml* (for Maven):

```
<dependency>
```

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

For Gradle, add the following to your *build.gradle*:

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

Configuring MicroProfile JWT Validation

MicroProfile JWT requires validation rules configuration to be defined in `src/main/resources/microprofile-config.properties` file. Below is an example configuration:

```
# Public key (PEM format) to verify JWT signatures
mp.jwt.verify.publickey.location=META-INF/publicKey.pem

# Expected issuer (e.g., your OIDC provider)
mp.jwt.verify.issuer=https://auth.example.com

# Optional: Validate token audience
mp.jwt.verify.audiences=order-service,payment-service
```

Explanation:

- The `mp.jwt.verify.publickey.location` property specifies the location of the public key used to verify the JWT's signature.
- The `mp.jwt.verify.issuer` property defines the expected issuer of the JWT, ensuring that tokens are only accepted if issued by a trusted authority.
- Optionally, the `mp.jwt.verify.audiences` property can specify the allowed audiences for the JWT, ensuring that the token is intended for the service.

Public Key Setup

Place the PEM-encoded public key in *src/main/resources/META-INF/publicKey.pem*. This key is used to verify incoming JWT signatures.

Request Flow in MicroProfile JWT

Understanding how JWTs are propagated and processed in a microservices architecture is critical to implementing secure and scalable authentication. This section explains the lifecycle of a JWT from client to service, including token extraction, validation, and claim usage.

How JWTs are Propagated in Microservices

JWTs are propagated via the `Authorization: Bearer` HTTP header across clients and services.

Client-to-Service

When a client authenticates (e.g., via a login endpoint), it receives a JWT from an authentication service. This token is then included in the header of subsequent requests to microservices. For example, a request header might look like this:

```
GET /api/orders HTTP/1.1
Authorization: Bearer eyJhbGciOiJSUzI1NiIs···
```

Service-to-Service

In microservices architecture, a client sends a JWT token to the initial service (for example, Order Service) using the `Authorization: Bearer` header. The initial service can forward the same token when calling another backend service (for example, the Inventory Service). Each microservice independently validates the JWT to enforce decentralized, stateless security.

In advanced scenarios involving multiple downstream services, careful consideration must be given to validating the JWT's `aud` (audience) claim to ensure the token is intended for the target service.

Token Extraction

MicroProfile JWT runtime handles token extraction and validation automatically. The token is parsed and validated as follows:

- Header Parsing: The runtime extracts the token from the Bearer schema.
- Decoding: The JWT is split into its header, payload, and signature components.

Token Validation

The token validation involves the following steps:

- Signature Verification: The public key validates the token's integrity.
- Standard Claims Validation: The runtime then validates standard claims:
  1. `iss`: It should match the `mp.jwt.verify.issuer` configuration property.
  2. `exp` : This checks if the token has not expired.
  3. `aud` : Optionally it checks for the included service(s) in `mp.jwt.verify.audiences`.

If valid, the JWT's claims populate the `SecurityContext`. Otherwise, MicroProfile JWT rejects the request with a `401 Unauthorized` status.

Accessing JWT claims via `SecurityContext`

The `SecurityContext` interface (from Jakarta EE) provides programmatic access to JWT claims. Once a token is validated, MicroProfile JWT injects the `JsonWebToken` into the `SecurityContext`, allowing developers to:

- Retrieve user identity (e.g., `sub` claim).
- Check user roles (e.g., `groups` claim).
- Access custom claims (e.g., `tenant_id` claim).

```
@GET
@Path("/user-profile")
public String getUserProfile(@Context SecurityContext ctx) {
    JsonWebToken jwt = (JsonWebToken) ctx.getUserPrincipal();
    String userId = jwt.getName(); // Extracts the "sub" claim
    Set<String> roles = jwt.getGroups(); // Extracts the "groups" claim
    String tenant = jwt.getClaim("tenant_id"); // Custom claim

    return "User: " + userId + ", Roles: " + roles + ", Tenant: " + tenant;
}
```

The `SecurityContext` simplifies working with JWTs, enabling seamless integration with Jakarta EE's security annotations like `@RolesAllowed`. By calling `securityContext.getUserPrincipal()`, the application can obtain the `JsonWebToken` instance, which contains all the claims from the JWT.

Role-Based Access Control (RBAC)

MicroProfile JWT simplifies RBAC by mapping JWT claims (e.g., `groups` or `roles`) to Jakarta EE roles. This enables declarative security using the `@RolesAllowed` annotation. This section explains how to configure and use this mapping effectively.

Default Role Mapping with the `groups` Claim

MicroProfile JWT seamlessly integrates with Jakarta EE's `@RolesAllowed` annotation to enforce role-based access control in microservices. By default, MicroProfile JWT maps roles from the groups claim in the JWT payload to Jakarta EE roles. The groups claim is a standard JWT claim that represents the roles or groups assigned to the user. For example, a JWT payload might include:

```
{
  "iss": "https://example.com/issuer",
  "sub": "user123",
  "groups": ["user", "admin"]
}
```

In this case, the user has two roles: user and admin.

Securing Endpoints

The roles in the groups claim can be used directly with the `@RolesAllowed` annotation to secure endpoints.

```
@Path("/orders")
public class OrderResource {

  @GET
  @Path("/{id}")
  @RolesAllowed("user") // Only users can access this method
  public Response getOrder(@PathParam("id") String id, @Context SecurityContext ctx) {
    String user = ctx.getUserPrincipal().getName();
    // Fetch order for the user
    return Response.ok("Order for user: " + user + ", ID: " + id).build();
  }

  @DELETE
  @Path("/{id}")
  @RolesAllowed("admin") // Only admins can access this method
  public Response deleteOrder(@PathParam("id") String id, @Context SecurityContext
ctx) {
    String admin = ctx.getUserPrincipal().getName();
    // Delete order as admin
    return Response.ok("Order deleted by admin: " + admin + ", ID: " + id).build();
  }
}
```

The `GET /orders/chapter10` service is accessible to users, whereas the `DELETE /orders/chapter10` is only available to users with the admin role.

Custom Role Mapping

If your JWT uses a claim other than groups to represent roles (e.g., roles or scopes), you can customize the mapping using the `mp.jwt.verify.roles` property in *microprofile-config.properties*:

```
# Optional: Map roles from the "groups" claim (default behavior)
mp.jwt.verify.roles=groups
```

The `groups` claim is the default claim used for role mapping in MicroProfile JWT Authentication. Therefore, you typically do not need to set the `mp.jwt.verify.roles` property unless your JWT uses a different claim name. For example, some identity providers (like OAuth 2.0 servers or OpenID Connect providers) might include roles in claims such as `roles`, `permissions`, or `scope` instead of `groups`.

In such cases, update the mapping in your *microprofile-config.properties* file:

```
mp.jwt.verify.roles=roles
```

This ensures that MicroProfile JWT Authentication correctly maps the roles for use with Jakarta EE security annotations like `@RolesAllowed`.

How the RBAC Works

- Token Validation: MicroProfile JWT validates the JWT's signature and claims.
- Role Extraction: Roles are extracted from the configured claim (groups by default).
- Access Control: The `@RolesAllowed` annotation checks if the user's roles match the required roles. If not, a `403 Forbidden` response is returned.

This approach ensures fine-grained security while maintaining compatibility with standard JWT practices.

Setting Token Expiry Times for Security

Short token expiry times reduce the surface area for the attackers. Here's how to configure token expiry effectively:

Configuring Token Expiry

Set the `exp` claim at issuance: Ensure your authentication service issues tokens with the `exp` claim.

```
{
  "exp": 1735689600 // Token expires at 2025-01-01 00:00:00 UTC
}
```

MicroProfile JWT automatically validates the `exp` claim during token verification. Beyond standard JWT validation settings, no additional configuration is needed.

MicroProfile JWT will reject tokens returning a 401 Unauthorized response if:

- The `exp` claim is missing or invalid.
- The current time exceeds the `exp` value.

Error Handling

MicroProfile JWT automatically validates tokens and rejects invalid requests with standardized HTTP responses. Common scenarios include:

Invalid Token (e.g., malformed JWT, invalid signature):

```
HTTP/1.1 401 Unauthorized
```

```
WWW-Authenticate: Bearer error="invalid_token"
```

Expired Token (exp claim validation failure):

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error="invalid_token", error_description="Token expired"
```

Missing Token

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error="missing_token"
```

Insufficient Permissions (e.g., missing role for @RolesAllowed):

```
HTTP/1.1 403 Forbidden
```

Best Practices for JWT Authentication

1. Use Standard Claims - Prefer the groups claim for roles unless your identity provider uses a different claim.
2. Consistent Role Names - Ensure role names (e.g., admin, user) are consistent across JWTs and @RolesAllowed annotations.
3. Least Privilege - Assign minimal required roles to endpoints to reduce security risks.
4. Combine with Other Annotations - Use @PermitAll or @DenyAll alongside @RolesAllowed for flexible security policies.

Security Best Practices for Microservices

But with more services comes more complexity, and with more complexity comes a greater risk of security breaches. So, how do you secure your microservices?

Securing microservices requires a layered approach, combining authentication, authorization, encryption, and monitoring. MicroProfile JWT simplifies access control while adhering to industry standards. Below are best practices tailored for MicroProfile JWT implementations:

1. Enforce Authentication with Validated JWTs: Ensure every request to a microservice includes a valid JWT. Configure MicroProfile JWT to validate tokens using a public key. Reject tokens with invalid signatures, missing claims, or expired exp values.
2. Implement Role-Based Access Control: Restrict endpoint access based on user roles defined in the JWT. Configure role mapping in `microprofile-config.properties` if using non-default claims

Use Short-Lived Tokens: To minimize exposure to compromised tokens, set short expiration times

(exp claim) for JWTs (e.g., 15–30 minutes).

1. Secure Token Transmission: Prevent token interception or tampering by using HTTPS to encrypt data in transit and store tokens in HTTP `Authorization: Bearer` headers (never in URLs or cookies).
2. Manage Cryptographic Keys Securely: Protect keys to sign/verify JWTs by storing public keys in secure locations (e.g., Kubernetes Secrets, AWS KMS). Rotate keys periodically and avoid hardcoding them in source control.
3. Validate and Sanitize JWT Claims: Validate all claims (e.g., iss, aud) in microprofile-config.properties, and Sanitize custom claims before use to prevent injection attacks and misuse of claims.
4. Monitor and Log Security Events: Log JWT validation errors, role mismatches, and token expiration events to detect breaches and audit access patterns. Integrate with monitoring tools (e.g., Prometheus, Grafana) to track anomalies.

These steps will help you secure your microservices against the most common attacks.

Conclusion

MicroProfile JWT offers a standards-based, interoperable approach for securing microservices. It simplifies identity propagation, access control, and stateless security across distributed services. Integrating with Jakarta EE enables secure, scalable, and interoperable authentication without a session state.

Further Reading:

- RFC 7519
- MicroProfile JWT 2.1 Spec
- Jakarta Security 3.0

MicroProfile Rest Client

In microservices architecture, developers often face the cumbersome task of implementing boilerplate code to consume REST APIs - manually constructing HTTP requests, parsing responses, and handling errors. The MicroProfile Rest Client specification addresses this by leveraging Jakarta RESTful Web Services (formerly JAX-RS) annotations to create type-safe Rest client interfaces. Instead of writing low-level HTTP logic, developers define Java interfaces that mirror the target service's endpoints. At runtime, MicroProfile Rest Client dynamically generates an implementation of these interfaces, automating HTTP communication while ensuring compile-time consistency between the client and server contracts.

This chapter introduces the MicroProfile Rest Client, a type-safe framework for simplifying service-to-service communication. We will begin by defining REST client interfaces using Jakarta RESTful Web Services annotations (`@GET`, `@Path`), configuring endpoints via MicroProfile Config, and implementing HTTP invocation. Next, we will explore handling HTTP communication, processing responses, and error handling. By the end of this chapter, you will be able to replace hand-written HTTP boilerplate code with declarative, maintainable clients while adhering to Jakarta EE and MicroProfile standards.

Topics to be covered:

- Introduction to MicroProfile Rest Client
- Setting up Dependencies
- Defining a Rest Client Interface
- Parameter Configuration
- Requests and Response Handling
- Working with JSON Data formats
- Error Handling Strategies

Introduction to MicroProfile Rest Client

The MicroProfile Rest Client specification simplifies RESTful service consumption in Java microservices by replacing error-prone manual HTTP handling with a type-safe, annotation-driven approach. Instead of writing boilerplate code, developers define Java interfaces that mirror the target service's API. Using Jakarta RESTful Web Services annotations like `@GET`, and `@Path`, these interfaces declaratively map methods to HTTP operations (e.g., `/users/{id}` to `getUser(id)`). The framework then generates an implementation at runtime, automating communication while ensuring compile-time consistency between client and server contracts. Tight integration with MicroProfile Config and CDI allows seamless configuration and injection, making it ideal for building resilient, maintainable clients that align with modern microservices practices.

Key Features of MicroProfile Rest Client

The MicroProfile Rest Client simplifies consuming RESTful services in Java microservices with the

following key features:

1. **Type-Safe and Declarative APIs** - The MicroProfile Rest Client allows developers to define REST clients as Java interfaces using Jakarta RESTful Web Services annotations like `@GET`, `@POST`, `@PUT`, `@DELETE`, `@Path`, `@Consumes` and `@Produces`. This approach improves code clarity and ensures compile-time validation, reducing the possibility of runtime errors .
2. **Integration with CDI (Context and Dependency Injection)** - This specification allows developers to seamlessly inject MicroProfile Rest Client interfaces using `@Inject` and `@RestClient` into CDI-managed beans, promoting better dependency management and integration with other components. By leveraging CDI lifecycle management, the MicroProfile Rest Client can benefit from scope management (e.g., `@ApplicationScoped`), proxying, and automatic initialization.
3. **Runtime Configurable with MicroProfile Config** - The behavior of MicroProfile Rest Client can be dynamically configured using MicroProfile Config. This allows properties like the base URL and other client settings to be adjusted without recompilation. The configuration can be provided through *microprofile-config.properties* or environment variables, making the client highly adaptable to different environments.
4. **Support for Asynchronous Execution** - For asynchronous execution, MicroProfile Rest Client can return `CompletionStage<T>`, allowing non-blocking requests. This significantly improves performance & scalability in high-concurrency environments.
5. **Automatic Handling of Redirect Responses** - MicroProfile Rest Client can automatically follow HTTP redirects, simplifying client implementation when working with services that return `3xx` responses.
6. **Secure Socket Layer (SSL) and Security Configuration** - Supports SSL/TLS configuration, including certificates and trust stores, ensuring secure communication between microservices.
7. **Propagation of Headers and Cookies** - Enables automatic propagation of HTTP headers, cookies and context (e.g., authentication tokens), facilitating session management across service calls.
8. **Exception Handling and Custom Providers** - Allows custom exception mapping and response handling, giving developers control over error response based on specific conditions, improving fault tolerance and user experience.
9. **Integration with MicroProfile Fault Tolerance** - This specification Supports resilience patterns like retries (`@Retry`), circuit breakers (`@CircuitBreaker`), and Bulkheads (`@Bulkhead`), ensuring stability in service-to-service communications.
10. **Integration with MicroProfile Long Running Actions (LRA)** - MicroProfile Rest Client can coordinate distributed transactions using LRA annotations (e.g., `@LRA`), enabling compensation logic for long-running processes. This ensures consistency across services in complex workflows.
11. **Portability and Standards Compliance**: This specification enables MicroProfile Rest Client to work across different MicroProfile-compatible runtimes, leveraging Jakarta EE standards (CDI, Jakarta RESTful Web Services, JSON Binding, JSON Processing).

Setting up Dependency for MicroProfile Rest Client

To use MicroProfile Rest Client 3.1 in your project, you need to include the necessary dependencies

in your build configuration. Below are configurations for Maven and Gradle:

Maven Configuration

For Maven-based projects, add the following dependency to your pom.xml file:

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

Gradle Configuration

For Gradle-based projects, add the following dependency to your build.gradle file:

```
dependencies {
    Implementation 'org.eclipse.microprofile.rest.client:microprofile-rest-client-
api:3.1'
    compileOnly 'org.eclipse.microprofile:microprofile:6.1'
}
```

> Tip: The MicroProfile Rest Client is an Eclipse Foundation project. For more details and updates on the project, visit the official repository: MicroProfile Rest Client on GitHub.

Creating MicroProfile Rest Client Interface

To create a MicroProfile Rest Client interface, you need to define a Java interface and annotate it with annotations to map it to a RESTful service.

The `@RegisterRestClient` Annotation

To use the MicroProfile Rest Client, annotate your client interface with `@RegisterRestClient`. This annotation registers the interface as a Rest client within MicroProfile runtime and enables it as a CDI bean, allowing it to be injected into other components.

Example:

```
package io.microprofile.tutorial.inventory.client;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;
```

```
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

import io.microprofile.tutorial.inventory.dto.Product;

@RegisterRestClient(configKey = "product-service")
@Path("/products")
public interface ProductServiceClient {

    @GET
    @Path("/{id}")
    Product getProductById(@PathParam("id") Long id);
}
```

Explanation: In the above code, we define a `ProductServiceClient` within the package `io.microprofile.tutorial.inventory.client`. The interface serves as a Rest client for interaction with a remote product service.

1. `@RegisterRestClient` - declares the `ProductServiceClient` interface as a MicroProfile Rest Client, enabling it to be injected into other CDI-managed components.
2. `configKey = "product-service"` - associates the client with a configuration key, allowing dynamic configuration via MicroProfile Config (e.g., using *microprofile-config.properties* or environment variables).
3. `@Path(/products)` - specifies the base URI path segement for the RESTful service.
4. `@GET` - indicates that the `getProductById()` method handles HTTP GET requests.
5. `@Path("/{id}")` – define a dynamic URI path parameter `{id}`, which will be replaced at runtime with the actual value provided.
6. `@PathParam("id")` - binds the method parameter `id` to the `{id}` placeholder in the request URL.
7. Return Type (`Product`) - specifies that the method returns a `Product` Data Transfer Object (DTO), representing the retrieved product data.

> Note: In CDI environments, it is recommended not to extend AutoCloseable in REST client interfaces. The container manages the lifecycle of injected clients automatically, ensuring proper resource handling without requiring manual closure.

Configuration via MicroProfile Config:

To configure the URI using MicroProfile Config, you need to add a config file named src/main/webapp/META-INF/microprofile-config.properties in your project. This file contains the configuration key and value pairs. In this example, we're configuring the base URI to http://localhost:8080/api/products. We can configure other client properties, such as followRedirects. The followRedirects property specifies whether the client should automatically follow HTTP redirects (3xx status codes) when making RESTful web service calls.

```
product-service/mp-rest/url=http://localhost:8080/api/products
```

```
product-service/mp-rest/followRedirects=true
```

Parameter Configurations

In MicroProfile Rest Client, you can dynamically configure headers, query parameters, and path parameters using Jakarta RESTful Web Services annotations. These annotations bind method parameters to different parts of the HTTP request, enabling flexible and dynamic RESTful client interfaces that can efficiently interact with various endpoints.

Supported Parameter Annotations

1. `@PathParam` – Binds a method parameter to a path variable in the URL.
2. `@QueryParam` – Maps a method parameter to a query string parameter in the request URL.
3. `@HeaderParam` – Attaches a method parameter to an HTTP request header.

Using Path Parameters (`@PathParam`)

Path parameters are used to insert dynamic values directly into the URL path.

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

@RegisterRestClient
@Path("/products")
public interface ProductServiceClient {

    @GET
    @Path("/{id}")
    Product getProductById(@PathParam("id") Long id);
}
```

Example

```
productServiceClient.getProductById(1L);
```

Resulting HTTP Request

```
GET /products/1
```

Why Use @PathParam?

1. Ensures URL structure consistency by enforcing path variables

2. Prevents hardcoding URLs, making the code cleaner and maintainable.

Using Query Parameters (`@QueryParam`)

Query parameters are typically used for filtering, pagination, or optional parameters in the request URL.

Example:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.QueryParam;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

@RegisterRestClient
@Path("/products")
public interface ProductServiceClient {

    @GET
    List<Product> getProductsByCategory(@QueryParam("category") String category);
}
```

Example Call:

```
productServiceClient.getProductsByCategory("electronics");
```

Resulting HTTP Request:

```
GET /products?category=electronics
```

Why Use @QueryParam?

1. Useful for filtering results (?category=electronics).
2. Ideal for pagination (?page=2&size=20).
3. Allows sending optional parameters without modifying the URL structure.

Using Header Parameters (@HeaderParam)

Header parameters are typically used for authentication, authorization, and metadata transmission between client and server.

Example:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.HeaderParam;
import jakarta.ws.rs.Path;
```

```
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

@RegisterRestClient
@Path("/orders")
public interface OrderServiceClient {

    @GET
    List<Order> getOrders(@HeaderParam("Authorization") String authToken);
}
```

Example Call:

```
orderServiceClient.getOrders("Bearer my-secret-token");
```

Resulting HTTP Request:

```
GET /orders
Authorization: Bearer my-secret-token
```

Why Use @HeaderParam?

1. Used for passing authentication tokens (Authorization: Bearer token).
2. Helps with custom metadata exchange (e.g., X-Correlation-ID: 12345).
3. Avoids exposing sensitive data in URLs (e.g., API keys).

Overview of Additional Annotations

1. `@CookieParam` - Binds a method parameter to the value of an HTTP cookie in the incoming request.
2. `@FormParam` — Maps a method parameter to a field in a submitted HTML form (`application/x-www-form-urlencoded` POST body).
3. `@MatrixParam` — Binds a method parameter to a matrix parameter embedded within the URL path segements (e.g., `/product;color=blue;size=large`).
4. `@BeanParam` — Aggregates multiple parameter annotations (path, query, header, etc.) into a single Java bean for cleaner method signature.

> Tip: These annotations eliminate manual string concatenation, making REST client calls type-safe and maintainable.

Handling Requests and Responses

In MicroProfile Rest Client, handling requests and responses involves defining methods in your interface that map to RESTful service endpoints. This ensures that:

1. HTTP requests are automatically constructed based on method definitions.
2. Responses are efficiently deserialized into Java objects (DTOs) or processed manually using `Response`.

Using Jakarta RESTful Web Services annotations, you can define standard HTTP operations such as @GET, @POST, @PUT, and @DELETE. The framework also supports additional methods like @HEAD, @OPTIONS, and @PATCH, providing complete control over HTTP communication when needed. Meanwhile, MicroProfile automatically handles serialization, deserialization, and request execution at runtime.

Handling JSON Data formats

By default, MicroProfile Rest Client supports JSON format without requiring additional configurations. Serialization and deserialization of request and response bodies are automatically handled using JSON-B (Jakarta JSON Binding) or JSON-P (Jakarta JSON Processing).

Developers can directly use Java objects as request bodies or response entities, eliminating the need for manual parsing.

Example:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.Consumes;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

@RegisterRestClient
@Path("/products")
@Produces("application/json")
@Consumes("application/json")
public interface ProductServiceClient {

    @GET
    @Path("/{id}")
    Product getProductById(@PathParam("id") Long id);
}
```

Explanation

1. The `@Produces("application/json")` annotation specifies that the client expects JSON responses. This determines the value of the `Accept` header in HTTP requests.
2. The `@Consumes("application/json")` annotation specifies that the client sends JSON requests. This determines the value of the `Content-Type` header of the request.
3. By default the media type `"application/json"` is used if `@Produces` and `@Consumes` are not explicitly set.
4. MicroProfile Rest Client automatically serializes Java objects to JSON and deserializes responses

into Product DTO (Data Transfer Object) Java object.

Error Handling

Effective error handling is crucial when consuming remote RESTful services. MicroProfile Rest Client provides a structured approach to error handling by mapping HTTP responses to exceptions using the `ResponseExceptionMapper` interface.

This mechanism allows developers to:

1. Convert specific HTTP response codes into custom exceptions.
2. Customize exception handling behavior at runtime.
3. Automatically throw mapped exceptions in client invocations.

Using `ResponseExceptionMapper` interface

The `ResponseExceptionMapper` interface allows mapping an HTTP Response object to a `Throwable` (custom exception). This improves error handling by ensuring meaningful exceptions are thrown instead of manually checking response codes.

How it Works

1. **Scanning and Prioritizing Exception Mappers**: When a client method is invoked, the runtime scans all registered `ResponseExceptionMapper` implementations. Mappers are then sorted in ascending order of priority, determined by the `@Priority` annotation. The mapper with the lowest numeric priority value is checked first.
2. **Handling Responses**: The `handles(int status, MultivaluedMap<String,Object> headers)` method determines whether a mapper should handle a given response. By default, it handles responses with status code 400 or higher, but we can override this behavior.
3. **Converting the Response to an Exception**: The `toThrowable(Response response)` method converts a response into a `Throwable` (exception). Checked exceptions are only thrown if the client method declares that it throws that type of exception of its superclass. Unchecked exceptions (`RuntimeException`) are always thrown.

Example:

```
package io.microprofile.tutorial.inventory.client;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;
import jakarta.ws.rs.core.Response;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import org.eclipse.microprofile.rest.client.annotation.RegisterProvider;

@RegisterRestClient(configKey = "product-service")
@RegisterProvider(ProductServiceResponseExceptionMapper.class)
@Path("/products")
```

```
public interface ProductServiceClient extends AutoCloseable {

    @GET
    @Path("/{id}")
    Response getProductById(@PathParam("id") Long id);
}
```

Explanation:

1. The REST client interface defines an endpoint for retrieving products.
2. The `@RegisterProvider` annotation registers `ProductServiceResponseExceptionMapper`, ensuring custom exception handling.

And below is the corresponding `ResponseExceptionMapper`:

```
package io.microprofile.tutorial.inventory.client;

import jakarta.ws.rs.core.Response;
import org.eclipse.microprofile.rest.client.ext.ResponseExceptionMapper;
import io.microprofile.tutorial.inventory.dto.ProductNotFoundException;

public class ProductServiceResponseExceptionMapper implements
ResponseExceptionMapper<Throwable> {

    @Override
    public Throwable toThrowable(Response response) {
        if (response.getStatus() == 404) {
            return new ProductNotFoundException("Product not found");
        }
        return new Exception("An unexpected error occurred");
    }
}
```

Explanation:

If the response status code is `404`, a `ProductNotFoundException` is thrown. Otherwise, a generic exception is returned.

Using the `RestClientBuilder` Class

While **CDI-based injection** is commonly used for REST clients in MicroProfile, programmatic creation using the `RestClientBuilder` class is beneficial when CDI is unavailable or when dynamic client instantiation is required. This builder provides a **fluent API** for configuring and constructing REST client proxies without relying on constructors that require numerous arguments.

Using `RestClientBuilder` simplifies object creation, improves code readability, and supports **method chaining**, where each configuration method returns the builder instance itself.

Example: Inventory Service Calls Product Service

In the MicroProfile Ecommerce Store, the `InventoryService` must verify whether a product exists before checking or updating inventory. This interaction can be handled by calling the `ProductService` using a REST client interface.

```
package io.microprofile.tutorial.store.inventory.service;

import io.microprofile.tutorial.store.inventory.client.ProductServiceClient;
import io.microprofile.tutorial.store.product.entity.Product;
import org.eclipse.microprofile.rest.client.RestClientBuilder;

import java.net.URI;
import java.util.concurrent.TimeUnit;

public class InventoryService {

    public boolean isProductAvailable(Long productId) {
        URI productApiUri = URI.create("http://localhost:8080/api");

        try (ProductServiceClient productClient = RestClientBuilder.newBuilder()
                .baseUri(productApiUri)
                .connectTimeout(3, TimeUnit.SECONDS)
                .readTimeout(5, TimeUnit.SECONDS)
                .build(ProductServiceClient.class)) {

            Product product = productClient.getProductById(productId);
            return product != null;

        } catch (Exception e) {
            // Log exception (omitted for brevity)
            return false;
        }
    }
}
```

Explanation

- The `isProductAvailable()` method accepts a product ID and returns `true` if the product exists in the catalog.
- A `URI` object is created pointing to the base path of the ProductService API using `URI.create()`.
- A `ProductServiceClient` instance is created using the builder pattern inside a `try-with-resource` block:
  - `newBuilder()` initializes the client builder.
  - `baseUri()` sets the root endpoint of the target service.
  - `connectTimeout()` and `readTimeout()` define connection and read timeouts respectively.
  - `build()` finalizes and returns the configured client proxy.

- Because `ProductServiceClient` extends `AutoCloseable`, the try-with-resources block ensures that the client is automatically closed after the operation, preventing resource leaks.
- If a `Product` object is successfully returned, `true` is returned.
- Any exceptions are caught and handled appropriately, returning `false` in case of failure.

This approach is especially useful for **utility services**, **batch jobs**, or environments where REST client configuration must be **dynamic or conditional**, and manual client lifecycle management is necessary.

> Tip: When building MicroProfile REST clients programmatically (using `RestClientBuilder`), ensure that your client interface extends `AutoCloseable` and uses try-with-resources to release resources automatically.

Conclusion

The MicroProfile Rest Client provides a declarative, type-safe, and efficient mechanism for interacting with RESTful services in Java microservices. It reduces boilerplate code and lets developers focus on core business logic while still offering fine-grained control through features like `RestClientBuilder`.

By integrating seamlessly with other MicroProfile specifications—such as **Config**, **Fault Tolerance**, and **JWT Authentication**—the Rest Client helps enhance the **security**, **resilience**, and **maintainability** of cloud-native applications.

Key Takeaways

- Removes boilerplate HTTP code, improving clarity and maintainability.
- Automatically handles JSON serialization and deserialization.
- Supports **CDI injection** for managed client lifecycles.
- Integrates with **Fault Tolerance** for retries, timeouts, and circuit breakers.
- Enhances **security** through header propagation and authentication mechanisms.

With MicroProfile Rest Client, building robust and maintainable microservices that communicate over REST becomes **simpler**, **more flexible**, and **more powerful**. This concludes the MicroProfile tutorial. You are now equipped with the foundational knowledge to build robust, cloud-native microservices using the MicroProfile specification. Thank you for following along, and happy coding!