O'REILLY®

Compliments of IBM

Developing Open Cloud Native Microservices

Your Java Code in Action

Graham Charters, Sebastian Daschner, Pratik Patel & Steve Poole

REPORT

Build

Smart

Java is the open language for modern, microservice applications. Explore Java for your next cloud app today.

ibm.biz/OReilly-Java

Developing Open Cloud Native Microservices

Your Java Code in Action

Graham Charters, Sebastian Daschner, Pratik Patel, and Steve Poole

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Developing Open Cloud Native Microservices
by Graham Charters, Sebastian Daschner, Pratik Patel, and Steve Poole

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Chris Guzikowski
Development Editor: Michele Cronin
Production Editor: Christopher Faucher
Copyeditor: Rachel Monaghan
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2019: First Edition

Revision History for the First Edition
2019-08-07: First Release

978-1-492-05272-2

[LSI]

Table of Contents

Foreword

Businesses do not care about microservices, twelve-factor apps, or cloud native Java. They care about delivering business value to their customers, open and community-driven APIs, and runtime portability.

With a great developer experience comes increased productivity with "to the point" code—meaning high maintainability. The less code you have to write, the easier it is to spot bugs and implement new functionality. J2EE was designed as a "micro cloud" to host multiple, isolated applications. The EJB programming restrictions discouraged you from accessing the local filesystem, loading classes dynamically, or changing your isolated environment in any way. These formally unpopular, but now obligatory, programming restrictions are older than J2EE (EJBs came first) and are still a good idea today.

Java EE 5 greatly simplified the programming model, servers became leaner, and the transition from shared application servers to single microservice runtimes began. Between Java EE 5 and Java EE 8 the platform was extended with internet-related APIs for REST support, JSON, WebSockets, and asynchronous behavior out of the box. The build is as simple as the programming model is. All the Java EE APIs are bundled as a single, "provided" dependency in Maven's *pom.xml*. Java EE/Jakarta EE is mature, stable, and therefore a genuinely boring runtime. Most Java EE APIs are iteratively refined for years, so expect no revolution. This is great news for businesses and pragmatic developers: demand for migrations is decreasing. The frequent Eclipse MicroProfile release cadence is quickly filling possible functionality gaps by delivering new APIs or extending existing ones on demand.

All popular runtimes are implementing Java EE/Jakarta EE and Eclipse MicroProfile APIs at the same time. You only have to download the Jakarta EE/MicroProfile runtime of your choice (a ZIP file), extract the archive, and define the Java EE/MicroProfile API as a "provided" dependency in Maven/Gradle (10 lines of code). In 10 minutes or less, the "Java Cloud Native Microservice Dream Team" is ready to deliver value to the customer with the very first iteration.

This book gives you a pragmatic introduction to cloud native Java, from the Java development kits and the open source ecosystem to a minimalistic coffee shop example. With MicroProfile and Jakarta EE, minimalism is the new best practice.

— *Adam Bien*
http://adam-bien.com

The Jakarta EE and Eclipse MicroProfile communities are significant new efforts that will be shaping the future of both Java and cloud computing for years to come. In particular, both of these communities are leading the industry in providing vendor-neutral specifications, coupled with multiple, liberally licensed, open source implementations for cloud native Java. In doing so, these two Eclipse Foundation-hosted communities are creating the safe choice in Java platforms for the new cloud-based systems being built today and tomorrow.

It is important to understand that Java remains a critically important technology in the cloud. Virtually all of the Fortune 1000 run significant portions of their business on Java. Just as importantly those same enterprises collectively have millions of developers with both knowledge of Java and of their businesses. The missions for Jakarta EE and MicroProfile are to provide paths forward for these enterprises and their developers as they rearchitect their applications to become cloud native.

Jakarta EE and MicroProfile represent two quite different paths to technological success. Jakarta EE is the successor to Java EE, the more than 20-year-old technology that laid the groundwork for Java's enormous success in the enterprise. Java EE's success was largely the result of careful evolutions of APIs and specifications, release-to-release compatibility that has lasted for years, and multi-vendor support and participation. MicroProfile only started in 2016 and is a demonstration of a truly open community's ability to innovate quickly. With a cadence of three releases per year, MicroProfile has rapidly evolved to a complete set of specifications for building, deploying, and managing microservices in Java.

Eclipse community projects are always driven by great developers, and at the Eclipse Foundation the cloud native Java community has had many important contributors. I would like to recognize (in no particular order) the contributions of just a few: Bill Shannon, Dmitry Kornilov, Ed Bratt, Ivar Grimstad, David Blevins, Richard Monson-Haefel, Steve Millidge, Arjan Tijms, John Clingan, Scott Stark, Mark Little, Kevin Sutter, Ian Robinson, Emily Jiang, Markus Karg, James Roper, Mark Struberg, Wayne Beaton, and Tanja Obradović are but a few individuals who have been leaders among this community. My apologies in advance for forgetting someone from this list!

I have known, or known of, the authors for many years, and during that time they have been tireless champions of Java technologies. This book will hopefully raise the profile of Java's important role in cloud native technologies, and lead to broader knowledge and adoption of the APIs, frameworks, technologies, and techniques which will keep Java relevant for this new generation of cloud-based systems and applications.

— Mike Milinkovich
Executive Director, Eclipse Foundation

Preface

Cloud native is a concept that's been around for a number of years. Initially, companies developed and promoted their own technologies and perspectives on software development; however, in recent years, the industry and technologies have matured, and this has led to greater collaboration around open source and open standards. We believe a user's interests are best served by adopting technologies that are based on open standards and open source, all produced by communities founded on open governance. It's taken us a few years to get here, but it's now possible and easy to build high-quality cloud native Java applications using technologies that fit these criteria. In this book, we'll show you how.

Prerequisites for Reading This Book

This book is primarily aimed at readers who have some knowledge of the Java programming language and who wish to get started with creating cloud native Java applications. Readers without an understanding of Java can still benefit from the book, as many of the principles will hold regardless of programming language or framework.

Why This Book Exists

This book exists to help Java developers begin their journey into cloud native. There is much to learn on this voyage, and this book will provide an introduction to important high-level concepts and guide the reader along a well-trodden and proven technical path.

What You Will Learn

By the end of this book you will understand the unique challenges that arise when creating, running, and supporting cloud native microservice applications. This book will help you decide what else you need to learn when embarking on the journey to the cloud, and how modern techniques can help with deployment of new applications in general.

The book will briefly explain important considerations for designing an application for the cloud. It covers the key principles for microservices of distribution, data consistency, continuous delivery, and the like, which not only are important for a cloud application but also support the operational and deployment needs of modern 24x7, highly available Java-based applications in general.

How This Book Is Organized

Technology moves at a fast pace. Keeping up with new innovations while still maximizing your choices can be a challenge. In this book we will explain how to use open technologies to develop Java applications that follow the principles of cloud native, helping you keep abreast of new thinking and ensuring your freedom of action.

We'll start in Chapter 1 by defining *cloud native* and explain why Java is the programming language of choice.

In Chapter 2 we'll expand on what we mean by *open technologies* and explain why taking an open approach is important. We'll outline how to identify good open technologies and then introduce the ones we'll use in the remainder of the book.

In Chapters 3, 4, and 5, we'll use a complete example to demonstrate how to develop a set of cloud native Java microservices using our selected open technologies. We'll show how to develop a microservice backed by a database and how to expose it as a JSON/HTTP REST service. We'll then explain how to secure the service, provide a human- and machine-readable API definition for consumers to use, gracefully handle service faults, and more. Finally, we'll describe how to make the service *observable* so that you can monitor the container, runtime, and application and be alerted to—and react to—any problems that arise.

Finally, in Chapter 6 we'll wrap things up and talk about future directions for open cloud native Java applications.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

TIP This element signifies a tip or suggestion.

NOTE This element signifies a general note.

! This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/IBM/ocn-java*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Developing Open Cloud Native Microservices* by Graham Charters, Sebastian Daschner, Pratik Patel, and Steve Poole (O'Reilly). Copyright 2019 Graham Charters, Sebastian Daschner, Pratik Patel, Steve Poole, 978-1-492-05272-2."

If you feel your use of code examples falls outside fair use or the permission given above, contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For almost 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472
> 800-998-9938 (in the United States or Canada)
> 707-829-0515 (international or local)
> 707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

First of all, we would like to thank the open source contributers who put their time into the technologies covered in this book. Without their experience and contribution to open source Java and Cloud technologies, it would not be possible to have this rich ecosystem on the cutting edge of software development.

We would also like to thank our reviewers, Chris Devers, Neil Patterson, Anita Chung, and Kevin Sutter, who helped shape this book and make it of the highest quality.

We would also like to thank Adam Bien and Mike Milinkovich for contributing the foreword, and for their leadership in community activities around open source Java and the open Java work at the Eclipse Foundation.

Finally, we would like to thank the team at O'Reilly for bearing with us as we worked to create this book. We hope you enjoy reading it!

CHAPTER 1

Introduction

What It Means to Be Cloud Native

Cloud native applications can be described in a single line: applications that utilize and are optimized, or *native*, for cloud computing environments. To fully understand what this means, you must understand cloud computing and how it differs from traditional monolith software development. Software professionals, to ensure their companies remain competitive, must employ a modern style of development and deployment that uses the compute and management infrastructure available in cloud environments. In this section, we will discuss cloud native in depth to prepare you for the rest of this book.

Microservice Oriented

First, cloud native architectures break from the traditional design of monoliths and rely on containers (e.g., Docker) and serverless compute platforms. This means that applications are smaller and composed at a higher level. We no longer extend an existing application's functionality by creating or importing a library into the application, which makes the application binary larger, slower to start and execute, and more memory-intensive. Instead, with cloud native we build new microservices to create a new feature and integrate it with the rest of the application using endpoint type interfacing (such as HTTP) and event type interfacing (such as a messaging platform).

For example, say we needed to add image upload capability to our application. In the past, we would have imported a library to implement this functionality, or we would have written an endpoint where we accept a binary type through a web form and then saved the image locally to our server's disk. In a cloud native architecture, however, we would create a new microservice to encapsulate our image services (upload, retrieve, etc.). We would then save and retrieve this image, not to disk, but to an object storage service in the cloud (either one we would create or an off-the-shelf service provided by our cloud platform).

This microservice also exposes an HTTP endpoint, but it is isolated from the rest of the application. This isolation allows it to be developed and tested without having to involve the rest of the application —giving us the ability to develop and deploy faster. As it is not tightly coupled with the rest of the application, we can also easily add another way to invoke the routine(s): hooking it into an event-driven messaging system, such as Kafka.

Loosely Coupled

This brings us to our second main discussion point on cloud native: we rely more on services that are loosely coupled, rather than tightly coupled monolith silos. For example, we use an authentication microservice to do the initial authentication. We then use JSON Web Tokens (JWT) to provide the necessary credentials to the rest of our microservices suite to meet the security requirements of our application.

The loose coupling of these small, independent microservices provides immense benefits to us software developers and the businesses that run on these platforms:

Cost
: We are able to adapt our compute needs to demand (known as *elastic computing*).

Maintainability
: We are able to update or bug-fix one small part of our application without affecting the entire app.

Flexibility
: We can introduce new features as new microservices and do staged rollouts.

Speed of development
: As we are not doing low-level management of servers (and dynamic provisioning), we can focus on delivering features.

Security
: As we are more nimble, we can patch parts of our application that need urgent fixes without extensive downtime.

Twelve-Factor Methodology

Along with these high-level cloud native traits, we should also discuss the *twelve-factor application methodology*, a set of guidelines for building applications in cloud native environments. You can read about them in detail on their website, but we'll summarize them for you here:

1. A versioned codebase (like a Git repository) matches a deployed service, and the codebase can be used for multiple deployments.
2. All dependencies should be explicitly declared and should not rely on the presence of system-level tools or libraries. Explicitly declaring and isolating dependencies ensures portability from developer machine to continuous integration/continuous delivery (CI/CD) to production server.
3. Configuration should be stored in the environment for things that vary between deployments (e.g., environment variables).
4. Backing services are treated as attached resources, and there is no distinction made between on-premise and third-party resources; all are addressed via locator/credentials or URL, provided via environment configuration.
5. Strict separation between the stages of build, release, and run ensures reproducibility.
6. Deploy applications as one or more stateless processes. Shared state should be portable and loadable from a backing service.
7. Share and export services via a declared port.
8. Scaling is achieved using horizontal scaling.
9. Fast startup and graceful shutdown maximize robustness and scaling.

10. Different environments (dev/test/prod) should be as similar as possible. They must be reproducible, so do not rely on external inputs in their construction.
11. Logs are to be emitted as event streams (stdout/stderror) for easy aggregation and collection by the cloud platform.
12. Admin tasks must be in source control, packaged with the application, and able to run in all environments.

Following these best practices will help developers succeed and will reduce manual tasks and "hacks" that can impede the speed of development. It will also help ensure the long-term maintainability of your application.

Rapid Evolution

Cloud native development brings new challenges; for example, developers often see the loss of direct access to the "server" on which their application is running as overly burdensome. However, the tools available for building and managing microservices, as well as cloud provider tools, help developers to detect and troubleshoot warnings and errors. In addition, technologies such as Kubernetes enable developers to manage the additional complexity of more instances of their microservices and containers. The combination of microservices required to build a full, large application, often referred to as a *service mesh*, can be managed with a tool such as Istio.

Cloud native is rapidly evolving as the developer community better understands how to build applications on cloud computing platforms. Many companies have invested heavily in cloud native and are reaping the benefits outlined in this section: faster time to market, lower overall cost of ownership, and the ability to scale with customer demand.

It's clear that cloud native is becoming *the* way to create modern business applications. As the pace of change is fast, it is important to understand how to get the best out of the technology choices available.

Why Java and the Java Virtual Machine for Cloud Native Applications?

In principle any programming language can be used to create microservices. In reality, though, there are several factors that should influence your choice of a programming language.

Innovation and Insight

The first consideration is simply the pace of innovation and where it is taking place. The Java community is probably the place where those with the deepest knowledge and experience in using the internet for business gather. The community that created Enterprise Java and has made it the de facto business platform is also the community that is leading the evolution of cloud native thinking. They are exploring all the aspects of what it means to be cloud native—whether it is serverless, reactive, or even event driven. Cloud native continues to evolve, so it's important to keep abreast of its direction and utilize the best capabilities as they're developed in the Java community.

Performance and Economics

Next, consider that a cloud environment has a significantly different profile from a traditional one that runs on a local server. In a cloud environment, the amount of compute resource is usually lower and, of course, you pay for what you use. That means that cloud native applications need to be frugal and yet still performant. In general, developers need runtimes that are fast to start, consume less memory, and still perform at a high level. Couple this need with cloud's rapid evolution, and you are looking for a runtime with a pedigree of performance and innovation. The Java platform and Java Virtual Machine (JVM) are the perfect mix of performance and innovation. Two decades worth of improvements in performance and steady evolution have made Java an excellent general purpose programming language. Cloud native Java runtimes like Eclipse OpenJ9 offer substantial benefits in runtime costs while maintaining maximum throughput.

Software Design and Cloud Solutions

Finally, it's important to understand that a modern cloud native application is more complex than traditional applications. This complexity arises because a cloud native solution operates in a world where scale, demand, and availability are increasingly significant factors. Cloud native applications have to be highly available, scale enormously, and handle wide-ranging and dynamic demand. When creating a solution, you must look carefully at what the programming language offers in terms of reducing design issues and bugs. The Java runtime, with its object-oriented approach and built-in memory management, helps remove problems that are challenging to analyze locally, let alone in a highly dynamic cloud environment.

Java and the JVM address these challenges by enabling developers to create applications that are easier to debug, easier to share, and less prone to failure in challenging environments like the cloud.

Summary

In this chapter we outlined the key principles of being cloud native, including being microservice oriented, loosely coupled, and responsive to the fast pace of change. We summarized how following the twelve-factor methodology helps you succeed in being cloud native and why Java is the right choice for building cloud native applications.

In the next chapter we explore the importance of an *open approach* when choosing what to use for your cloud native applications. An open approach consists of open standards to help you interoperate and insulate your code from vendor lock-in, open source to help you reduce costs and innovate faster, and open governance to help grow communities and ensure technology choices remain independent of undue influence from any one company. We'll also outline our technology choices for the cloud native microservices shown in the remainder of the book.

CHAPTER 2

Open Technology Choices

In this chapter we're going to look at open technology choices for cloud native Java applications. For us, "open" encompasses three principles, shown in Figure 2-1.

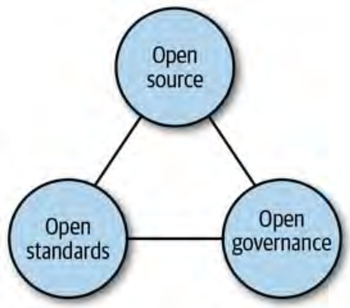


Figure 2-1. Three principles of open technologies

We'll start by talking about the role of open source, why it's important to us, and how to evaluate candidate projects. Next, we'll talk about the role of open standards, the benefits they provide, and their relationship to open source. We'll then talk about open governance, its importance across both open source and open standards, and how it helps in building open communities.

We'll show you how to use your new understanding of open source, standards, and governance to make informed open technology choices. Finally, we'll describe the open technologies we've chosen to use in the subsequent chapters of this book.

Open Source

Most developers are aware of the concept of open source. It's worthwhile, however, to distinguish between *free* software and *open source* software. The benefits of free (as in no cost) software are evident to all of us. It's great to be able to use something without paying for it. It's important, though, to understand if there are hidden costs to using "free" software. Quite often there are usage restrictions, such as time limits, that impact your ability to use the software as part of a solution.

In essence, open source means the source code is available for anyone to see. But that's not the only reason we care about open source —in fact, as users of open source, we very rarely look at the source code. So why does open source matter to us? There are a number of reasons:

Cost
: I can use it without paying, and there is both community and paid support available.

Speed
: If I use this open source project, it will help me get my job done quicker.

Influence
: If I find a problem, I can fix it or raise an issue. If I need an enhancement, I can contribute it or request a new feature.

Community
: If I have a problem, the community will hopefully help; I don't need to open a support ticket. I can also become part of the community.

Opportunity
: Significant, diverse open source projects grow larger markets where there will be demand for my skills.

As you can see, many of the characteristics of open source—the reasons we'd want to use it—don't stem from its simple availability. For example, just because the source is available doesn't mean there's a community to support it. The reality is, there's a wide-ranging set of attributes of open source projects that you need to consider when choosing what is right for you, including the following:

License
: Does the open source license permit me to use the software in the way I'd like to? Many open source projects come with permissive licenses on usage that allow modification, sharing, or creation of commercial solutions. Some, of course, do not allow this flexibility.

Purpose
: Why does the project exist? This goes beyond its technical scope and covers factors such as why the contributors are participating.

Community
: Does the open source project have a vibrant contributor and user community?

Standards
: Does the project implement open standard APIs, allowing me to switch to other providers?

Governance
: Is the project maintained under an open governance model, such as those under the Apache Software Foundation, Eclipse Foundation, or Linux Foundation? We'll talk more about open governance later in this chapter.

The open source approach is probably the best way to develop and evolve complex software. The right combination of license, community, governance, and standardization has produced some of the world's most successful technologies. Getting this combination right can be a challenge, however, so in the next sections we'll go into more detail on the considerations for community, standards, and governance.

Open Community

Open source is not just about the code: it's also about how the code is designed, created, tested, and supported. It's about the people involved and how they interact. There are many approaches to open source, from a single individual sharing their work on GitHub all the way to large team efforts, spread out across companies and geographies.

So what should we look for in an open source community?

Vibrancy

Ideally, vibrant open source projects with multivendor participation (i.e., multiple companies and/or individuals) and open governance should be preferred, as they have been shown to offer the maximum benefit to participants and consumers alike. However, many open source projects are single-individual or single-company efforts.

The vibrancy of a project is measured in terms of factors such as the number of active contributors, number of recent contributions, contributor company affiliations, and support for the user base. If a project isn't very active, has limited contributors, ignores outside contribution, or neglects its user base, these are all reasons to think twice before adopting. Here are some questions to ask yourself when vetting a project:

- Is the project overly reliant on an individual or single company?
- Is the project managed under an open governance model?
- Is the project license permissive so that I can fork and patch if necessary?
- Does the project implement open standard APIs, and are there viable alternative implementations?
- Does the project exist only to promote a commercial product?
- Does the community actively support its user base?
- Does the community consider the impact of changes on its existing users?
- Does the project actively encourage new contributions—ideas, fixes, features?

Vendor neutrality

Software development is a creative process. Developers spend valuable time designing, writing, testing, fixing, and even supporting the software. Some developers do contribute to open source for the love of it. They enjoy giving something to the community, or it scratches a metaphorical itch. However, for the vast majority of open source projects, open source is a company's business model. It might be that it facilitates collaboration with other companies and so enables markets to grow, similar to collaboration on open standards. Often such projects live in open source foundations, such as the Eclipse Foundation, Linux Foundation, or the Apache Software Foundation.

Many open source projects are solely or predominantly created by a single company. Here, the open source is freely available to use with the intention of selling support or building a commercial offering that contains additional functionality.

These are common patterns, and there's nothing wrong with them. However, they encourage centralized single vendor control over a project and, in the case of the commercial offering, discourage contribution of important features to the open source project.

When you're choosing an open source technology, it's important to understand the community involvement and associated business strategies. Any risk associated with adoption is greatly reduced if there is a clear history of broad, meritocratic participation across the community or if the technology is an implementation of open standards.

In the world of cloud, it's easy to overlook the importance of open technologies. Cloud solutions promise to help us deliver and gain value faster than traditional approaches. Coupled with the fact that application lifespans are decreasing, it's not hard to fall for the seductive argument of using just one vendor's technologies. After all, what can possibly go wrong with limiting oneself to a single vendor?

Thankfully, the ubiquity of container technologies (e.g., Docker and Kubernetes clustering) gives us a common cloud foundation on which we can rely. This allows us to decouple the cloud native applications and implementation technology choice from the cloud provider choice.

A focus on collaboration

Like with open standards, open source has repeatedly demonstrated that collaboration and sharing is in fact a business enabler and amplifier. Examining the existing software communities, we can easily see that the more successful ones are those that understand that collaborating and innovating together creates a fast-paced ecosystem whose potential audience is larger than the individual markets of each of the participating companies.

Everyone benefits more from the creation of a large market where no one participant owns the standards or the implementations. The collaborative model also fuels innovation because everyone in the community has a vested interest to keep it evolving and, since the community is larger, the rate of innovation and quality is often higher.

Open Standards

There's no question that open standards are incredibly important. Imagine a world without TCP/IP, HTTP, HTML, and the like. Standards enable contracts for interoperability and portability. They protect users against vendor and implementation lock-in, but also enable ecosystems to grow and thrive by enabling collaboration and more rapid and greater adoption. They enable vendors to collaborate or interoperate and then differentiate themselves through qualities of service, such as performance, footprint, or price. They also enable developers to build skills applicable to more employers and broader markets.

In the early 2000s, the collaboration between many companies and individuals around Java EE created a vibrant ecosystem that made Java the dominant language for enterprise applications. The APIs it defined enabled multiple implementations (IBM WebSphere, Oracle JBoss, Oracle WebLogic, etc.), and to this day, those APIs continue to thrive through implementations from IBM, Oracle, Payara, Tomitribe, and others.

Historically, the Enterprise Java standards have been created through the Java Community Process (JCP). In recent years, Eclipse has become the place for Enterprise Java standards, initially with the development of the Eclipse MicroProfile APIs and more recently with Jakarta EE (the open future of the Java EE specifications).

An open standards approach, in conjunction with structured and impartial governance from an organization like the Eclipse Foundation, means that innovation will be maximized and all the participants will have equal footing.

Open Governance

Having a clear, fair, and impartial community process is essential for the long-term health and prosperity of any open community, whether it's open source or open standards. Communities with a good governance model may still fail for other reasons, but without one they are rarely successful. Good intentions to play fair at the start of a collaboration can easily be lost. The draw of fame or commercial gain can turn a collaboration based on simple personal agreements into a combat zone where friends of many years can become bitter enemies.

Collaborators in open source and open standards projects need to understand up front that there are fair rules about how agreement is reached and how disputes are settled. There should be clear guidelines about how intellectual property is handled, how ownership and responsibilities are assigned, how contributions are licensed and accepted, and so on. Ideally, there will be other specifications, such as codes of conduct and some form of community charter.

This may sound like a lot of work and overhead for a new collaboration to incur, but that need not be the case. There are many good examples of communities that have procedural frameworks in place to help with these challenges. In fact, some of these communities are designed with the aim of encouraging and nurturing the growth of open source projects or open standards that meet the criteria we've explained in this chapter. Examples of such communities are the Apache Software Foundation, the Eclipse Foundation, the Linux Foundation, OASIS, and W3C.

Having an open governance model has another major benefit: it will attract more participation, because those joining understand and agree with the rules up front. Knowing that they will be treated equally and fairly is a significant factor for new players because they are assured that their contributions will not be misused and that their voice and opinion will be equal to others.

As you've now seen, there are many facets of open technology to think about. In the first half of this chapter, we've explained what we consider to be important ones both for the long-term viability of an open source project and for you as a consumer of that project. In the second half of this chapter, we'll turn to the technology considerations themselves and our views on the combination of technology and community that we've chosen for open cloud native Java.

Choosing Application Technologies

When considering any project, you need to factor in how critical the technology is to your solution, how complex it is, and how easy it would be to switch out. If you're choosing a single-vendor open source project, picking one that exposes standard APIs greatly reduces the associated risk.

Small, modular components in a framework—that is, ones that can easily be replaced with alternative approaches—offer less risk compared to building on top of a framework or runtime that is not modular and built on standard APIs.

Selecting the right cloud native technologies comes down to a mix of need, risk, complexity, and community. The right combination is not necessarily the same for everyone. If you are seeking a Java-based approach that will have the best chance of navigating the complexities of cloud, the Eclipse Foundation provides an excellent platform. The Eclipse Foundation offers a vendor-neutral, community-led, and (above all) community-focused attitude; the Eclipse MicroProfile and Jakarta EE technologies are the best starting point in our opinion.

The Eclipse Foundation is the home of many great technologies and has a justified reputation as a place where communities can grow and work together to build leading-edge, best-of-breed solutions. Originally set up as an openly governed organization for the development of open source tools and runtime technologies, the Eclipse Foundation has more recently moved to support the development of open standards. This means that, as organizations go, Eclipse is pretty unique in being able to tick all three boxes: open standards, open source, and open governance.

The Foundation is an exemplar of how communities can work together to achieve something bigger than any one of the participants could do on their own. Its reputation as a safe, fair, and *active* home for open source and open standards is one of the reasons Oracle chose to contribute its Java EE codebase and specifications to the Foundation. Let's now take a look at Eclipse technologies in a little more detail.

Java EE and Jakarta EE

Java EE has formed the basis of Enterprise Java since 1999. From its early days delivering basic web container capabilities to the final release of Java EE 8, it has provided the foundational technologies for web and microservice-based applications.

Java EE originally consisted of 35 specifications and, to provide portability assurances, compliant platform vendors were required to provide implementations of them all. But, with the advent of composable runtimes, such as Open Liberty and Wildfly Swarm, the days of monolithic Java EE platforms came to an end. The time came to think of Java EE as a catalog of capabilities to pick and choose from. You'd like to write a REST service? Just include JAX-RS and JSON-B. Need database persistence? Just include JPA.

For the first 18 years of Java EE's life, it was governed by a process managed by Sun, and more recently, Oracle. This governance led to strained relationships at times between the participants and did not lend itself to the most inclusive and collaborative environment. In 2017, Oracle announced it would relinquish control of Java EE, and it subsequently chose Eclipse as the new custodians. The new community chose the name Jakarta EE to represent the new platform.

Since the announcement, a great deal of work has been done to contribute the reference implementations (RIs), compliance tests (TCKs), and specifications and to set up the legal infrastructure necessary for the future development of Jakarta EE. This has resulted in the first release of Jakarta EE, Jakarta EE 8, that provides parity with Java EE 8. From here on, an openly governed, open community will create new specifications to contribute to Jakarta EE 9 and beyond.

Java EE 8, and therefore Jakarta EE 8, still consists of 35 specifications, a number of which (e.g., JAX-RS and JSON-B/P for REST services[1]) are essential for cloud native applications, and many of which (e.g., JPA, Bean Validation, Contexts and Dependency Injection for database access, and dependency injection and validation of simple components) will be important. Coupling this with the fact that there are multiple open source implementation options available, we believe Jakarta EE technologies are a great foundation for cloud native microservices.

1 Java API for RESTful Web Services and JSON Binding/Processing, respectively.

Eclipse MicroProfile

Eclipse MicroProfile is an open collaboration around technologies for Enterprise Java microservices. Initially formed as its own organization, MicroProfile soon moved to the Eclipse Foundation. The MicroProfile community is represented by many large companies and Java user groups, including IBM, Red Hat, Oracle, Microsoft, the LJC (London Java Community), and SOUJava, and the projects themselves have 140 individual contributors. The MicroProfile collaboration delivers specifications and compliance tests, and it benefits from multiple open source implementations, including Open Liberty, SmallRye, Payara, TomEE, and Quarkus.

MicroProfile is designed to complement Java EE, reusing technologies where appropriate. In fact, the first version was based purely on the Java EE 7 technologies of JAX-RS, CDI, and JSON-P.

Subsequent MicroProfile releases have moved up to Java EE 8, adding JSON-B. They've also added many new specifications developed by the MicroProfile community. These broadly fall into three categories, as shown in Figure 2-2 and described in the following list:

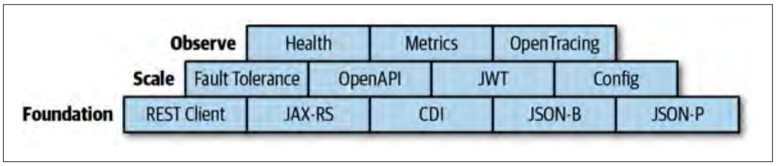


Figure 2-2. Specifications developed by the MicroProfile community

Foundation
: These are the foundational technologies for writing and calling microservices. Most are from Java EE, but MicroProfile also adds a really simple type-safe REST client.

Scale
: These are not about scaling a single microservice; Kubernetes does that perfectly well. Instead, they cover the APIs a developer needs in order to start building large numbers of cloud native microservices owned by independent teams—for example, the ability to publish a service API definition for another team to use, or gracefully handling problems with the services you depend on, such as intermittent failures or slow responses.

Observe
: These are the technologies that help with monitoring services for health (e.g., whether or not the service is ready to receive requests or is dead and needs restarting), retrieving service metrics to enable alerting of abnormal behavior, and tracing service requests.

To keep pace with the industry demands for cloud native technologies, MicroProfile now releases new specifications every four months, and a number of vendor implementations typically soon follow.

Given that MicroProfile is based on tried-and-tested Enterprise Java technologies, is designed by a vibrant and open collaborative community, and benefits from a number of independent open source implementations, it's an ideal choice for cloud native microservices.

JVM Choices for Cloud Native Applications

It's very common to not give a great deal of thought to the choice of Java Virtual Machine. Even as a key part of any Java solution, the JVM is generally taken for granted.

For most of its life the development of the JVM has been strictly controlled by the JCP and various specification enhancements called Java Specification Requests (JSR).

This community process has ensured that the JVM provides the same runtime experience for applications regardless of the underlying operating system or hardware. The JVM is probably the most successful cross-platform environment ever. Although there are always edge cases, it is safe to say that the vast majority of Java applications will run as is on any JVM and will function exactly the same.

Using the JCP process as the vehicle for change has become too heavyweight for modern needs. More and more of the JVM's direction is now being developed via the OpenJDK project. This project maintains the reference codebase for the JVM, and many minor (and sometimes not so minor) decisions about the JVM's behavior are made there. The OpenJDK community is funded and staffed by Oracle with both monetary and infrastructure contributions by Red Hat and others. This arrangement means that Oracle's commercial interests will spill into the direction the OpenJDK project takes, but as long as those interests match the wider community, that is not a concern. Developing a JVM requires deep skills, knowledge, and experience, so it is not surprising that there are few contributors to the effort who are not funded by a commercial entity.

Because the OpenJDK project owns and maintains the Java codebase (including the JVM), it is almost the only game in town for Java runtimes. The leaders of the project work hard to ensure that the many millions of Java developers and end users can continue to rely on there being a fit-for-purpose Java runtime for their current and future needs.

The JVM from the OpenJDK project is called Hotspot. There are others implementations available. We'll examine one in a moment, but first it's important to understand that because the JVM's behavior is so clearly specified and there is a large compliance suite (which checks that the JVM in question conforms to the specification), replacing one JVM with another is not a particularly difficult or risky thing to do.

Why Would You Want to Use a Different JVM?

We've said before that when selecting open source components, you must think carefully about your choice. What happens if one of the selected components goes away, the license changes, or something else happens that's detrimental to your project? Another, more programmatic consideration is whether the direction of the related community is aligned with industry thought or wider pressures.

In this instance it's worth looking at the history of JVM development from an economic point of view. The JVM specification has evolved over 20-odd years to provide a best-of-breed platform for business applications running on large servers. The applications created have evolved to span multiple on-premise servers and are designed to be available 24/7. In this model the economics favor applications (and runtimes) that provide maximum throughput over many months or even years. This model drives design thinking to maximize performance at the cost of initial startup and memory consumption. If the application is going to run long term, then increasing startup time at the cost of improved runtime efficiency is an acceptable tradeoff.

For most of Java's lifetime, this tradeoff has been the underlying driver for the direction of Java features. With the rise of cloud-based solutions, the economics have changed—or rather, the JVM now has an additional, conflicting set of performance and memory demands upon it. Short-lived services need to start quickly, and given that clouds often charge based on memory consumption, you want them to have a small memory footprint.

Modifying the Hotspot JVM codebase to support these new cloud economics is a considerable challenge and will take time to achieve. Work is already underway in the OpenJDK project to make Hotspot more "cloud friendly," but it's going to take time because many Hotspot features will need to be substantially redesigned to deal with supporting two conflicting performance profiles.

The two main open source JVMs used in the industry are Hotspot (from the OpenJDK project and contributed by Oracle) and Eclipse OpenJ9 (contributed to the Eclipse Foundation by IBM). Most of us are familiar with Hotspot but not so much OpenJ9. OpenJ9 is the codebase of the JVM that IBM and its customers (some of the largest enterprises in the world) have been using for over 20 years. It is designed to work on a wide variety of systems, from mobile phones to mainframes. The need to deal with such a range of environments has resulted in a design that is flexible and frugal. For instance, the runtime profile required by mobile phones is for applications to start fast and use as little memory as possible. OpenJ9 can typically achieve memory savings of around 50% less than Hotspot for the same workload. Sound familiar? Yes, these are the same characteristics we described as being important for cloud native applications. This is why for our applications we have chosen to use OpenJ9.

With OpenJ9 on the scene, the Java community now has two choices of open source, robust, enterprise-capable JVMs for running Java applications.

Where Do You Get a Java Runtime?

Whichever JVM you choose, you should consider where to access it and how to maintain and support it.

There's been a flurry of recent announcements from various Java vendors offering an OpenJDK-based Java runtime. In addition to the existing commercial offerings from Oracle, IBM, and more, there are now many freely available, free-to-use alternatives. This is all good news for users. The choice of vendor is now much wider than ever before. Still, there are a few points to consider when making your decision.

First, although JVM behavior is well specified, the rest of the Java runtime is less so. Alternative JVMs are easy options, but the remainder of the runtime (what is usually referred to as the *class libraries*) is not readily plug replaceable. This means it is imperative to understand how the vendor deals with accessing and maintaining the class libraries. Given the impracticalities of having a third-party version of the class libraries, the vendor will have retrieved that code from the OpenJDK project. There are various routes for how this may occur. The main consideration is the latencies between the OpenJDK codebase and the corresponding vendor binaries. Delays in the process will increase the likelihood of your Java choice missing bug fixes or security patches.

Another consideration is the mechanism and frequency with which the vendor sends bug fixes to the OpenJDK project. Delays here can mean that bug fixes in the vendor's offering may not be present in offerings from other vendors.

The final consideration is about testing. How does the vendor go about ensuring their offering is compatible and comparable with other runtimes?

Driven by a desire to have consistent and compatible Java runtimes across the multiple Java distributions available today, the London Java Community began an initiative that has grown into what is now seen as *the* source for free, open, and supported Java. This initiative is called AdoptOpenJDK.

The AdoptOpenJDK website provides daily built binaries for all the Java versions currently in play. With a choice of Hotspot or OpenJ9 downloads (or even Docker images), AdoptOpenJDK provides a one-stop shop for free Java. AdoptOpenJDK has many sponsors, including some major technology vendors such as IBM and Microsoft, as well as many others that are more consumer focused. All the sponsors share a desire for consistent and well-tested Java runtimes that are frequently updated and will be available for a long time.

The AdoptOpenJDK community is providing a consistent choice for end users and long-term support with security fixes for free. With its focus on daily updates, a close relationship with the OpenJDK community, and probably the most comprehensive testing regime, AdoptOpenJDK is an great choice for Java binaries for your cloud native Java applications.

Cloud Native Environments

Modern applications are typically deployed and executed using cloud native technologies such as containers, container orchestration, and service meshes. These offer a much more effective and manageable way of operating production-grade systems. Docker and Kubernetes in particular have emerged as de facto standards for container and orchestration technology. Service mesh technology, such as Istio, extends these approaches and introduces additional cross-cutting concerns such as security or advanced traffic management on a networking level.

In this book, we'll be focusing on the foundations of cloud native application development; the business logic and everything that is mainly written in Java. While we'll cover integration points of our Java applications with these cloud native environments, the details of their use are beyond the book's scope.

Continuous Delivery and Engineering Practices

In order to quickly deliver software with quality and reliability, modern applications are built and shipped using *continuous delivery* (CD) pipelines. Every step required to get from the source code to the running production system should be automated and run without human intervention. Automated software tests need to be part of the pipeline, and they need to verify every previous step as well as the functional and nonfunctional requirements of the application.

The goal of CD is that every software build that passes the whole pipeline is ready for production and will deliver value to its users.

Implementing CD is certainly not unique to cloud native applications; however, with the requirements of a modern application in a fast-moving world, shipping with automation, speed, and predictability is more important than ever.

Cloud native technology facilitates building and deploying software in an automated way, thanks to the great support of Infrastructure-as-Code and automation in technologies such as Docker and Kubernetes.

Microservice architectures are developed by cross-functional teams with a one-to-one ownership of team to application (microservice), with each team likely owning multiple microservices and the responsibilities of developing, maintaining, and monitoring those microservices. Following the "You build it, you run it" mantra of DevOps, teams consist of cross-functional engineers with different focus areas.

CD is essential to successful cloud native applications. However, this book focuses on the foundational development aspects of cloud native and so we will not discuss it further.

Summary

In this chapter, we started by outlining the important principles of good open technology around open source, open standards, and open governance. We then showed how to use these principles to evaluate open source and open standards choices. Finally, we talked about the lead role the Eclipse Foundation has taken in producing open cloud native technologies, detailing our choices of Eclipse Jakarta EE and Eclipse MicroProfile.

Lastly, we talked about the role of the JVM and the runtime characteristics required by cloud native applications. We introduced the Eclipse OpenJ9 JVM and discussed how its runtime profile of fast startup and low memory footprint makes it a good choice for cloud native Java applications. We also introduced AdoptOpenJDK as a reliable source of prebuilt OpenJDK binaries.

In the next chapter, we'll start getting our hands dirty in the code. We'll begin by diving further into the Jakarta EE and MicroProfile technologies for implementing a REST service backed by a database.

In our implementation, we have chosen to use the Open Liberty runtime. Open Liberty is a leader in implementing the Java EE and MicroProfile specifications. It starts fast and can be customized to include only the runtime components you need, making it great for lightweight cloud native microservices. Because our code is using Open Liberty through the Java EE and MicroProfile APIs, if we change our mind, it's relatively easy to switch to one of the many other implementations available to us.

CHAPTER 3
Foundation

In this chapter, we're going to lay the foundation for developing cloud native microservice applications. First we'll focus on how to implement the business logic in plain Java with as little coupling to framework-specific APIs as possible. We will then look at the edges of the applications that communicate with other applications and databases. We'll see how to persist our business objects and how to implement HTTP-based services.

When it comes to implementing the communication boundaries, there are two main approaches to developing cloud native microservices: *contract-first* and *implementation-first*. With contract-first, the service API (the contract) is defined—for example, through Swagger or OpenAPI—and then used to generate the service implementation skeleton. With implementation-first, the service is implemented and then the contract generated (e.g., though runtime- or tools-based OpenAPI generation). Both are valid approaches, but we mainly see developers using implementation-first, which is the approach taken in this chapter.

Rapidly Developing Service Implementations

Let's dive into implementation with an example application based on a coffee shop. The first aspect enterprise developers should focus on is implementing the business logic—not on cross-cutting concerns, integration, observability, or anything else for now, but only on what adds value to the application and its users. In other words, at the core of our microservices we start from a plain Java view and only model and implement components that directly relate to our business use case.

The convenience of Enterprise Java is that the programming model adds little weight to our individual classes and methods, thanks to the declarative approaches of both Jakarta EE and Eclipse MicroProfile. Typically, the classes of our core domain logic are simple Java classes that are merely enhanced with a few annotations.

This is why in this book we start by only covering Java and CDI, then gradually add more specifications as our application requires some more cross-cutting features. With this plain approach you can achieve a lot.

Implementing Domain Classes Using CDI

In our coffee-shop application, one of the entry points for our use case, sometimes referred to as a *boundary*, is a class called `CoffeeShop`. This class implements functionality to order a cup of coffee or retrieve the status of previous orders:

```
public class CoffeeShop {

    @Inject
    Orders orders;

    @Inject
    Barista barista;

    public List<CoffeeOrder> getOrders() {
        return orders.retrieveAll();
    }

    public CoffeeOrder getOrder(UUID id) {
        return orders.retrieve(id);
    }

    public CoffeeOrder orderCoffee(CoffeeOrder order) {
        OrderStatus status = barista.brewCoffee(order);
        order.setStatus(status);

        orders.store(order.getId(), order);
        return order;
    }

    public void processUnfinishedOrders() {
        // ...
        }
    }
```

The `CoffeeShop` class exposes the use cases for ordering a coffee, retrieving a list of all orders or a single one, and processing unfinished orders. It defines two dependencies, `Orders` and `Barista`, to which it delegates the further execution.

As you can see, the only Enterprise Java–specific declarations are the injections of our dependencies via `@Inject`. Dependency injection, as well as inversion of control in general, is one of the most useful patterns for developing our applications. We developers are not required to instantiate and wire dependent components, including all their transitive dependencies, which means we can focus on efficiently writing the business domain logic. We define the dependencies as "we need to use this component in our class" without regard to the instantiation. The life cycle of our instances, or beans, is managed by CDI.

TIP

You can mix and match using CDI beans in different scopes and injecting them as needed. The injection framework makes sure that all combinations work as desired.

The `CoffeeOrder`, which represents the entities of our domain, is written using plain Java only for now. It's a POJO (plain old Java object) containing properties for the order ID, type, and status:

```
public class CoffeeOrder {

    private final UUID id = UUID.randomUUID();
    private CoffeeType type;
    private OrderStatus status;

    // getters & setters ...

}
```

The `CoffeeType` and `OrderStatus` types are Java enums that define the available types of drinks (`ESPRESSO`, `LATTE`, `POUR_OVER`) and their order statuses (`PREPARING`, `FINISHED`, `COLLECTED`).

The components that implement our business logic should be tested well. Writing test cases is beyond the scope of this book. However, with the approach of plain Java first, we can efficiently develop test cases that cover the majority of our business logic by using test frameworks such as JUnit and mocking frameworks such as EasyMock or Mockito.

Scopes

Besides dependency injection, CDI also enables us to define the scope of the beans. The bean's scope determines its life cycle—for example, when it will be created and when it will be destroyed. Instances of the `CoffeeShop` class are created with an implicit *dependent* scope; that is, the scope is dependent on the scope of whatever uses them. If, for example, a request-scoped HTTP endpoint is injected with our `CoffeeShop` bean, the `CoffeeShop` instance life cycle will also exist within the same request scope.

If we need to define a different scope, say, for a class that exists only once in our application, we annotate the class accordingly. The following example shows the application-scoped `Orders` class:

```
@ApplicationScoped
public class Orders {

    private final ConcurrentHashMap<UUID, CoffeeOrder> orders =
            new ConcurrentHashMap<>();

    public List<CoffeeOrder> retrieveAll() {
        return orders.entrySet().stream()
                .map(Map.Entry::getValue)
                .collect(Collectors.toList());
    }

    public CoffeeOrder retrieve(UUID id) {
        return orders.get(id);
    }

    public void store(UUID id, CoffeeOrder order) {
        orders.put(id, order);
    }

    public List<CoffeeOrder> getUnfinishedOrders() {
        return orders.values().stream()
                .filter(o -> o.getStatus()
                        != OrderStatus.COLLECTED)
                .collect(Collectors.toList());
    }
}
```

The `Orders` class is responsible for storing and retrieving coffee orders, including their status. The `@ApplicationScoped` annotation declares that there is to be one instance of the `Orders` bean. No matter how many injection points we have in our application—`CoffeeShop` being one of them—they will always be injected with the same instance.

> Be aware of the default concurrency management of Enterprise Java Bean (EJB) singletons if you're using them instead of application-scoped CDI beans. CDI beans don't manage concurrency, and it's the developer's responsibility to make them thread-safe.

The most commonly used scopes that are available in CDI are dependent, request, application, and session. If for some reason these capabilities are not enough, developers can write their own scopes and extend the features of CDI. In a typical enterprise application, however, this is seldom required.

Configuration

For a typical application we'll need to configure a few things, such as how to look up and access external systems, how to connect to databases, or which credentials to use. The good news is that in a cloud native world, we can externalize a lot of different kinds of configuration from the application level to the environment. We don't have to configure and change the application binaries; instead we can have the different configuration values injected from the environment (e.g., such as Kubernetes ConfigMaps).

As developers, we want to focus on configuration that relates to the application business logic. Depending on your business, your applications might be required to behave differently in different environments.

In general, we want to be able to inject configuration values with minimal developer effort. We just covered dependency injection, and ideally, we'd like to have a similar way to inject configured values into our code.

With CDI we could write CDI producers that look up our configured values and make them available. But there's an even easier method: using MicroProfile Config.

MicroProfile Config

MicroProfile Config defines functionality that allows developers to easily inject configured values, just like we can inject service or other beans using CDI. For example, it ships with default config sources for environment variables that we can use right away. Available environment values in our systems are loaded and ready to be injected without any further developer effort.

Let's assume that we want to enhance our coffee-shop example to define default coffee drinks if the clients don't say which type of coffee they'd like to have. To do that, we set some default `CoffeeType` in our `CoffeeOrders` if the provided type is empty.

Have a look at our updated `CoffeeShop` class:

```java
public class CoffeeShop {

    @Inject
    @ConfigProperty(name = "coffeeShop.order.defaultCoffeeType",
            defaultValue = "ESPRESSO")
    private CoffeeType defaultCoffeeType;

    // ...

    public CoffeeOrder orderCoffee(CoffeeOrder order) {
        setDefaultType(order);
        OrderStatus status = barista.brewCoffee(order);
        order.setStatus(status);

        orders.store(order.getId(), order);
        return order;
    }

    private void setDefaultType(CoffeeOrder order) {
        if (order.getType() == null)
            order.setType(defaultCoffeeType);
    }

    // ...
}
```

The `CoffeeType` is resolved from the environment variable `coffeeShop.order.defaultCoffeeType`. Some operating systems don't support dots in their variable names, which is why MicroProfile Config supports multiple ways of replacing the dots with underscores. We could thus define this variable as `COFFEESHOP_ORDER_DEFAULTCOFFEETYPE`. If that environment variable is not set in the running application, the value will default to ESPRESSO.

The CoffeeType enum defines multiple values that can be resolved by the string representations, and so we can choose the ESPRESSO string representation as the default.

Persisting Service Data

We just saw how to implement the main business logic components into our applications. Besides stateless processing logic, most applications require us to persist state, usually the domain entities that represent the core of our business.

There are many database technologies to choose from. In this chapter we want to focus on *relational database management systems* (RDBMSes), which offer an arguably straightforward way of persisting domain objects that, in our experience, covers the majority of use cases.

Java Persistence API

In the Enterprise Java world, the Java Persistence API (JPA) is one of the most widely used technologies to persist domain entities. It offers an effective, declarative way to map types and their properties to relational database tables. JPA integrates well with models that are built following the concepts of domain-driven design. Persisting entities doesn't introduce much code overhead and doesn't overly constrain the modeling. This enables us to construct the domain model first, focusing on the business aspects, and integrate the persistence functionality afterward.

JPA's main concepts are the entity beans, which represent the individual persisted domain entities, and the entity manager, which is responsible for storing and retrieving entity beans.

Mapping domain models

JPA enables us to directly map our domain entities as well as aggregates to the database. In order to do that, we define domain types like the CoffeeOrder as JPA entity beans:

```
@Entity
@Table(name = "orders")
public class CoffeeOrder {
    @Id
    private String id;

    @Basic(optional = false)
    @Enumerated(EnumType.STRING)
    @Column(name = "coffee_type")
    private CoffeeType type;

    @Basic(optional = false)
    @Enumerated(EnumType.STRING)
    private OrderStatus orderStatus;

    // getters & setters

}
```

The `@Entity` annotation defines the `CoffeeOrder` as an entity bean. Each entity bean is required to define an identity field for instances of the entity bean—that is, contain an ID property, annotated with `@Id`. The individual fields are usually mapped to database columns, which are also configured declaratively, using the corresponding annotations. Full examples can be found in the JPA documentation.

The `CoffeeOrder` example will persist our coffee order to the `orders` table, with the enumerations persisted as string representations.

Managing persisted entities

The `EntityManager` is the main entry point that manages the persistence of our entities. Our business logic invokes its functionality during the processing of a coffee order use case:

```
@ApplicationScoped
public class CoffeeShop {

    @PersistenceContext
    EntityManager entityManager;

    // ...

    @Transactional
    public CoffeeOrder orderCoffee(CoffeeOrder order) {
        order.setId(UUID.randomUUID().toString());
        setDefaultType(order);
        OrderStatus status = barista.brewCoffee(order);
        order.setOrderStatus(status);

        return entityManager.merge(order);
    }
    }
```

The `merge` operation makes the coffee order a managed entity, meaning JPA will manage storing and retrieving it from the database.

The `@Transactional` annotation states that the `orderCoffee` is to be executed within a transaction. The default for the annotation is to require a transaction, so if there isn't one already, a new one will be started. If a new transaction is started, once this method finishes execution (returns) the container will automatically commit the transaction, and the coffee order will be persisted to the database. The `merge` operation causes the coffee order to be persisted.

If we define aggregate entities that contain not only primitive types or value objects but also references to other entities, the persist operations are invoked on the root entities and cascaded to the referenced subentities. The difference between entities and value objects, such as strings, enumerations, or currency values, is the notation of identity. It doesn't make a difference to (most) businesses which instance of a 10 euro bill we refer to, but it does make a difference which coffee order has just been completed successfully. The latter needs to be identified individually and thus represents a domain entity.

The `EntityManager` implements the DAO (Data Access Object) pattern. Depending on the complexity of the invocations made on the `EntityManager` type, it is often not necessary to encapsulate them in a separate DAO-like type.

Integrating RDBMSes

Our basic example shows the persistence configuration that is required on the project code level. In order to integrate the database into our application, we need to define the data source, in other words, how to connect to the database.

Ideally, we can abstract the detailed configuration from our application configuration. As we saw earlier, environment-specific configuration should not be part of the application code but rather managed by the infrastructure.

JPA manages the persistence of entities within persistence contexts. The entity manager of a persistence context acts as a cache for the entities and uses a single persistence unit corresponding to a database instance. If only one database instance is used within the application, the entity manager can be obtained directly, as shown in the example, without specifying the persistence unit.

Persistence units are specified in the *persistence.xml* file, which resides in the *META-INF* directory of our project:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/
 persistence/persistence_2_2.xsd">
  <persistence-unit name="coffee-orders" transaction-
type="JTA">
  </persistence-unit>
</persistence>
```

This example configures a persistence unit that doesn't specify a data source. Our application container is required to define a default data source, so that is what is used here. With this approach we can decouple the infrastructure configuration from our application binary build. This means we only need to build the application once and can change the configuration as we take it through testing, staging, and into production—a best practice in the world of cloud native microservices.

If we use multiple data sources in our application, we define multiple persistence units and refer to the data sources via JDNI:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/
persistence/persistence_2_2.xsd">
  <persistence-unit name="coffee-orders" transaction-
type="JTA">
    <jta-data-source>jdbc/CoffeeOrdersDB</jta-data-source>
  </persistence-unit>
  <persistence-unit name="customers" transaction-type="JTA">
    <jta-data-source>jdbc/CustomersDB</jta-data-source>
  </persistence-unit>
</persistence>
```

In this case, we are required to qualify the `EntityManager` lookups with the corresponding persistence unit:

```
@PersistenceContext(unitName = "coffee-orders")
EntityManager entityManager;
```

Transactions

As mentioned before, Enterprise Java makes it easy to execute business logic within transactions. This is required once we make use of relational databases and when we want to ensure that our data is stored in an all-or-nothing fashion. In one way or another, the majority of enterprise applications require ACID (atomic, consistent, isolated, durable) transactions.

In a distributed system, a business use case might involve multiple external systems, databases, or backend services. Traditionally, these *distributed transactions* have relied on the use of a two-phase commit (2PC) protocol to coordinate updates across the external systems. Achieving this consistency across distributed systems takes time and resources, and thus it comes at the cost of availability. In modern internet-scale systems, availability is often key, so other techniques based around the goal of *eventual consistency* have been employed. These include patterns such as Sagas and CQRS (command query responsibility separation). In moving to an eventual consistency model, a system becomes more loosely coupled and responsive, with the caveat that the data may be a little stale. For a more detailed understanding of these principles, we recommend you look at the literature on CAP theorem.[1]

In order to guarantee data consistency, our systems typically require us to use transactions in which a single database participates. As we've seen in the example, the `@Transactional` annotation enables this functionality without requiring developers to write boilerplate code or extensive configuration. If required, we can further refine how multiple, nested methods are executed. For example, methods that are executed within an active transaction can suspend the transaction and start a new transaction that is active during their execution, or they can be part of an existing transaction. For further information, have a closer look at the semantics of the parameters of

1 For more information, see this Illustrated Proof of the CAP theorem.

the `@Transactional` annotation and the Java Transaction API (JTA) specification.

TIP

In our `CoffeeShop` example, we used a CDI bean. An alternative approach, although less in favor nowadays, would have been to use a session EJB. With session EJBs, by default, all the EJB methods are transactional, so there would be no need for the `@Transactional` annotation.

Implementing REST Services

Now that you've seen how our domain entities can be persisted, let's have a look at how to integrate other applications using HTTP-based communication. You'll learn how to implement HTTP-based services using JAX-RS, how to map entities to transfer objects, and how applications can benefit from the ideas behind hypermedia.

Once we have implemented our business logic and potentially implemented persistence, we have to further integrate our application into our system. Business logic that is not accessible from outside the application mostly provides little value, and typically enterprise systems are required to provide endpoints to communicate with other systems.

The following example shows how to implement REST services using Enterprise Java technology. We'll implement JAX-RS resource classes that handle the HTTP functionality and make our business logic accessible outside of the application.

Boundary Classes

JAX-RS resource classes typically represent the boundaries, or the entry points, of our business use cases. Clients make HTTP requests and thus start a specific business process in the backend.

The following shows a JAX-RS resource class that implements the HTTP handling for retrieving coffee orders:

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;

@Path("/orders")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
```

```
public class OrdersResource {

    @Inject
    CoffeeShop coffeeShop;

    @GET
    public List<CoffeeOrder> getOrders() {
        return coffeeShop.getOrders();
    }
}
```

The `@Path` annotation declares the class as a JAX-RS resource for handling the `/orders` URL path. There are annotations for all standard HTTP verbs, such as `@GET`, `@POST`, and `@HEAD`. These annotations declare a method as a JAX-RS resource method. Once a client makes an HTTP `GET` request to the `/orders` resource, the request will be handled in the `getOrders` method of this class.

JAX-RS automatically maps Java objects to HTTP requests and responses. It supports content negotiation; that is, clients can tell the servers which content type they use for request bodies, and which one they expect for the server's responses. This example resource class supports JSON, which is specified by the `@Consumes` and `@Produces` annotations. Developers can declare further content type capabilities by implementing and registering the `MessageBodyWriter` and `MessageBodyReader` types in JAX-RS, but for most microservices, JSON is the format used. We must pay attention so as not to use the wrong import for the `@Produces` annotation, since CDI also defines one with the same name but different behavior.

In order to create some coffee orders, clients typically create new resources by sending `POST` requests to the backend's URLs. The following shows the JAX-RS resource method for creating new orders:

```
@Path("/orders")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class OrdersResource {

    @Inject
    CoffeeShop coffeeShop;

    @Context
    UriInfo uriInfo;

    @POST
    public Response orderCoffee(CoffeeOrder order) {
        final CoffeeOrder storedOrder = coffeeShop
            .orderCoffee(order);
        return Response.created(buildUri(storedOrder)).build();
    }

    private URI buildUri(CoffeeOrder order) {
        return uriInfo.getRequestUriBuilder()
                .path(OrdersResource.class)
                .path(OrdersResource.class, "getOrder")
                .build(order.getId());
    }
}
```

The `orderCoffee` resource method will receive the `POST` request with the coffee order as the body in the JSON content type and will then map it to an `CoffeeOrder` object. The resource method calls the business functionality of the boundary and returns a `Response` object, a wrapper object for HTTP responses with additional information, to indicate that the resource has been created successfully. The JAX-RS implementation will map this to the `201 Created` HTTP status code and the `Location` header field. We'll see in "Validating Resources" on page 38 how other response codes can be returned when the input data is invalid.

This is all we need to do to implement HTTP endpoints on our side.

Mapping Entities to JSON

By default, all properties of a business entity for which we define getter and setter methods are mapped to JSON. However, often we require some further control over how exactly the properties of an object are serialized, especially when the mapping slightly varies from what we represent in Java.

In Enterprise Java, there are two ways to map Java objects from and to JSON. The first way is to declaratively map the properties, using a standard called JSON-B (the Java API for JSON Binding). This is what we implicitly used in the previous examples. By default, this approach will map all properties for which an object defines getter and setter methods to JSON object fields. Nested object types are handled recursively.

The second way is to programmatically create or read JSON objects using a technology called JSON-P (the Java API for JSON Processing). JSON-P defines methods that we can call directly to create arbitrary objects. This approach provides the greatest flexibility in how objects are mapped. Let's look at an example of how to programmatically map our coffee order type:

```
@Path("/orders")
public class OrdersResource {

    @Inject
    CoffeeShop coffeeShop;

    @Context
    UriInfo uriInfo;

    @GET
    public JsonArray getOrders() {
        return coffeeShop.getOrders().stream()
                .map(this::buildOrder)
                .collect(JsonCollectors.toJsonArray());
    }

    private JsonObject buildOrder(CoffeeOrder order) {
        return Json.createObjectBuilder()
                .add("type", order.getType().name())
                .add("status", order.getStatus().name())
                .add("_self", buildUri(order).toString())
                .build();
    }

    private URI buildUri(CoffeeOrder order) {
        return uriInfo.getRequestUriBuilder()
                .path(OrdersResource.class)
                .path(OrdersResource.class, "getOrder")
                .build(order.getId());
    }
}
```

The `JsonArray` type defines a JSON array of arbitrary elements. The `getOrders` resource method maps the individual coffee orders to `JsonObject` and aggregates them into the array type. We can see how the object builder allows us to compose objects with our desired structure. The resulting JSON objects differ slightly from the declaratively mapped approach.

The question often arises as to when to use JSON-B or JSON-P. Typically, we find that the declarative approach of JSON-B is the simplest and covers most use cases.

If its default serialization is not what you need, then JSON-B also allows you to control how the Java objects are mapped to JSON, especially how and whether individual properties are being mapped.

The following example slightly modifies the mapping of our coffee order:

```
public class CoffeeOrder {

    @JsonbTransient
    private final UUID id = UUID.randomUUID();

    @JsonbTypeAdapter(CoffeeTypeDeserializer.class)
    private CoffeeType type;

    @JsonbProperty("status")
    private OrderStatus orderStatus;

    // methods omitted
}
```

The `@JsonbTransient` annotation declares the `id` field as transient; that is, this field will be ignored by JSON-B and neither be read from nor written to JSON. `@JsonbProperty` allows for further customization of the JSON object field name.

If the default mapping of a Java type doesn't work for us, we can always declare a custom type adapter, for example, using the `@JsonbTypeAdapter` annotation as shown in the previous example. You can have a look at the adapter for the coffee type enum in the `CoffeeTypeAdapter` class, found in the code example project.

These and a few other ways of customizing the JSON mapping built into JSON-B already fulfill a majority of cases. If more flexibility or control is required, we can instead programmatically create or read JSON structures using JSON-P. This approach is especially helpful when dealing with Hypermedia REST resources, as you'll see later in this chapter.

Validating Resources

Request data that is received from clients needs to be sanitized before it can be used further. For security reasons you should never trust the data that has come from an external source, such as a web form or REST request. In order to make it simple to validate input, Enterprise Java ships with the Bean Validation API, which allows us to declaratively configure the desired validation. The good news for developers is that this standard integrates seamlessly with the rest of the platform, including, for example, JAX-RS resources.

To ensure that only valid coffee orders are accepted in our application, we enhance our JAX-RS resource method with Bean Validation constraints:

```
...

@POST
public Response orderCoffee(@Valid @NotNull CoffeeOrder order) {
    final CoffeeOrder storedOrder = coffeeShop.
        orderCoffee(order);
    return Response.created(buildUri(storedOrder)).build();
}
```

The `@NotNull` annotation ensures that we'll receive properly populated orders. `@Valid` makes sure that the order itself is validated for potential subsequent validation constraints.

Let's look at our enhanced coffee order class that defines what makes a valid order:

```
public class CoffeeOrder {

    @JsonbTransient
    private final UUID id = UUID.randomUUID();

    @NotNull
    @JsonbTypeAdapter(CoffeeTypeDeserializer.class)
    private CoffeeType type;

    private OrderStatus status;

    // ... getters & setters
}
```

The type of a coffee order must not be `null` either; that is, clients must provide a valid enumeration value. The value is automatically mapped by the provided JSON-B type adapter, which returns a `null` if an invalid value is transmitted. Consequently, validation will fail for any invalid values.

JAX-RS integrates with Bean Validation such that if any constraint validations fail, an HTTP status code of `400 Bad Request` is automatically returned. Therefore, the presented example is already sufficient to ensure that only valid orders can be sent to our application.

REST and Hypermedia

Representational State Transfer (REST) provides an architectural style of web services that is often well suited to the needs and structure of web applications. The idea is to loosely couple applications with interfaces that are accessible in a uniform way, through URLs. There are a few REST constraints, such as the use of uniform interfaces, identification of individual resources (i.e., the entities in our domain), and the use of *Hypermedia As The Engine Of Application State*, commonly referred to as HATEOAS. The representations of the domain entities are modified in a uniform way—in HTTP, for example, using the methods `GET`, `POST`, `DELETE`, `PATCH`, and `PUT`.

Hypermedia allows for linking related resources and resource actions, and then making them accessible through URLs. If some HTTP resources are somewhat related to others, for example, they can be linked to each other and accessed through full URLs that the client directly follows. In this way, the server guides the clients through the available resources using semantic link relations. The clients don't need to know how the URLs are structured on the server. Roy T. Fielding described Hypermedia's usage in his dissertation, "Architectural Styles and the Design of Network-based Software."

The following shows a basic example of a coffee order representation in the JSON format:

```json
{
  "type": "ESPRESSO",
  "status": "PREPARING",
  "_links": {
    "self": "https://api.coffee.example.com/orders/123",
    "customer": "https://api.coffee.example.com/customers/234"
  }
}
```

The `_links` field of the representation gives some example links—to the coffee order resource itself and to the customer who created this order. If a client would like to know more about the customer, it would follow the provided URL in a subsequent `GET` request.

It's usually not sufficient to only follow links and read resources using `GET`, so you can also exchange information—say, on how to modify resources—using `POST` or `PUT` requests.

The following demonstrates an example Hypermedia response that uses the concept of actions. There are a few Hypermedia-aware content types that support these approaches, such as the Siren content type on which this example is based:

```
{
  "class": [ "coffee-order" ],
  "properties": {
    "type": "ESPRESSO",
    "status": "PREPARING"
  },
  "actions": [
    {
      "name": "cancel-order",
      "method": "POST",
      "href": "https://api.coffee.example.com/cancellations",
      "type": "application/json",
      "fields": [
        { "name": "reason", "type": "text" },
        { "name": "order", "type": "number", "value": 123 }
      ]
    }
  ],
  "links": [
    "self": "https://api.coffee.example.com/orders/123",
    "customer": "https://api.coffee.example.com/customers/234"
  ]
}
```

In this example, the server enables the client to cancel a coffee order and describes its usage in the `cancel-order` action. A new cancellation means the client would `POST` a JSON representation of a cancellation containing the order number and the reason to the provided URL. In this way, the client requires knowledge only of the `cancel-order` action and the origin of the provided information (i.e., the order number, which is given, and the cancellation reason, which is known only by the client and may be entered in a text field in the UI).

This is one example of a content type that enables the use of Hypermedia controls. There is no real standard format that the industry has agreed upon. However, this Siren-based example nicely demonstrates the concepts of links and actions. Whatever content type and representation structure is being used, the projects need to agree upon and document their usage. But as you can see, this way of structuring the web services requires far less documentation, since the usage of the API is baked into the resource representations already. This also greatly reduces the likelihood of API documentation becoming out of date with the code. We'll see later in Chapter 4 how OpenAPI can further improve this approach.

One of the benefits of using HATEOAS is that control over how the resources are accessed resides on the server side. The server owns the communication and is even free to change the URL structures, since the clients are no longer required to make any assumptions about how the URLs are being constructed.

Decoupling the communication also results in less duplication of business logic. Clients do not need to contain the logic for the conditions under which an order can be canceled; they can simply display the functionality for canceling orders if the corresponding action is provided in the HATEOAS response. Only the knowledge that is required to reside on the client side—for example, how the cancellation reason is provided—needs to be implemented on the client side (i.e., UI decisions).

Let's come back to how we map our domain entities to JSON on the server side. As you can see, the JSON structures for Hypermedia resources can become quite complex, which in turn may result in complex Java type hierarchies if we were to use the declarative approach of JSON-B. For this reason, it's worth considering the programmatic approach using JSON-P to create these Hypermedia resources instead. The code that creates the JSON-P objects representing the coffee orders can be factored out into a separate class with the single responsibility to remove redundancy in the JAX-RS resources, if required.

Summary

As you've seen in this chapter, we can already implement the vast majority of our enterprise application using plain Java and CDI. At its core, our business logic is written in plain Java with some dependency injection added to simplify defining dependent components. MicroProfile Config enables us to inject required configuration with minimal impact in the code. What's left is mainly integration into our overall enterprise system, as well as nonfunctional requirements such as resiliency and observability.

We saw how to integrate persistence into our applications using JPA and how to map domain entities to relational databases with minimal developer effort. Thanks to the previous specification work being done in the JTA standard, we can define transactional behavior without obscuring the business code.

We can implement REST endpoints using the JAX-RS standard with JAX-RS resources. The declarative programming model allows us to efficiently define the endpoints with default HTTP bindings. It also allows us to further customize the HTTP request and response mappings, if required.

Enterprise Java supports binding our entities to and from JSON, either declaratively using JSON-B or programmatically using JSON-P. Which approach makes more sense depends on the complexity of the entity representations. The requests can be validated using Bean Validation, which allows developers to specify the validation programmatically or declaratively as well. Enterprise developers might want to explore the concepts behind Hypermedia that allow further decoupling from the server, make the server resources discoverable, and make communication more flexible and adaptive.

CHAPTER 4

Cloud Native Development

In the previous chapter we saw how to develop REST services that are backed by a database. We discussed how Enterprise Java makes this simple, allowing the developer to focus largely on the business logics and use a small set of annotations to define database persistence and to provide and call REST services with JSON payloads.

This ability to use annotations to reduce the amount of coding required is great for small numbers of simple services, but you'll soon encounter limitations if you're scaling to tens or hundreds of services for which individual teams are responsible. Your business logic is now split across processes with remote APIs. These APIs will need to be appropriately secured. A client request potentially passes through tens of services, all managed independently and with networks in between, which adds the potential for latency and reliability problems. Your independent teams now need to be able to collaborate and communicate their service APIs. These are just some of the costs associated with cloud native development, but thankfully there are APIs and technologies available to help.

Securing REST Services

Background

It's important to start by recalling that by default the HTTP protocol is stateless. The protocol supports various "verbs" for requests (`GET`, `POST`, `PUT`, `DELETE`, etc.). These are the building blocks for the RESTful approach underpinning microservices. All calls are stateless, as they are simply requests to retrieve or modify state on the server. There is no capability within the protocol to define any sort of relationship between these calls. This design approach means that HTTP services can balance workload effectively across multiple servers (and the like) because any call can be routed to any available responder.

This stateless design is effective for public data where the caller can remain anonymous, but at some point it becomes essential to differentiate one client from another.

As mentioned before, prior to a client authenticating themselves, a service does not need to be able to differentiate between callers. They can remain anonymous and undifferentiated. Once a client is authenticated to the server, however, then they are no longer anonymous. The client may have particular powers to modify the state of the server; hence, the server must ensure there are appropriate controls in place to prevent hijacking of the communications between the user and the server.

Application architectures therefore face a continuous challenge in determining how to communicate securely and *statefully* with an authenticated client when the underlying protocol is stateless.

The Common Approach

Most application server frameworks provide a basic mechanism to achieve this via a session mechanism that stores a unique identifier in a cookie called `JSESSIONID` that is sent to the client. In this model the session ID is simply a randomly generated key that can be used by the client to show that its request is part of a previous conversation with the server.

This approach does work, but it has some significant weaknesses:

Server affinities
: The session ID is a key to more important data stored on the server. This is fine when there is only one server. When there are multiple servers that could handle the request from the client, though, they must somehow communicate to each other what are valid session IDs and what is the important related data. Otherwise, a client request routed to a new server would find that its `JSESSIONID` is not recognized and hence the request would be rejected.

Session ID hacking or spoofing
: The session ID on its own merely indicates to the server that the client has been seen before. The random nature of the session ID protects against simple spoofing. If it were a simple numeric value, it's easy to see how a malicious actor could create a fake session ID and try to break into an existing conversation between client and server. The random nature of the session ID prevents simple numerical attacks but does not prevent stolen session IDs from being reused.

Lack of access granularity
: Since the session ID is just a key to identifying the client, it does not restrict the request's capability. A stolen or hijacked session ID could be used to access the server in any way the client is authorized to do—even if unrelated to the original request from the client.

Multiple authentication
: `JSESSIONID` is a server concept. It is created by the application server and shared across related instances. If the client needs to talk to different services, then it will need to authenticate separately with them. If they too provide `JSESSIONID`s, then the client will need to manage multiple conversations to prevent repeated authentication. Reauthentication is both a waste of resources and a potential security risk.

Access propagation
: When the application itself is making a request to a service on behalf of the client, the `JSESSIONID` is not suitable to pass on to the next service. The application will need to authenticate with the new service on the client's behalf. In this case the application has to request additional authentication information from the client or use some preloaded authentication data. Neither of these options is particularly optimal, and both carry a risk of being potential security exposures.

Introducing JSON Web Tokens

In light of all the aforementioned weaknesses, much effort has been applied to creating an improved approach that addresses or reduces these concerns. As one example, in 2015 the IETF published rfc7519, which proposed a compact solution called *JSON Web Token* (JWT).

The JWT approach is based on providing an encoded, signed token that the client can use to access an application securely without needing the application to hold session state. The token is not specific to any application and can be passed to downstream services with no need for reauthentication. JWT tokens are readable and verifiable by anyone but, because they are signed, cannot be modified without detection.

A JWT token looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiw
iZW1haWwiOiJqb2VAbWFpbHNlcnZlci5jb20iLCJleHAiOjEyMDAxMTY0OTAsImd
yb3VwcyI6WyJtZW1iZXIiXX0.MR_71yUi80M9b_Hb9MnCrquuosvanX2hwggNsgV
cMe0
```

Looking closely, we can see the token is separated into three parts by periods (dots). Logically the token consists of a header, a payload, and a signature. The header and payload are base64 encoded. Once split and decoded, the token is more easily understandable.

Header

```
{
"alg": "HS256",
"typ": "JWT"
}
```

The JSON that makes up the header typically has two properties. The `"alg"` field defines the encryption algorithm used in the signature. The `"typ"` field specifies the type of the token, which by definition is `"JWT"`.

Payload

This section contains *claims*, which are optional. There are multiple predefined claims, some of which, although technically optional, are generally essential.

Claims are logically grouped into three types:

Registered claims
: Claims that are the most obviously useful or essential. The list includes the token expiration time, the token's issuer, and the subject or principle of the token.

Public claims

Claims intended to be shared between organizations and to be in some way domain specific. Public claims are registered in the IANA JSON Web Token Registry.

- *Private claims* cover all other claims that individuals and groups want for their own usage.

Here is a typical JWT claims example:

```
{
"sub"    :  "joe",
"email"  :  "joe@mailserver.com",
"exp"    :  1200116490 ,
"groups" :  [ "member" ]
}
```

This example shows the following claims:

- "sub" or subject, the value that will be returned via the Micro-Profile `JsonWebToken.getCallerPrinciple()` method.
- "email", a private claim that can be accessed by using `JsonWebToken.getClaim("email")`.
- "exp" or expiration date, the date and time after which this token is considered to be invalid.
- "groups", the list of groups or roles the subject is a member of. This can be automatically checked with the `@RolesAllowed` annotation.

Signature

The signature is an encrypted version of the header and payload joined together. It is created by base64 encoding the header and pay-load and then encrypting the result.

The signature can be signed by either a public or private key. In either case, the token recipient can easily assert that the data has not been modified. Of course, if the issuer signed the token with a pub-lic key, then any potential bad actor could create a fraudulent token by using the public key. So while it is technical feasible to use public keys, it is best to use a private key. Doing so provides additional proof that the token issuer is who they claim to be, as only they have the private key.

JWT with MicroProfile

Eclipse MicroProfile provides first-class support for generating and consuming JWT elements. There are easy-to-use annotations and methods for setting and reading group access, identities of participants, times of token issue and expiration, and, of course, setting and reading custom claims.

Enabling JWT as the authentication method

Using JWT with MicroProfile is straightforward. Use an annotation on the `Application` class to enable JWT as the login method:

```
@LoginConfig(authMethod = "MP-JWT")
public class CoffeeShopApplication extends Application {
```

Consuming JWT

On each endpoint class, use CDI to inject the current JWT instance:

```
@Path("/orders")
public class OrdersResource {

        @Inject
        private JsonWebToken jwtPrincipal;
```

Each endpoint using JWT support looks similar to the following example:

```
 @GET
    @RolesAllowed({"member"})
    @Path("coffeeTypes")
    public Response listSpecialistCoffeeTypes() {
    ...
```

Notice how by using the `@RoleAllowed` annotation we can check that the client user is in the required member group. If the client is not a member, then the server will automatically reject the request.

Additional benefits of using JWT

JWT can be used to store important information about the client that otherwise might have had to be cached on the server. Being able to reduce sensitive data stored on the server greatly helps in situations where the server has been compromised. It is hard to steal sensitive data if it is not actually there!

Imagine that in our coffee-shop example members can order special, super-strong coffees if they are 18 or over. Under normal circumstances, the server will ask for the user's date of birth, which will be stored in a database. Subsequent interactions with the user will require the server to retrieve the user's record to check their age and whether they are a coffee club member.

By including private claims in the token that confirms the user is 18 or over and a club member, the server can quickly check that the user is eligible for the special coffees without having to retrieve sensitive data.

This capability is particularly powerful. If the information is not something that should be revealed to the user or a third party, it can be encrypted inside the token.

In this example, the JWT token claims would look something like this:

```
"sub"    : "joe",
"name   ": "Joe Black",
"exp"    : 1200116490,
"groups" : [ "member" ]
"adult"  : "true"
```

The Java code checking the claim during the order would simply be:

```
@POST
  @RolesAllowed({"member"})
  @Path("/orderMemberCoffee")

  public Response orderMemberCoffee(CoffeeOrder order) {

    JsonValue claim=jwtPrincipal.getClaim("adult");

    if(claim==null || claim!=JsonValue.TRUE) {

      return Response.status(Response.Status.FORBIDDEN).build();

    }
    // normal processing of order
```

Encrypting claims

Since JWT contents are essentially public, if the claim information is sensitive, then it can be worthwhile to encrypt the contents of the claim and even obscure the claim name itself. In this example, the actual age of the user is needed:

```
"age" : "25"
```

Once encrypted and obfuscated, it appear as follows:

```
"a7a6a43128392fc" : "Bd+vK2AnxSNZduoGxFdbpBOfZ3mkPfBcw14t
4uU29nA="
```

Final Thoughts on JWT

JWT provides a strong mechanism for validating that the client has not forged any of the claims and is whom they say they are. However, like all publicly shared data in an HTTP or HTTPS request, the data in the JWT token can potentially be stolen and can be used as is against the service. Detecting this kind of spoofing is beyond the scope of this book, but it's important to know that it can and does occur.

Handling Service Faults

System outages in large businesses can cost them tens of thousands of dollars per minute in lost revenue. *High availability* (HA) is therefore critical to business success. Many businesses make significant investments with the goal of achieving four-nines availability (99.99% available, or less than 52 minutes, 36 seconds of outage per year) or even five-nines availability (99.999% available, or less than 5 minutes, 15 seconds of outage per year). You may be wondering what this has to do with microservices, and to answer that, we need to do some sums.

Consider a company running a monolithic application with five-nines availability. They split the application up into microservices, each deployed and managed independently. They calculate that on average each request to their application passes through 10 microservices and each individual microservice has five-nines availability. What's the overall availability of their microservice-based application? It's actually now only four nines.

This is the probability of a request being successful:

$$(0.99999)^{10} = 0.9999 = 99.99\,\%$$

If a request passes through 100 microservices, the availability of that request is actually only three nines, meaning 1 in 1,000 requests will encounter a problem—not great for customer satisfaction.

This is the probability of a request being successful:

$$(0.99999)^{100} = 0.999 = 99.9\%$$

The solution to this problem is to expect issues and handle them gracefully. If, as a client of a service, you can be tolerant of its faults and not propagate those issues back to your clients, then your availability is not impacted. Do this for all your service dependencies, and your overall availability is not impacted at all.

Fault tolerance is the concept of designing into a system the ability to gracefully handle faults. There are a number of different strategies for handling faults, and the approach you choose depends on the types of problems you might encounter and the purpose of the service being called. For example, a slow service might require a different strategy from a service that suffers intermittent outages.

MicroProfile Fault Tolerance implements a number of strategies, as summarized here:

Retry
: The Retry strategy is useful for short-lived transient failures. You can configure the number of times a service request will be retried and the time interval between retries.

Timeout
: Timeout allows you to time a request out before it completes. This is useful if you are calling a service that might not respond in a reasonable amount of time—for example, within your service's service level agreement (SLA) response time.

Fallback
: Fallback allows you to define an alternative action to take in the event of a failure—for example, calling an alternative service or returning cached data.

Circuit breaker
: A circuit breaker helps prevent repeated calls to a failing service. If a service begins to have issues, the circuit is opened and immediately fails requests until the service becomes stable and the circuit is closed.

Bulkhead
: Bulkhead is useful when you are calling a service that is at risk of being overloaded. You can restrict the number of current requests to a service and queue up or fail requests over this limit.

Asynchronous
: Asynchronous allows you to offload requests to separate threads and then use Futures to handle the responses. An object representing the result of an asynchronous computation, the Future can be used to retrieve a result once the computation has completed.

It's also possible to combine these policies for the same microservice dependency. For example, you can use Retry along with Fallback so that if the retries ultimately fail, you can call a fallback operation to return something useful.

Let's look at an example of MicroProfile Fault Tolerance. The following code is for a client of the `Barista` service:

```
@Retry
@Fallback(fallbackMethod="unknownBrewStatus")
public OrderStatus retrieveBrewStatus(CoffeeOrder order) {
    Response response
    = getBrewStatus(order.getId().toString());
    return readStatus(response);
}

private OrderStatus unknownBrewStatus(CoffeeOrder order) {
    return OrderStatus.UNKNOWN;
}

private Response getBrewStatus(String id) {
    return target.resolveTemplate("id", id)
            .request().get();
}
```

This code makes a remote call to the `Barista` service to retrieve the status of an order. The client may throw an exception, for example, if it is unable to connect to the `Barista` service. If this occurs, the `@Retry` annotation will cause the request to be retried, and in the event none of the request is successful, the `@Fallback` annotation causes the `unknownBrewStatus` method to be called, which returns `OrderStatus.UNKNOWN`.

Publishing and Consuming APIs

A common characteristic of companies that succeed with microservices is how they organize their teams. Spotify, for example, has *squads* that are responsible for each microservice; they manage the microservice from concept to development, from test to production, and finally to end of life.

Services don't live in isolation, so it's important for teams to be able to communicate to potential users what their services do and how to call them. Ideally that communication should be both human- and machine-readable, enabling a person to understand the service and a service client to easily call it, for example, by generating a service proxy at build time.

Open API is an open specification at the Linux Foundation designed to do just that. Open API is a standardization of Swagger contributed by SmartBear. It describes service APIs in either YAML or JSON format, and there are a number of tools that take these definitions and generate proxies or service stubs for various languages, including Java.

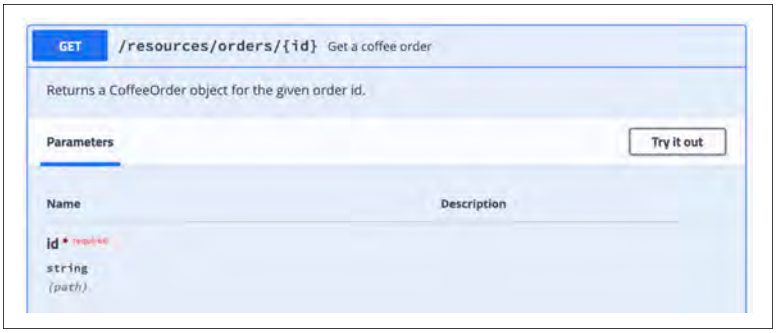
Rather than write Open API definitions from scratch, it's preferable to generate them from the service implementation. This is simpler for developers, as they don't need to be familiar with the Open API format or restate things already said in the code. It also ensures that the API definition is in sync with the code and that tests can be used to quickly flag breaking changes. It's possible to generate a machine-readable Open API definition directly for service implementations, such as those using JAX-RS and JSON-B. To augment the service definition with documentation, MicroProfile provides additional annotations that cover things such as operation documentation and API documentation URLs. The following example shows a JAX-RS/JSON-B service using the `@Operation` annotation to add a human-readable description:

```
@GET
@Path("{id}")
@Operation(summary="Get a coffee order",
  description=
  "Returns a CoffeeOrder object for the given order id.")
public CoffeeOrder getOrder(@PathParam("id") UUID id) {
  return coffeeShop.getOrder(id);
}
```

The resulting OpenAPI YAML definition would be as follows. For the sake of brevity, only the definitions relating to the `getOrder` method are shown:

```yaml
openapi: 3.0.0
info:
  title: Deployed APIs
  version: 1.0.0
servers:
- url: http://localhost:9080/coffee-shop
- url: https://localhost:9443/coffee-shop
paths:
  /resources/orders/{id}:
    get:
      summary: Get a coffee order
      description: Returns a CoffeeOrder object for the given...
      operationId: getOrder
      parameters:
      - name: id
        in: path
        required: true
        schema:
          type: string
          format: uuid
      responses:
        default:
          description: default response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/CoffeeOrder'
components:
  schemas:
    CoffeeOrder:
      type: object
      properties:
        id:
          type: string
          format: uuid
        type:
          type: string
          enum:
          - ESPRESSO
          - LATTE
          - POUR_OVER
        orderStatus:
          type: string
          enum:
          - PREPARING
          - FINISHED
          - COLLECTED
```

Some environments also provide a UI so that you can try out the API. Figure 4-1 shows the OpenAPI UI for retrieving a coffee order. Clicking "Try it out" allows you to enter the order `id` and get back the `CoffeeOrder` JSON.



Returns a CoffeeOrder object for the given order id.

Figure 4-1. OpenAPI UI for retrieving coffee orders

Summary

In this chapter we've discussed a number of areas you need to focus on when developing cloud native microservices: end-to-end security through your microservices flow, graceful handling of network and service availability issues to prevent cascading failures, and simple sharing and use of microservices APIs between teams. While these areas aren't unique to the microservices world, they're essential to success within it. Without these approaches, your microservice teams will struggle to share and collaborate while remaining autonomous.

We've shown how using the open standards of JWT and Open API and their integration into Enterprise Java through MicroProfile, along with MicroProfile's easy-to-use Fault Tolerance strategies, makes it relatively easy to address these requirements. For additional step by step instructions on how to build a cloud native microservices application in Java, please visit *ibm.biz/oreilly-cloud-native-start*.

In the next chapter we'll move on to cloud native microservice deployment and how to take to make your services observable so you can detect, analyze, and resolve problems encountered in production.

CHAPTER 5

Running in Production

Observability refers to the ability to continually monitor your micro-service to understand how it's performing, predict problems, and diagnose issues. Monitoring your runtime and application code is nothing new, but in a cloud native world you need to be able to do so with tens, hundreds, or thousands more instances and in cloud native deployment environments (e.g., containers and Kubernetes).

A common question that arises around observability is performance impact. We caution against compromising observability for the sake of performance. Modern JVMs incur only a small overhead when capturing runtime metrics, and the alternative is to essentially "fly blind." The last thing you need is for your service to go wrong, which you find out only when people start banging on your door, and you realize you have absolutely no diagnostic information to resolve the problem.

Eclipse MicroProfile provides three APIs to help monitor microser-vices: MicroProfile Health, MicroProfile Metrics, and MicroProfile OpenTracing. Each covers different aspects of observability, from the availability of your service to the performance of the service runtime.

In this chapter we'll explain how to use MicroProfile to make your microservices observable. We'll also discuss how to integrate your microservices with the Kubernetes infrastructure to enable Kuber-netes to manage your service life cycle and traffic delivery. Finally, we'll show how to hook up your microservices to monitoring and alerting tools to enable proactive detection and management of issues.

Reporting Health

It's common practice to report the health of a service through a REST endpoint. Kubernetes liveness and readiness probes can be configured to call these endpoints to get the health status of a service and take appropriate action. For example, if a service reports itself as not being ready, then Kubernetes will not deliver work to it. If a check for liveness fails, then Kubernetes will kill and restart the container.

Because of these different liveness and readiness remediation strategies, the types of health checks you perform will likewise differ. For example, readiness should be based on transient events that are outside your container's control, such as a required service or database being unavailable (presumably temporarily). Liveness, however, should check for things that are unlikely to go away without a container restart—for example, running low on memory.

The MicroProfile Health API takes away the need to understand which HTTP responses to provide for the different health states, allowing you to focus on just the code required to determine whether or not the service is healthy.

The next example is a readiness health check for the `coffee-shop` service, denoted by the `@Readiness` annotation.

```
@Readiness
@ApplicationScoped
public class HealthResource implements HealthCheck {

    private boolean isHealthy() {

        try {
            Client client = ClientBuilder.newClient();
            WebTarget target =
              client.target(
              "http://barista:9080/barista/resources/brews");
            Response response = target.request().get();

            int status = response.getStatus();
            if (status != 200) {
              return false;
            } else {
              System.err.println(status);
```

```
                }
                return true;
            } catch (Exception e) {
                e.printStackTrace();
                return false;
            }
        }

        @Override
        public HealthCheckResponse call() {
            boolean up = isHealthy();

            return HealthCheckResponse.named("coffee-shop")
                                      .withData("barista",
                                       String.valueOf(up))
                                      .state(up).build();
        }
    }
```

Because the `coffee-shop` service requires the `barista` service in order to work, the health check determines whether the `barista` is available by calling the `GET` method on the `brews` resource. So long as the service returns an HTTP `200 OK`, the health check reports its status as `UP`.

> **TIP**
>
> Early versions of MicroProfile Health supported only one type of health check: `@Health`. Use MicroProfile Health 1.3, which adds `@Readiness` and `@Liveness` to match the Kubernetes readiness and liveness probe concepts.

When the preceding health check succeeds, the health endpoint returns the `200 OK` and the following JSON:

```
{
  "checks": [
    {
      "data": {
        "barista": "true"
      },
      "name": "coffee-shop",
      "state": "UP"
    }
  ],
  "outcome": "UP"
}
```

If the health check fails, it returns `503 SERVICE UNAVAILABLE` with the following JSON:

```
{
  "checks": [
    {
      "data": {
        "barista": "false"
      },
      "name": "coffee-shop",
      "state": "DOWN"
    }
  ],
  "outcome": "DOWN"
}
```

A service can implement multiple health checks. The overall health response is an aggregation (logical `AND`) of all the checks. If any one of the checks is `DOWN`, then the overall outcome is reported as `DOWN`.

Kubernetes Integration

As mentioned earlier, MicroProfile Health is designed to work seamlessly with Kubernetes. Kubernetes allows you to configure two types of health probe when you deploy your microservice: readiness and liveness. The extracts of YAML in the next example show the configuration for readiness problems for the `coffee-shop` service. An initial delay is set to give the service sufficient time to start and report its true status. After this delay, Kubernetes will check the readiness every five seconds, and if it returns a `503 SERVICE UNAVAILABLE` HTTP response code, then Kubernetes will stop delivering requests to it:

```
spec:
  containers:
  - name: coffee-shop-container
    image: example.com/coffee-shop:1
    ports:
    - containerPort: 9080
    # system probe
    readinessProbe:
      httpGet:
        path: /health/ready
        port: 9080
      initialDelaySeconds: 15
      periodSeconds: 5
      failureThreshold: 1
```

The next block of YAML shows an example configuration of a liveness probe for the coffee-shop services. As with the readiness probe, delay and sampling periods are specified. The key difference is the actions Kubernetes will take if the liveness check fails. In this case, Kubernetes will assume the container is sick and kill and restart it:

```
spec:
  containers:
  - name: coffee-shop-container
    image: example.com/coffee-shop:1
    ports:
    - containerPort: 9080
    # system probe
    livenessProbe:
      httpGet:
        path: /health/live
        port: 9080
      initialDelaySeconds: 15
      periodSeconds: 5
      failureThreshold: 1
```

Monitoring JVM, Runtime, and Application Metrics

Understanding how well your services are performing and behaving is essential to ensuring their smooth operation. To enable this, MicroProfile Metrics provides an endpoint for retrieving metrics about your running service. By default, you can retrieve JVM metrics, prefixed with base:, and these are produced in a Prometheus format. The output can then be used to set up alerts or provide dashboards—for example, through Grafana.

Additional metrics can also be generated, including runtime metrics, prefixed with vendor: and application metrics, prefixed with application:. Some application metrics can be automatically generated—for example, metrics about Fault Tolerance retries—but MicroProfile also provides APIs for instrumenting the service for additional application metrics.

The following is an extract of the metrics output from the coffee-shop services. Only a subset of the data available is shown:

```
# TYPE base:cpu_system_load_average gauge
# HELP base:cpu_system_load_average Displays the system load
# average for the last minute. The system load average is the
# sum of the number of runnable entities queued to the available
# processors and the number of runnable entities running on the
# available processors averaged over a period of time. The way
# in which the load average is calculated is operating system
# specific but is typically a damped time-dependent average. If
# the load average is not available, a negative value is
# displayed. This attribute is designed to provide a hint
# about the system load and may be queried frequently.
# The load average may be unavailable on platforms
# where it is expensive to implement this method.
base:cpu_system_load_average 0.34
# TYPE base:thread_count counter
# HELP base:thread_count Displays the current number of live
# threads including both daemon and non-daemon threads.
base:thread_count 82
# TYPE base:classloader_current_loaded_class_count counter
# HELP base:classloader_current_loaded_class_count Displays the
# number of classes that are currently loaded in the Java
# virtual machine.
base:classloader_current_loaded_class_count 17785
# TYPE base:memory_committed_heap_bytes gauge
# HELP base:memory_committed_heap_bytes Displays the amount of
# memory in bytes that is committed for the Java virtual
# machine to use. This amount of memory is guaranteed for the
# Java virtual machine to use.
base:memory_committed_heap_bytes 1.23011072E8
# TYPE base:thread_daemon_count counter
# HELP base:thread_daemon_count Displays the current number of
# live daemon threads.
base:thread_daemon_count 79
# TYPE base:classloader_total_unloaded_class_count counter
# HELP base:classloader_total_unloaded_class_count Displays the
# total number of classes unloaded since the Java virtual
# machine has started execution.
base:classloader_total_unloaded_class_count 3
# TYPE base:memory_max_heap_bytes gauge
# HELP base:memory_max_heap_bytes Displays the maximum amount of
# heap memory in bytes that can be used for memory management.
# This attribute displays -1 if the maximum heap memory size is
# undefined. This amount of memory is not guaranteed to be
# available for memory management if it is greater than the
# amount of committed memory. The Java virtual machine may fail
# to allocate memory even if the amount of used memory does not
# exceed this maximum size.
base:memory_max_heap_bytes 5.36870912E8
# TYPE base:cpu_process_cpu_load_percent gauge
# HELP base:cpu_process_cpu_load_percent Displays the
# 'recent cpu usage' for the Java Virtual Machine process.
base:cpu_process_cpu_load_percent 0.020974123871630043
# TYPE vendor:servlet_coffee_shop_com_sebastian_daschner
# _coffee_shop_jaxrs_configuration_response_time_total_seconds
# gauge
# HELP vendor:servlet_coffee_shop_com_sebastian_daschner
# _coffee_shop_jaxrs_configuration_response_time_total_seconds
# The total response time of this servlet since the start of
# the server.
vendor:servlet_coffee_shop_com_sebastian_daschner_coffee_shop
_jaxrs_configuration_response_time_total_seconds 4.5836283
# TYPE vendor:servlet_com_ibm_ws_microprofile_openapi_open_api
# _servlet_response_time_total_seconds gauge
# HELP vendor:servlet_com_ibm_ws_microprofile_openapi_open_api
# _servlet_response_time_total_seconds The total response time
# of this servlet since the start of the server.
vendor:servlet_com_ibm_ws_microprofile_openapi_open_api_servlet
_response_time_total_seconds 0.35594430000000005
# TYPE vendor:servlet_com_ibm_ws_microprofile_metrics_private
# _private_metrics_rest_proxy_servlet_response_time_total
# _seconds gauge
# HELP vendor:servlet_com_ibm_ws_microprofile_metrics_private
# _private_metrics_rest_proxy_servlet_response_time_total
# _seconds The total response time of this servlet since the
# start of the server.
vendor:servlet_com_ibm_ws_microprofile_metrics_private_private
_metrics_rest_proxy_servlet_response_time_total_seconds
0.34659330000000005
# TYPE vendor:session_default_host_metrics_active_sessions
# gauge
# HELP vendor:session_default_host_metrics_active_sessions
# The number of concurrently active sessions. (A session is
# active if the product is currently processing a request that
# uses that user session).
vendor:session_default_host_metrics_active_sessions 1
# TYPE vendor:servlet_coffee_shop_com_sebastian_daschner_coffee
# _shop_jaxrs_configuration_request_total counter
# HELP vendor:servlet_coffee_shop_com_sebastian_daschner_coffee
# _shop_jaxrs_configuration_request_total The number of visits
# to this servlet since the start of the server.
vendor:servlet_coffee_shop_com_sebastian_daschner_coffee_shop
_jaxrs_configuration_request_total 29
# TYPE vendor:threadpool_default_executor_size gauge
# HELP vendor:threadpool_default_executor_size The size of the
# thread pool.
vendor:threadpool_default_executor_size 12
# TYPE application:ft_com_sebastian_daschner_coffee_shop_control
# _barista_retrieve_brew_status_retry_calls_succeeded_retried
# _total counter
application:ft_com_sebastian_daschner_coffee_shop_control
_barista_retrieve_brew_status_retry_calls_succeeded_retried
_total 0
# TYPE application:ft_com_sebastian_daschner_coffee_shop_control
# _barista_retrieve_brew_status_invocations_failed_total counter
application:ft_com_sebastian_daschner_coffee_shop_control
_barista_retrieve_brew_status_invocations_failed_total 0
```

```
# TYPE application:ft_com_sebastian_daschner_coffee_shop_control
# _barista_retrieve_brew_status_retry_calls_failed_total counter
application:ft_com_sebastian_daschner_coffee_shop_control
_barista_retrieve_brew_status_retry_calls_failed_total 6
# TYPE application:ft_com_sebastian_daschner_coffee_shop_control
# _barista_retrieve_brew_status_invocations_total counter
application:ft_com_sebastian_daschner_coffee_shop_control
_barista_retrieve_brew_status_invocations_total 38
# TYPE application:ft_com_sebastian_daschner_coffee_shop
# _control_barista_retrieve_brew_status_retry_calls_succeeded_
# not_retried_total counter
application:ft_com_sebastian_daschner_coffee_shop_control
_barista_retrieve_brew_status_retry_calls_succeeded_not_retried
_total 32
# TYPE application:ft_com_sebastian_daschner_coffee_shop_control
# _barista_retrieve_brew_status_retry_retries_total counter
application:ft_com_sebastian_daschner_coffee_shop_control
_barista_retrieve_brew_status_retry_retries_total 18
# TYPE application:ft_com_sebastian_daschner_coffee_shop
# _control_barista_retrieve_brew_status_fallback_calls_total
# counter
application:ft_com_sebastian_daschner_coffee_shop
_control_barista_retrieve_brew_status_fallback_calls_total 6
```

MicroProfile provides APIs for a number of precanned metrics types, including timers for recording method timing, counters for total number of requests or max concurrent requests, and gauges to sample values. The following code shows how to get request timing information for the `orderCoffee` method. This could be useful to configure alerts in the event that response times deteriorate:

```
    @POST
    @Timed(
        name="orderCoffee.timer",
        displayName="Timings to Coffee Orders",
        description = "Time taken to place a new coffee order.")
    public Response orderCoffee(
    @Valid @NotNull CoffeeOrder order) {
        final CoffeeOrder storedOrder
        = coffeeShop.orderCoffee(order);
        return Response.created(buildUri(storedOrder)).build();
    }
```

Here is an example output for the resulting metric:

```
# HELP application:com_sebastian_daschner_coffee_shop_boundary_
# orders_resource_order_coffee_timer_seconds Time taken to
# place a new coffee order.
application:com_sebastian_daschner_coffee_shop_boundary_orders
_resource_order_coffee_timer_seconds_count 7
application:com_sebastian_daschner_coffee_shop_boundary_orders
```

```
_resource_order_coffee_timer_seconds{quantile="0.5"} 0.0318028
application:com_sebastian_daschner_coffee_shop_boundary_orders
_resource_order_coffee_timer_seconds{quantile="0.75"} 0.0413141
application:com_sebastian_daschner_coffee_shop_boundary_orders
_resource_order_coffee_timer_seconds{quantile="0.95"} 0.5347786
application:com_sebastian_daschner_coffee_shop_boundary_orders
_resource_order_coffee_timer_seconds{quantile="0.98"} 0.5347786
application:com_sebastian_daschner_coffee_shop_boundary_orders
_resource_order_coffee_timer_seconds{quantile="0.99"} 0.5347786
application:com_sebastian_daschner_coffee_shop_boundary_orders
_resource_order_coffee_timer_seconds{quantile="0.999"} 0.5347786
```

Dashboards and Alerts

It's one thing to generate metrics data from a microservice, but it's another to actually gather it, visualize it, and react. This is where monitoring dashboards come in. There are a number of tools available for this purpose; one common combination is Prometheus and Grafana. Prometheus is a powerful open source tool for gathering metrics data and alerting. Grafana provides great dashboard capabilities for analytics, monitoring, and alerting, and has first-class integration with Prometheus.

Creating a dashboard and alerts is simple. An instance of Prometheus can be pointed at the MicroProfile Metrics endpoint (e.g., *https://localhost:9443/metrics*). Prometheus periodically calls the endpoint to retrieve the metrics data. Any fields can be used to produce charts and set up alerts. If you'd like a Grafana dashboard, you can configure Grafana to populate a dashboard using data retrieved from Prometheus.

Figure 5-1 shows some of the metrics MicroProfile has generated for the `coffee-shop`. The chart on the left shows mean, max, and min times for the `orderCoffee` operation. These were generated from the earlier `@Timed` annotation example. The chart on the right shows two metrics generated by default by MicroProfile Fault Tolerance. These are showing successful requests over time and the number of calls to the Fallback operation over time. You can see that after a while we start to see failures and the subsequent use of the Fallback, due to instability in the `barista` service. Of course, if this were running in Kubernetes, the health endpoint would have told Kubernetes we were not ready to receive traffic.

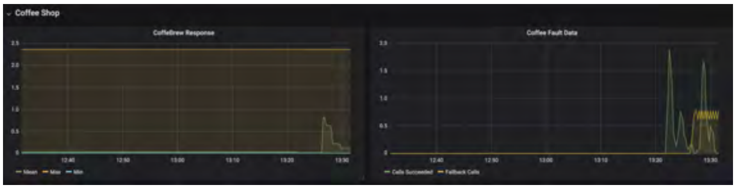


Figure 5-1. Grafana dashboard for the coffee-shop service MicroProfile application metrics

Figure 5-2 shows a dashboard for some of the default JVM metrics. These include heap size, system load, CPU load, and threads. These are all interesting and important metrics to track in order to detect when a service may encounter difficulties.

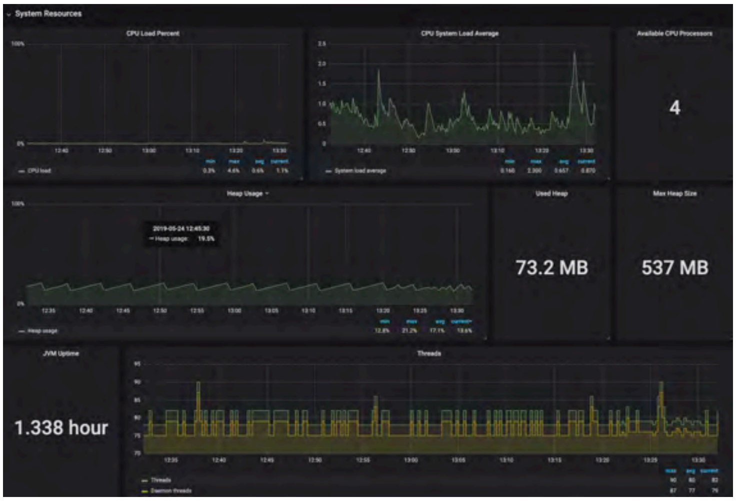


Figure 5-2. Grafana dashboard for default JVM metrics

Monitoring dashboards are essential, but the dashboards can contain so much data that it's difficult to spot when something is going wrong, and this is where alerts come in. Both Prometheus and Grafana provide the ability to set up alerts in order to be notified of aberrant situations. Figure 5-3 shows a dashboard for an alert we've set up to notify us when the Fallback calls exceed a threshold of 0.5. We can see that the alert has triggered. We can also set up Grafana to notify us via various channels, including email, Slack, webhook, and Kafka.

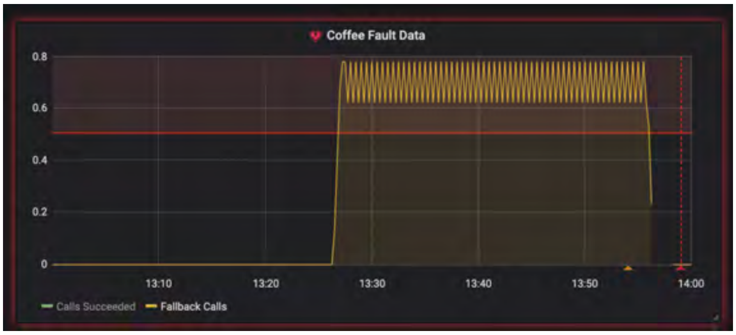


Figure 5-3. Grafana dashboard showing a triggered alert

Tracing Microservice Requests

You've deployed your tens or hundreds of microservices, and for a while, all seems fine. One day, you start encountering intermittent problems; some requests are timing out and some are slow. Health and Metrics alone may not give you the information necessary to diagnose the issue. At some point you're going to want to trace the requests through your interconnected microservices. You can think of this as the microservice equivalent of step-debugging through your code. You want to see the path your requests take and how long they take, in order to identify where they start to go wrong. This is where OpenTracing comes in.

OpenTracing is a language-neutral standard API for performing distributed tracing. MicroProfile OpenTracing enables distributed tracing of requests passing through your Java microservices. REST methods are automatically traced, which means you get a useful level of tracing without making any changes to your code.

MicroProfile OpenTracing requires an implementation of the OpenTracing Tracer API. This can be implemented over any number of tracing infrastructures, such as Apache Zipkin or Jaeger. In our example we're using Apache Zipkin.

Figure 5-4 shows an example Zipkin trace for placing a coffee order. You can see the request to `coffee-shop` `ordercoffee` takes approximately 20 ms and that the request to that service results in a call to the `barista` `updatecoffeebrew` operation.

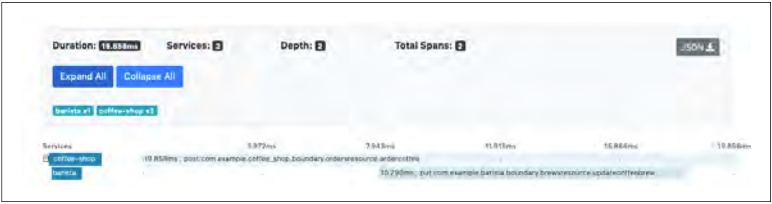
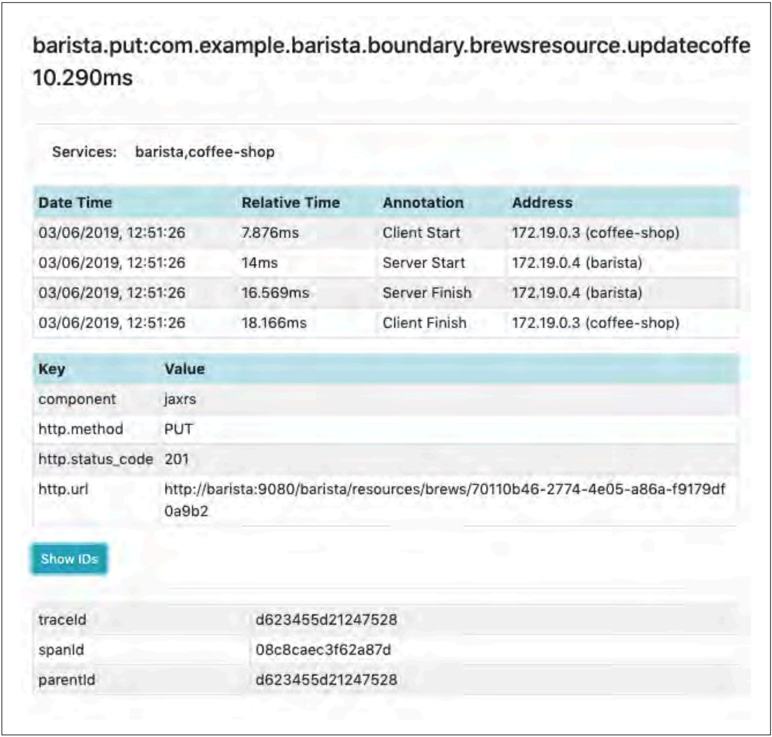


Figure 5-4. Zipkin trace for an ordercoffee request

It's possible to drill down on the *spans* to see more details. Figure 5-5 shows the details of the `updatecoffeebrew` request. It shows the timings for the requests, the HTTP request information (URL, methods, status), and the OpenTracing IDs for the request.



barista.put:com.example.barista.boundary.brewsresource.updatecoffe
10.290ms

Services: barista,coffee-shop

Date Time	Relative Time	Annotation	Address
03/06/2019, 12:51:26	7.876ms	Client Start	172.19.0.3 (coffee-shop)
03/06/2019, 12:51:26	14ms	Server Start	172.19.0.4 (barista)
03/06/2019, 12:51:26	16.569ms	Server Finish	172.19.0.4 (barista)
03/06/2019, 12:51:26	18.166ms	Client Finish	172.19.0.3 (coffee-shop)

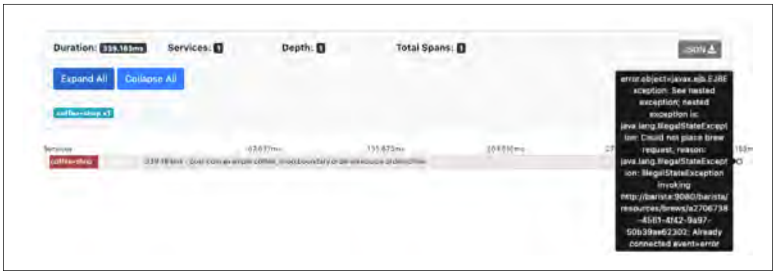
Key	Value
component	jaxrs
http.method	PUT
http.status_code	201
http.url	http://barista:9080/barista/resources/brews/70110b46-2774-4e05-a86a-f9179df0a9b2

Show IDs

traceId	d623455d21247528
spanId	08c8caec3f62a87d
parentId	d623455d21247528

Figure 5-5. Zipkin span details for an updatecoffeebrew request

We can also see failed requests, for example, when the `barista` service is unavailable. Figure 5-6 shows the trace for one such `ordercoffee` request.



error.object=javax.ejb.EJBException: See nested exception; nested exception is: java.lang.IllegalStateException: Could not place brew request, reason: java.lang.IllegalStateException: IllegalStateException invoking http://barista:9080/barista/resources/brews/a2706738-4501-4142-9a97-50b39ae62302: Already connected event=error

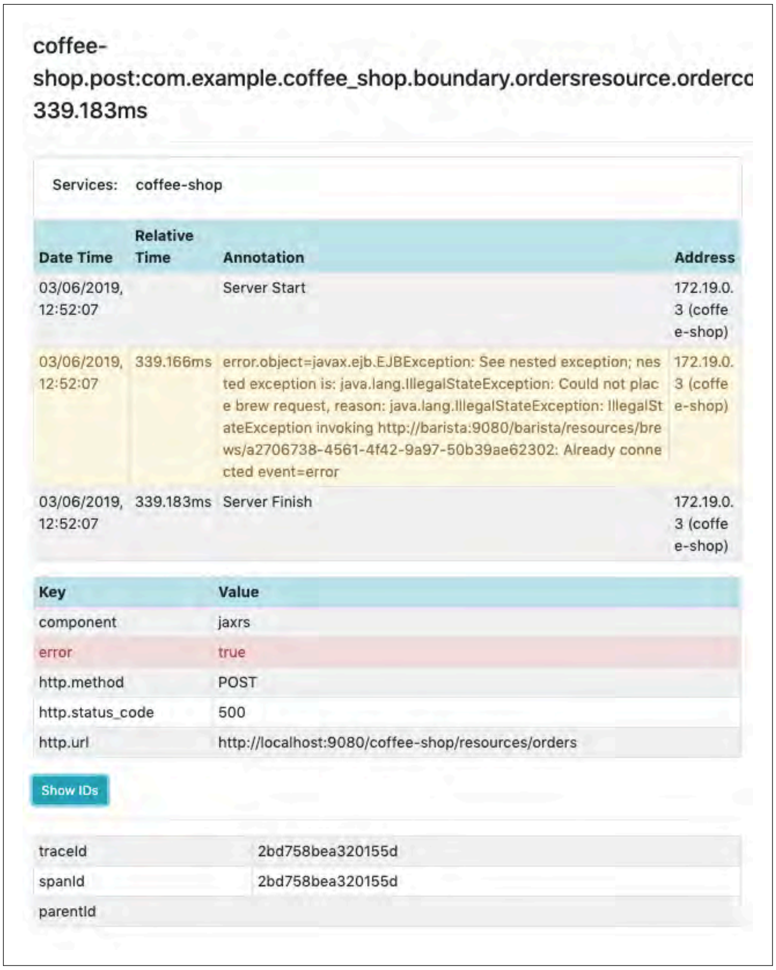
Figure 5-6. Zipkin trace for a failed ordercoffee request

Figure 5-7 shows the details of the failed request where we can see the exception and the HTTP methods and status.

Not all useful trace points correspond to REST API calls. For example, in our applications, an EJB Timer is used to periodically check the status of orders. If you want to control which methods are traced, MicroProfile provides the ability to explicitly trace spans using the `@Traced` annotation. In the absence of an explicit trace span, all we see in OpenTracing is the request coming into the `barista` service.

The following code shows the explicit trace span added to the `OrderProcessor.java`:

```
@Traced
public void processOrder(CoffeeOrder order) {
    OrderStatus status = barista.retrieveBrewStatus(order);
    order.setOrderStatus(status);
}
```



coffee-shop.post:com.example.coffee_shop.boundary.ordersresource.orderco
339.183ms

Services: coffee-shop

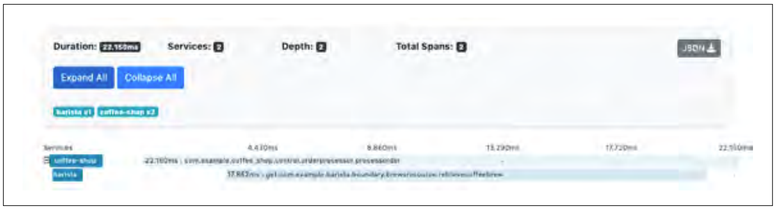
Date Time	Relative Time	Annotation	Address
03/06/2019, 12:52:07		Server Start	172.19.0.3 (coffee-shop)
03/06/2019, 12:52:07	339.166ms	error.object=javax.ejb.EJBException: See nested exception; nested exception is: java.lang.IllegalStateException: Could not place brew request, reason: java.lang.IllegalStateException: IllegalStateException invoking http://barista:9080/barista/resources/brews/a2706738-4561-4f42-9a97-50b39ae62302: Already connected event=error	172.19.0.3 (coffee-shop)
03/06/2019, 12:52:07	339.183ms	Server Finish	172.19.0.3 (coffee-shop)

Key	Value
component	jaxrs
error	true
http.method	POST
http.status_code	500
http.url	http://localhost:9080/coffee-shop/resources/orders

traceId	2bd758bea320155d
spanId	2bd758bea320155d
parentId	

Figure 5-7. Zipkin trace details for a failed ordercoffee request

Figure 5-8 shows the trace for the application-defined span. It shows how the `barista retrievecoffeebrew` operation is being called from the `coffee-shop` `processorder` operation.



22.150ms : com.example.coffee_shop.control.orderprocessor.processorder

17.863ms : get com.example.barista.boundary.brewsresource.retrievecoffeebrew

Figure 5-8. Zipkin trace for an application-defined span for processorder

Summary

Often when you are developing a new application, be it a traditional monolithic application or new cloud native application, observability is considered secondary to getting the functionality written. However, we've seen that when deploying microservices, the decomposition and distribution of application code makes it essential to consider observability from day one.

We've seen the three key aspects to observability:

- The readiness and liveness of your services through MicroProfile Health and the ability to integrate these with Kubernetes infrastructure
- The service runtime analysis through MicroProfile Metrics and the ability to monitor and alert through tools such as Prometheus and Grafana
- The ability to understand the flow of requests through a system of microservices through OpenTracing and tracer tools such as Apache Zipkin

OpenTelemetry

At the time of this writing, a new specification for distributed tracing is being developed, called OpenTelemetry. It is being created by the Cloud Native Computing Foundation (CNCF) and is the convergence of OpenTracing and a Google project called OpenCensus. Discussions are underway in the MicroProfile community to adopt OpenTelemetry, and the expectation is that this will become the preferred approach for distributed tracing. Given the heritage of OpenTelemetry and the fact that MicroProfile OpenTracing enables the most useful tracing by default, any migration is likely to be simple.

CHAPTER 6

Wrap-up and Conclusions

In the previous three chapters, we've seen how to get started with developing cloud native Java applications using open technologies. As industry understanding of cloud native has evolved and matured, a number of use cases have emerged requiring more sophisticated patterns, such as Reactive, Saga, and CQRS, as well as a general increase in asynchronous execution. These are beyond the scope of this introductory book, but we'll briefly discuss them here before we finish with our conclusions.

Asynchronous Execution and Reactive Extensions

Traditionally, the execution of business logic in enterprise applications happens synchronously; that is, a single thread usually executes the business logic from the start in the boundary until the business method returns. However, more recently, demand has grown for asynchronous and reactive execution, and subsequently a number of specifications have been enhanced with these features.

JAX-RS, for example, offers ways to asynchronously invoke HTTP calls in the JAX-RS client in both asynchronous and reactive ways, by natively supporting types such as `CompletionStage`. The same is true for asynchronous JAX-RS endpoint resources that handle the execution of the business logic in separate threads, decoupled from the default HTTP request thread pool.

CDI events are another example where the decoupling of business code can happen in an asynchronous way. Since Java EE 8, CDI events can be fired and handled asynchronously, without blocking the code that emits the event. Another technology that supports asynchronous communications is the WebSockets protocol, which is also natively supported in Enterprise Java.

The Eclipse MicroProfile community recently released two reactive specifications: *Reactive Streams Operators* and *Reactive Messaging*. Reactive Streams Operators defines reactive APIs and types. Reactive Messaging enhances CDI to add `@Incoming` and `@Outgoing` annotations to mark methods that process and produce messages reactively, which could, for example, be integrated with Kafka.

Over time we will likely see the integration of these APIs in various enterprise specifications, especially to streamline the use of multiple technologies with regard to reactive programming. The proper plumbing of stream, handling backpressure, and potential errors usually results in quite a lot of boilerplate code that could be reduced if different specifications support the same APIs.

Threading

The application server is traditionally responsible for defining and starting threads; in other words, the application code neither is supposed to manage its own threads nor start them. This is important, as information essential to the correct execution of the business logic (e.g., transaction and security contexts) is often associated with the threads. The container defines one or more thread pools that are used to execute business logic. Developers have multiple ways to configure these pools and to further define and restrict the executions—for example, by making use of the bulkheads pattern. We saw an example for this in Chapter 4, using MicroProfile Fault Tolerance.

Another aspect of asynchronous processing is the ability to have timed executions, (i.e., code that is executed in a scheduled or delayed manner). For this, Enterprise Java provides EJB timers, which are restricted to EJBs, or the managed scheduled executor services, which are part of the concurrency utilities.

Transactions and Sagas

In Chapter 3 we showed how to access database systems using local transactions. Enterprise Java traditionally supports two-phase commit (XA) transactions to coordinate transactional behavior between applications.

Distributed two-phase commit transactions can exhibit scalability challenges when used in modern microservice architectures, and so these often focus on eventual consistency with the ability to scale. Longer-running business transactions that span multiple systems are usually then implemented using the Saga pattern. This is more an architectural pattern than a specific implementation. We suggest you familiarize yourself with this class of approach and consider applying it once the business logic of your overall system requires multiple applications to take part in a long-running transaction.

Command Query Responsibility Separation

Another related eventual consistency pattern is *Command Query Responsibility Separation*, or CQRS. Here, the logic and data updating the system is separated from the logic and data required to support the various application views (frontends). This allows the data and logic for queries to be optimized for high throughput without compromising the performance of the paths for updating the backend. This pattern has proved popular in systems that have a large number of queries for a relatively small number of updates.

The separation of updates from queries leads to the need for eventually consistency rather than two-phase commit transactions. The storage of separate view data optimized for queries makes it a good fit for GraphQL APIs, where the views can submit queries for just the data they require. To support this, the MicroProfile community is in the process of defining a GraphQL specification.

Conclusions

Our primary motivation for writing this book was to enable you to be successful in cloud native development. As strong believers in using open technologies wherever possible, we discussed how being open not only helps avoid vendor lock-in, but also improves quality and longevity. Using the open criteria of open source, open standards, and open governance, as well as their interrelationships, we explained our rationale for selecting particular implementations and approaches for cloud native Java development.

Enterprise Java, with its open specifications and open source implementations, is already very well suited for the majority of today's applications, and we've shown you how to get started with using those capabilities.

To introduce the concepts for cloud native development, we walked you through a complete example of developing a cloud native application using only open source Java technologies based purely on open standards. We showed how many preexisting APIs can help you develop cloud native microservices and how new APIs are now available to handle important cloud native functionality. For example, we showed how to secure your services, build resiliency into your application code, and make your code *observable* with metrics, health checks, and request tracing.

This book has focused on the foundational technologies of cloud native Java application development. Cloud native goes far beyond the development of Java code, though, and we've intentionally only touched on other aspects such as containers, Kubernetes, Istio, and continuous integration/continuous delivery (CI/CD). These areas warrant books of their own. We've also only briefly touched on how cloud native is causing the industry to reimagine how applications and solutions are architected and how new architecture patterns (e.g., Sagas, CQRS) and technologies (gRPC, RSocket, GraphQL, Reactive, asynchronous execution) are emerging to support them. Again, these topics could take up many books all by themselves.

Looking forward, the future appears bright for open Enterprise Java technologies. Jakarta EE has just made its first release to create a foundation for future specifications. MicroProfile continues to grow, in terms of both the valuable technologies it provides and community and vendor implementations. With more Reactive APIs becoming available, and GraphQL, Long Running Actions (a framework for weaving together microservices to achieve eventual consistency), and other specifications in the pipeline, MicroProfile will soon also have the foundational capabilities for building emerging cloud native architecture patterns.

Finally, innovation isn't happening just in the APIs and architectures we use, but also in the developer tools and how they're used. We're seeing open tools emerging that are designed specifically for cloud native development (e.g., Eclipse Codewind), including in hosted environments (e.g., Eclipse Che). We think this is going to be a very exciting area of innovation over the coming years, with the potential for great improvement in the cloud native developer experience. For the latest content about developing modern applications with the open Java ecosystem, please visit *ibm.biz/oreilly-java-tech*.

We hope that you've found this book useful and that it has inspired and enabled you to give the technologies a try. We wish you every success in your open cloud native journey.

About the Authors

Graham Charters is an IBM senior technical staff member and WebSphere Applications Server developer advocacy lead based at IBM's R&D Laboratory in Hursley, UK. He has a keen interest in emerging technologies and practices and in particular programming models. His past exploits include establishing and contributing to open source projects at PHP and Apache, and participating in, and leading industry standards at OASIS and the OSGi Alliance.

Sebastian Daschner is a lead Java developer advocate for IBM. His role is to share knowledge and educate developers about Java, enterprise software, and IT in general. He enjoys speaking at conferences; writing articles and blog posts; and producing videos, newsletters, and other content.

Pratik Patel is a lead developer advocate at IBM. He co-wrote the first book on Enterprise Java in 1996, *Java Database Programming with JDBC* (Coriolis Group). He has also spoken at various conferences and participates in several local tech groups and startup groups. He hacks Java, iOS, Android, HTML5, CSS3, JavaScript, Clojure, Rails, and—well, everything except Perl.

Steve Poole is a long-time Java developer, leader, and evangelist. He is a DevOps practitioner (whatever that means). He has been working on IBM Java SDKs and JVMs since Java was less than 1. He is a seasoned speaker and regular presenter at international conferences on technical and software engineering topics.