Kubernetes Native Microservices

with Quarkus, and MicroProfile

John Clingan
Ken Finnigan

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Kubernetes Native Microservices with Quarkus and Microprofile

Version 3

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Kubernetes Native Microservices with Quarkus and MicroProfile*. To get the most benefit from this book, you'll want to have enterprise Java skills, like Jakarta EE or Spring.

We have been working with enterprise Java for over 15 years and evolving our skills to keep pace with the industry shift from monoliths to microservices, from Java EE to MicroProfile, and from virtual machines to Kubernetes. We have helped create and evolve MicroProfile as a collection of microservices specifications. We have also helped create and evolve Quarkus as a MicroProfile implementation. More specifically, Quarkus is a MicroProfile implementation that targets containerized applications on Kubernetes specifically.

We often speak at industry conferences, Java Meetups, and participate in various online forums regarding Java, MicroProfile, Quarkus, and Kubernetes. There is a tremendous amount of interest where these technologies all intersect, but a lack of content that discusses how to best use these technologies together and modernize existing skills. Our goal is to demonstrate how to develop Java microservices with Quarkus and MicroProfile that specifically target Kubernetes as a deployment environment.

There are step-by-step instructions that make heavy use of Quarkus Live Coding to make code changes, so you experience immediate feedback. You can follow each step, make a code change, and immediately try it out. Recompiling, repackaging, and restarting the application with Quarkus is rare and primarily occurs during deployment. Live coding makes developing microservices extremely productive. It also makes learning MicroProfile and Quarkus with Kubernetes quite fun!

We encourage feedback at liveBook's Discussion Forum and ongoing dialogue to make the book more clear and the experience enjoyable. See you online!

—John Clingan and Ken Finnigan

brief contents

1 Introduction to Quarkus, Eclipse MicroProfile, and Kubernetes

This chapter covers:

- Microservices Overview
- Overview and history of Eclipse MicroProfile
- Quarkus introduction
- Kubernetes introduction

Entire books are available on Quarkus, Microservices, MicroProfile, Spring, and Kubernetes. However, they tend to focus on each specific topic. This book covers how to combine these topics into an effective and integrated development and deployment stack. Kubernetes Native Microservices are microservices that utilize and integrate with Kubernetes features naturally and efficiently. The result is a productive developer experience that is consistent with the expectations of Kubernetes platform administrators.

This chapter begins by defining microservices, and how and why they have evolved over the last decade as a popular enterprise software architecture. We'll then provide a brief history and overview of MicroProfile and its growth into a significant collection of microservices-related specifications. With a baseline understanding of microservices and MicroProfile, we will be introducing Quarkus as a Java runtime that supports these technologies. Lastly, we'll introduce some core Kubernetes concepts and why they make Kubernetes an ideal microservice deployment platform.

NOTE	A "runtime" is an execution environment that includes a collection of packaged frameworks that collectively support a developer's application logic. Java EE (now Jakarta EE [1]) application servers, Spring Boot, and Quarkus are all examples of Java runtimes since each is a Java execution environment that includes Java frameworks that support application logic.

1.1 What is a microservice?

Conducting an internet search will result in hundreds of microservices definitions. There is no industry consensus on a single definition, but there are some common and well-understood principles. We are using a definition that aligns with those principles, but that has a particular emphasis on one principle - *isolation*. As defined in Enterprise Java Microservices [2], *a microservice consists of a single deployment executing within a single process, isolated from other deployments and processes, that supports the fulfillment of a specific piece of business functionality*.

We are going to put a bit more emphasis on the runtime aspect of isolation than most other writings. With Kubernetes as the target deployment platform, there is an opportunity for optimizing code and the Java runtime itself. While a microservice is isolated business functionality, it nearly always interacts with other microservices. That is the basis of many code examples for this book. There are a couple of useful points to make when breaking down the selected definition.

First, a microservice implements a specific piece of business functionality. This is known as a a *bounded context* [3], which is a logical separation of multiple business problem domains within an enterprise. By logically breaking down a business domain into multiple bounded contexts, each bounded context more accurately represents its specific view of the business domain and becomes easier to model.

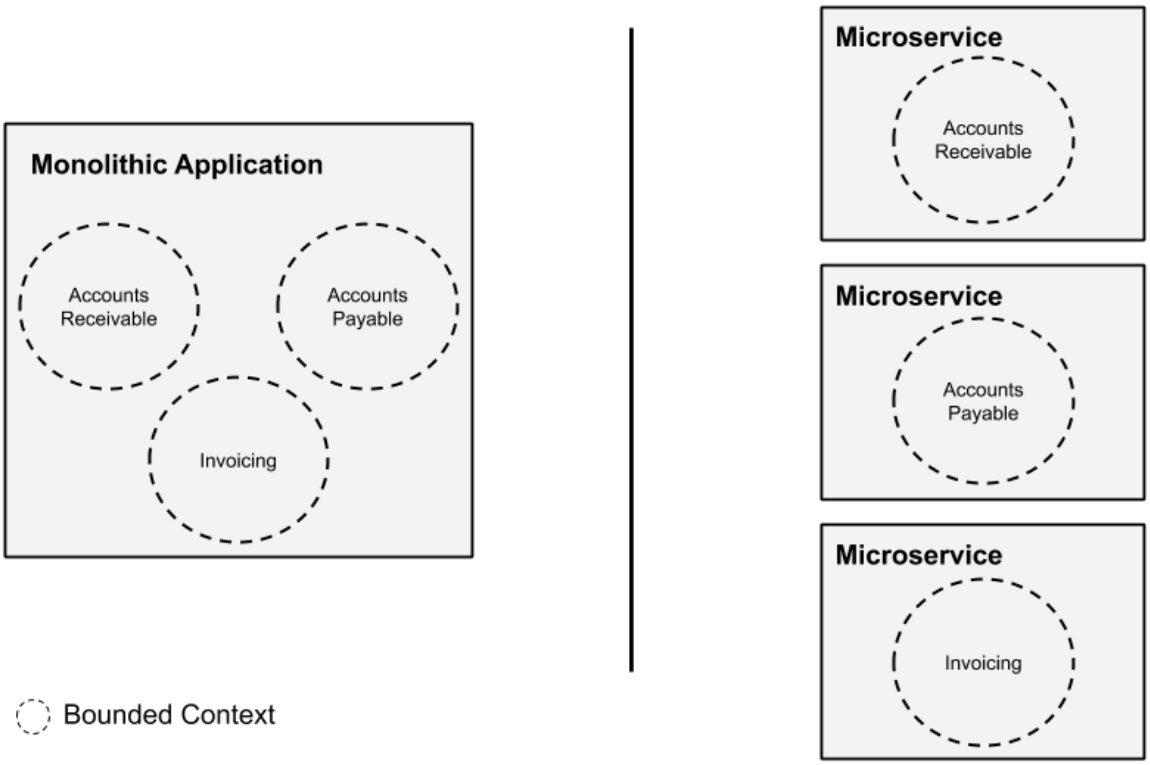


Figure 1.1 Bounded Context - monolith vs microservices

As represented in Figure 1.1, the set of bounded contexts for a small business accounting application may include accounts receivable, accounts payable, and invoicing. A traditional monolithic application would implement all three bounded contexts. Multiple bounded contexts within in a single monolith can result in "spaghetti code" as a result of unnecessary inter-dependencies and unplanned intermixing of contexts. In a microservices architecture, each of these capabilities is modeled individually as a bounded context and implemented as a microservice that addresses each specific bounded context.

Next, a microservice executes within a single isolated process. While this is not a concrete requirement, it has become a preferred architectural approach. There are some practical reasons behind this based on over a decade of experience of deploying applications to Java EE application servers and servlet containers like Apache Tomcat. We'll refer to these synonymously as "application servers".

From a technical perspective, application servers can host multiple microservices. However, this deployment model has fallen out of favor for many reasons:

- **Resource management.** One microservice can starve other microservices of resources. The Java Virtual Machine (JVM) does not have built-in resource management to limit resource consumption by different applications within the same JVM instance.
- **Patching / upgrading.** Patching or upgrading an application server negatively impacts the availability of all hosted microservices simultaneously.

- **Versioning.** Each microservice development team may want to evolve at a different pace, causing an application server versioning requirements mismatch. Some may want to leverage new features of the latest version while others may prefer to avoid introducing risk since the current version is stable in production.
- **Stability.** One poorly written microservice can cause stability issues for the entire application server, impacting the availability of the remaining stable applications.
- **Control.** Developers rightfully cede control of shared infrastructure, like application servers, to a separate DevOps team. This limits developer options like JDK version, tuning for a specific microservice's optimal performance, application server version, and more.

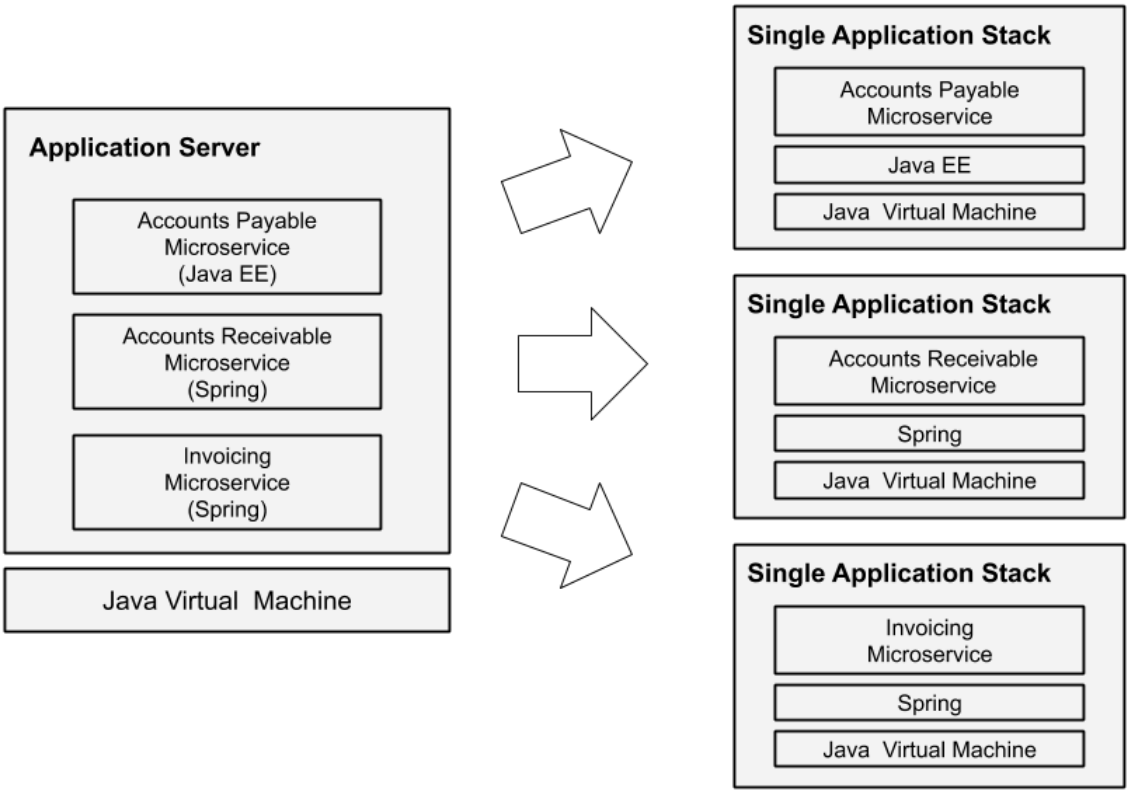


Figure 1.2 Application servers vs single application stacks

Figure 1.2 shows that these issues have driven the industry towards adopting a *single application stack* for microservices, which is a one-to-one mapping between a microservice application and its runtime. This began nearly a decade ago by deploying a single microservice per application server, and shortly thereafter evolved into specialized microservice runtimes like Dropwizard, Spring Boot, and more recently Quarkus to improve the developer and administrator experience. We'll refer to these single application stacks as Java microservice runtimes, and cover this concept in more detail later in the chapter. Note that with microservices it is easier split out and optimize the stack for a particular runtime like Java EE or Spring. An added benefit of single application stack is that it can also be implemented in non-Java technologies like Node.js or Golang, although these are out of scope of this discussion.

1.1.1 The rise of microservices

Early microservices tended to directly communicate with one another. This approach is sometimes referred to as "smart services with dumb pipes". A possible downside to this approach is the encoding within each service of the knowledge of *what happens next*. Tightly coupling this knowledge into the code makes it inflexible to dynamic change, and a potentially tedious task for engineers if it experiences regular change. If the knowledge around *what happens next* changes frequently, consider implementing the functionality using a business rules engine, or utilizing events as part of an Event-Driven architecture. We will be utilizing both approaches in the example application.

With the popularity of Netflix, with their thousands of microservices, and other unicorns like them, the popularity and enthrall of microservices exploded. Microservices became *the thing* everyone wanted to develop for their next project.

The rise of microservices lead to perceived benefits in delivery speed, better utilization of resources with smaller teams, and shifting of operational concerns to the team developing the code. This last item we now refer to as *DevOps*.

However, microservices was not the panacea that everyone hoped it would be. The benefits we mentioned above don't come automatically by virtue of developing a microservice. It takes organizational and people change for all the benefits to be achieved. It's often forgotten, not all implementation patterns, such as microservices, are right for every organization, team, or even group of developers. Sometimes it's necessary to acknowledge that microservices is not appropriate for a given situation, but will be perfect for another. As with everything in software engineering, do your homework and don't blindly adopt a pattern because *it's cool*. That is the path to disaster!

1.1.2 Microservices architecture

So what is a microservice architecture, and what does it look like?

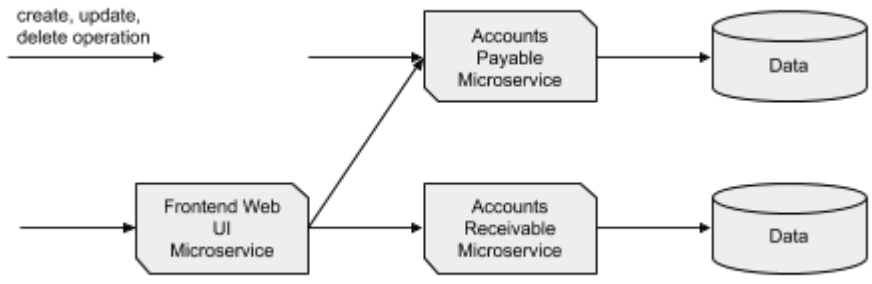


Figure 1.3 Microservices architecture - collaborating microservices

Figure 1.3 is just one example of many possible architectures that are applicable when developing microservices. There can be microservices calling databases, microservices calling other microservices, microservices communicating with external services, microservices passing messages, or events, to brokers and streaming services. For example, to add a user experience, a Frontend Web UI microservice has been added whose purpose is to add, update, delete, and view relevant information in the Accounts Payable and Accounts Receivable microservices. While the freedom to architect microservices in any desired manner offers limitless options, the downside is it offers limitless options. With limitless options, it quickly becomes difficult to chart a path towards a meaningful microservices architecture. The key is to start with the smallest possible piece of functionality and begin building out from there. When it's the first time a team is developing microservice architectures, it's even more critical to not create a **big picture** up front. Taking the time to create that big picture without previous experience of microservices architecture design will expend time when it's likely the final architecture will actually be very different. During the process of gaining experience with microservices, the architecture will shift over time towards a more appropriate one.

> **NOTE** **An alternative approach is to develop a monolith of loosely coupled components that can then be extracted out into microservices, strangling the monolith, if deemed necessary down the road.**

In short, a microservices architecture can be almost anything that incorporates the coordination of services into a cohesive application that meets business requirements.

Granted, with a limitless set of options for what can constitute a microservices architecture, architects and developers can benefit tremendously from having patterns and recommendations for how they can be designed.

This is where microservices specifications come to the aid of enterprise Java developers.

1.1.3 The need for microservices specifications

Java EE has been the standard-bearer for enterprise Java specifications for roughly 20 years. However, Java EE has been traditionally focused on 3-tier monolithic architecture with a steady, measured evolution and a strong focus on backward compatibility. Java EE stopped evolving between 2014 and 2017 just as the industry began to heavily adopt microservices.

During that pause, the Java EE community began to experiment and deliver early microservices APIs. There was an increasing risk of API fragmentation across Java runtimes that had been known for application portability. In addition, there was a risk of losing reusable skills. For example, Java EE APIs like JPA and JAX-RS are used with non-Java EE platforms like Spring

and Dropwizard, making it easier to switch to a Java runtime that better meets businesss criteria. To avoid fragmentation and loss of reusable skills, the community made the decision to collaborate on microservice specifications.

1.2 Eclipse MicroProfile

To avoid Java API fragmentation and to leverage the collective vendor and community knowledge and resources, MicroProfile was founded in June 2016 by IBM, London Java Community (LJC), Payara, Red Hat, and Tomitribe. The tagline "Optimizing Enterprise Java for a Microservices Architecture" recognizes that Java offers a solid foundation for building microservices. MicroProfile extends that foundation through the creation and evolution of Java API specifications for well-understood microservices patterns and cloud-related standards. These common APIs can be used by multiple frameworks and implementations, or runtimes.

Today, twelve specifications have been developed by the MicroProfile community, listed in Table 1 and Table 2. Most of the specifications in Table 1 will be covered in future chapters.

Table 1.1 MicroProfile Platform Specifications

Specification	Description
Config	Externalize application configuration
Fault Tolerance	Defines multiple strategies to improve application robustness
Health	Express application health to the underlying platform
JWT RBAC	Secure RESTful Endpoints
Metrics	Expose platform and application metrics
Open API	Java APIs for the OpenAPI specification that document RESTful endpoints
Open Tracing	Defines behaviors and an API for accessing an OpenTracing compliant Tracer object
REST Client	Type-safe invocation of REST endpoints

Table 1.2 MicroProfile Standalone Specifications

Specification	Description
Context Propagation	Propagates contexts across units of work that are thread-agnostic.
GraphQL	Java API for the GraphQL query language
Reactive Streams Operators *	Allows two different libraries that provide asynchronous streaming to be able to stream data to and from each other.
Reactive Streams Messaging *	Provides asynchronous messaging support based on Reactive Streams

> **NOTE** MicroProfile has grown to include 12 specifications. There is a concern that including too many specifications in the overall platform is a barrier to entry for new implementations. For this reason, any new specification is outside the existing platform and referred to as a "standalone" specification. The MicroProfile community plans to review how to organize specifications in the future.

1.2.1 History of MicroProfile

MicroProfile is quite unique in the industry. Whereas specification organizations tend to evolve in an intentionally slow and measured manner, MicroProfile delivers industry specifications that evolve rapidly. In four short years, MicroProfile has released 12 specifications with nearly all having multiple updates and some having major updates. This delivers new features that work across multiple implementations in the hands of developers up to three times per year. In other words, MicroProfile keeps pace with changes in the industry.

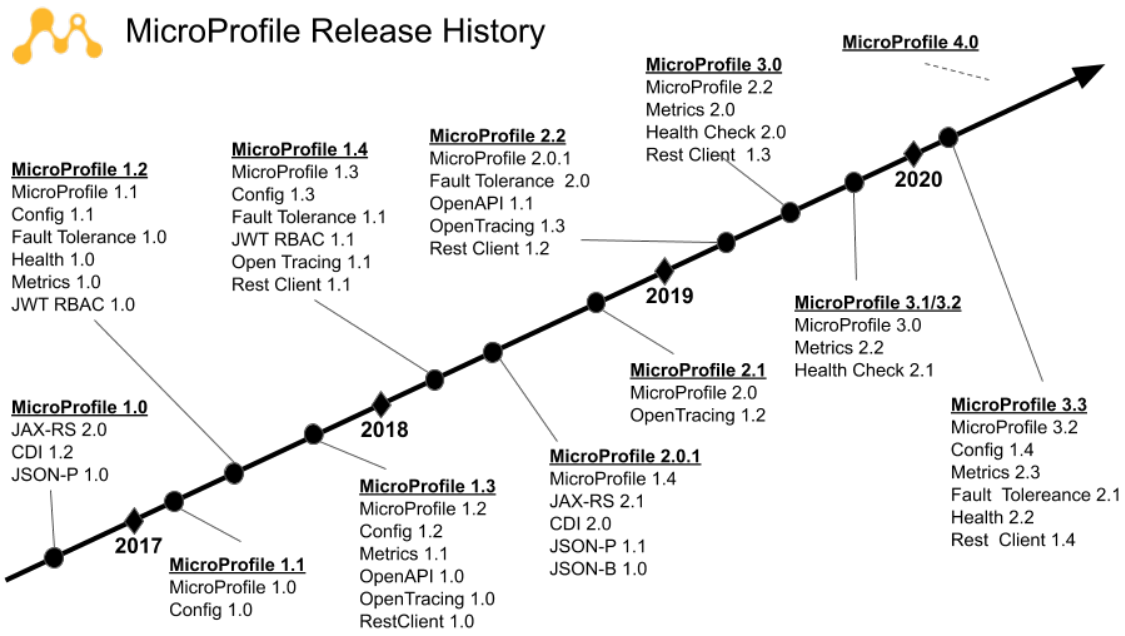


Figure 1.4 MicroProfile releases

Figure 1.4 puts this in perspective. MicroProfile 1.0 was released in September 2016, adopting three Java EE specifications to define its core programming model, specifically Java API for

RESTful Services (JAX-RS) 2.0, Contexts and Dependency Injection (CDI) 1.2, and JSON Processing (JSON-P) 1.0. The MicroProfile founders looked to expand the vendor and community members, while also beginning specification development. The community immediately recognized that hosting MicroProfile in a vendor-neutral foundation would facilitate these goals. After considering the options, the Eclipse Foundation became the home of Eclipse MicroProfile in December 2016. Over the next four years, MicroProfile released 3 major releases and nine minor releases that adopted JSON-B from Java EE and defined twelve "homegrown" specifications outlined in Table 1.1 and Table 1.2.

1.2.2 MicroProfile community core principles

As an Eclipse Foundation working group, MicroProfile follows some of the Foundation's core tenets like open source, vendor neutrality, and community engagement and collaboration. The MicroProfile Working Group draft charter [4] extends those tenets with the following additional principles:

- **Limited Processes.** MicroProfile uses the Eclipse Development Process and the Eclipse Foundation Specification Process. Any additional processes specific to MicroProfile are only created when necessary.
- **Experiment and innovate.** MicroProfile as a community provides an industry proving ground to incubate and experiment with well-established problems needing cross-Java-Runtime APIs, gather user feedback, adapt and iterate at a fast pace.
- **No backward compatibility guarantee.** Major versions of a specification developed within MicroProfile may break backward compatibility.
- **Implementation first.** MicroProfile specifications are released only after an implementation has been created and both the specification and implementation have had sufficient time for community review.
- **Encourage brand adoption.** Define guidelines that would allow usage of the MicroProfile brand without charge.
- **Openness.** Transparency, inclusiveness and eliminating barriers to participate are highly-valued principles. Public meetings and lists are preferred. Lists are favored for key decisions. Specifications have been managed in a way that provides open access to all MicroProfile committers.
- **Low barrier to entry.** It is MicroProfile's intent to operate a low-cost Working Group. Budget will be evaluated annually and as membership changes for opportunities to maintain low fees and costs.

These tenets make MicroProfile somewhat different than most organizations that create specifications. For example, MicroProfile considers itself an agile project and is willing to break backward compatibility. This willingness results from a rapid-moving specification project, and any breaking changes are well thought out with strong justification and as narrow a scope as possible.

1.3 Quarkus

Quarkus is a Java microservices runtime. Does the industry really benefit from yet another Java microservice runtime? Yes! To understand why, let's take a look at some inherent problems with existing runtimes.

Most Java microservice runtimes utilize existing frameworks that were developed for shared environments like application servers, where each application had its own set of requirements. These frameworks are mature and still very relevant, but haven't fundamentally changed since the mid 2000's and continue to heavily rely on dynamic runtime logic using Java reflection. More specifically, no substantive optimizations have been made to these frameworks for Java microservice runtimes. The result is high RAM utilization and slower startup time due to a large amount of work at application startup.

Another pain point is that developer productivity often suffers with Java microservice runtimes. Every time a developer makes a change they have to save the file, rebuild the application, restart the application, and refresh the browser. This can take 10s of seconds, significantly impacting the productivity of a developer. Multiply that by the developers in a team over time, and it quickly equates to a large sunk resource cost for an enterprise.

Developers and DevOps teams began to feel the pain of developing and deploying Java microservices, and have been increasingly considering alternatives like Node.js and Golang due to their reduced RAM requirements and fast startup time. These alternatives can also achieve a 5x to 10x deployment density on the same hardware, significantly reducing cost.

Quarkus is a Java runtime that takes a fresh look at the needs of the modern Java microservice developer. It is designed to be as productive as Node.js for developers, and consume as few resources as Golang. To many developers, Quarkus feels both new and familiar at the same time. It includes a lot of new, impactful features while supporting the APIs that developers are already familiar with.

When developing microservices, runtimes often do not consider the target environment. Most runtimes are deployment environment agnostic to be broadly relevant. While Quarkus is used in a wide variety of deployment environments, it has specific enhancements and optimizations for Linux containers and Kubernetes. For this reason, Quarkus is referred to as *Kubernetes Native Java*.

1.3.1 Developer joy

Developer joy is a top priority for Quarkus. Developers are rightfully enamored with the productivity of dynamic language runtimes like Node.js, and Quarkus is driving to deliver that experience even though Java is a "static" (pre-compiled) language.

The top developer joy feature is live coding, where code changes are detected, recompiled, and reloaded without having to restart the JVM. Live coding is enabled when Quarkus is started in developer mode using `mvn quarkus:dev`. Specifically, Quarkus checks for code changes when it receives external events like HTTP requests or Kafka messages. The developer simply makes code changes, saves the file, and refreshes the browser for near-instant updates. Live coding even works with pom.xml changes. The Quarkus maven plugin will detect pom.xml changes and restart the JVM. It is not uncommon for Quarkus developers to start Quarkus in developer mode, and then minimize the terminal window, never having to restart the JVM during a coding session.

NOTE	**Quarkus supports both Maven and Gradle. This book will reference Maven commands and features, but equivalent capabilities are available with Gradle.**

Another developer joy feature is a unified configuration. Quarkus supports APIs and concepts from multiple ecosystems like Java EE, Eclipse Vert.x, and even Spring. Each of these ecosystems defines its own collection of configuration files. Quarkus unifies configuration so that all configuration options can be specified in a single `application.properties` configuration file. Quarkus supports MicroProfile Config, an API specification that includes support for multiple configuration sources. The MicroProfile Config chapter will discuss this in more detail.

Future chapters will discuss additional developer joy features as they are utilized. For example, *Chapter 4 - Database Access with Panache* discusses how to replace boilerplate database access code with a simplified data access API layered on the Java Persistence API (JPA) and Hibernate.

1.3.2 MicroProfile support

Quarkus is a Java runtime with a focus on developing microservices to run on Kubernetes. MicroProfile is a collection of Java specifications for developing microservices. Therefore, it is a natural fit for Quarkus to implement MicroProfile specifications to facilitate microservices development. Also, developers can re-host their existing MicroProfile applications on Quarkus for improved productivity and runtime efficiency. Quarkus is continually evolving to stay current with MicroProfile releases. At the time of this writing, Quarkus supports MicroProfile 3.3 as described in *Section 1.2 - Eclipse MicroProfile* and all standalone MicroProfile specifications. Besides CDI and MicroProfile Config, which are included in the Quarkus core, each MicroProfile specification is available as a Quarkus Extensions that can be included using Maven dependencies.

1.3.3 Runtime efficiency

Quarkus has become well known for its fast startup time and low memory usage, earning its "Supersonic, Subatomic Java" marketing tagline. Quarkus can run applications on the JVM. It can also compile the application to a native binary using GraalVM [5] Native Image. Table 1.3 compares Quarkus startup times with a traditional cloud-native Java stack, packaged and run as uber-jars.

Table 1.3 Startup + time to first HTTP response (seconds)

	Traditional Cloud Native Java Stack	Quarkus JVM	Quarkus Native
REST Application	4.3	.943	.016
CRUD Application	9.5	2.03	.042

The REST application replies to HTTP REST requests, and the CRUD application creates, updates, and deletes data in a database. This table demonstrates that Quarkus can start significantly faster than traditional Java runtimes. Next, let's look at the memory usage.

Table 1.4 Memory usage (Megabytes)

	Traditional Cloud Native Java Stack	Quarkus JVM	Quarkus Native
REST Application	136	73	12
CRUD Application	209	145	28

As shown in Table 1.4, Quarkus achieves compelling RAM and startup time improvements over traditional cloud-native Java runtimes. It achieves this by re-thinking the problem. Traditional cloud-native Java runtimes do a lot of work when they boot. Each time an application boots it scans configuration files, scans for annotations, instantiates and binds annotations to build an internal meta-model before executing application logic.

Quarkus, on the other hand, executes these steps during compilation and records the results as bytecode that executes at application startup. In other words, Quarkus executes application logic immediately upon startup. The result is rapid startup time and lower memory utilization.

1.4 Kubernetes

During the 2000's, virtual machines were the go-to platform for hosting Java application servers, which in turn often hosted dozens of monolithic applications. This was sufficient until the adoption of microservices within the enterprise, which caused an explosion in the number of application instances to hundreds, thousands, up to tens of thousands for large organizations. Virtual machines utilize too many compute and management resources at this scale. For example, a virtual machine contains an entire operating system image, consuming more RAM and CPU resources than needed by the microservice, and must be tuned, patched, and upgraded. This was typically managed by a team of administrators, leaving little flexibility to developers.

These limitations led in part to the popularity of Linux containers, in part due to its balanced approach to virtualization. Containers, like virtual machine images, include the capability of packaging an entire application stack in container images. These images can be run on any number of hosts and instantiated any number of times to achieve horizontal scalability for service reliability and performance. Linux containers are significantly more efficient than virtual machines because all containers running on the same host share the same Linux operating system kernel.

While containers offer efficient execution of microservices, managing hundreds to thousands of container instances and ensuring proper distribution across container hosts to ensure scalability and availability is difficult without help from an orchestration platform for containers. Kubernetes has become that platform, and is available from popular cloud providers and can also be installed locally within a datacenter.

This also redraws the boundary between developers and those that manage the Kubernetes clusters. Developers are no longer required to utilize the Java version, application server version, or even the same runtime that had been dictated to them in the past. Developers now have the freedom to choose their own stack, as long as it can be containerized.

1.4.1 Introduction to Kubernetes

Kubernetes is a container orchestration platform that offers automated container deployment, scaling, and management. It originated at Google in various forms as a means to run internal workloads, was publicly announced in mid-2014, and version 1.0 was released mid-2015. Coinciding with the 1.0 release, Google worked with the Linux Foundation to form the Cloud Native Computing Foundation (CNCF) with Kubernetes being its first project. Today, Kubernetes has over 100 contributing organizations and well over 500 individual contributors. With such large, varied, and active contributions, Kubernetes has become the de-facto standard enterprise container orchestration platform. It is quite broad in functionality, so we'll focus on the underlying Kubernetes features and concepts that are most relevant when developing and deploying a microservice.

Kubernetes was not available before 2015, so early microservice deployments had to not only manage microservices, but they also had to manage infrastructure services to support a microservices infrastructure. Kubernetes offers some of these infrastructure services out of the box, making Kubernetes a compelling microservices platform. Although we are focusing on Java microservices, the following built-in features are runtime agnostic:

- **Service Discovery.** Services deployed to Kubernetes are given a stable DNS name and IP address. For a microservice to consume another microservice, it only has to locate the service by a DNS name. Unlike early microservice deployments, there is no need for a third party service registry to act as an intermediary to locate a service.
- **Horizontal Scaling.** Applications can be scaled out and scaled in manually or

automatically based on metrics like CPU usage.

- **Load Balancing.** Kubernetes will load balance across application instances. This removes the need for client-side load balancing that became popular during the early days of microservices.
- **Self Healing.** Kubernetes will restart failing containers and direct traffic away from containers that are temporarily unable to serve traffic.
- **Configuration management.** Kubernetes can store and manage microservice configuration. Configurations can change without updating the application. This removes the need for external configuration services used by early microservice deployments.

The Kubernetes architecture enables these features and is outlined below, and followed by a summary of each architectural component.

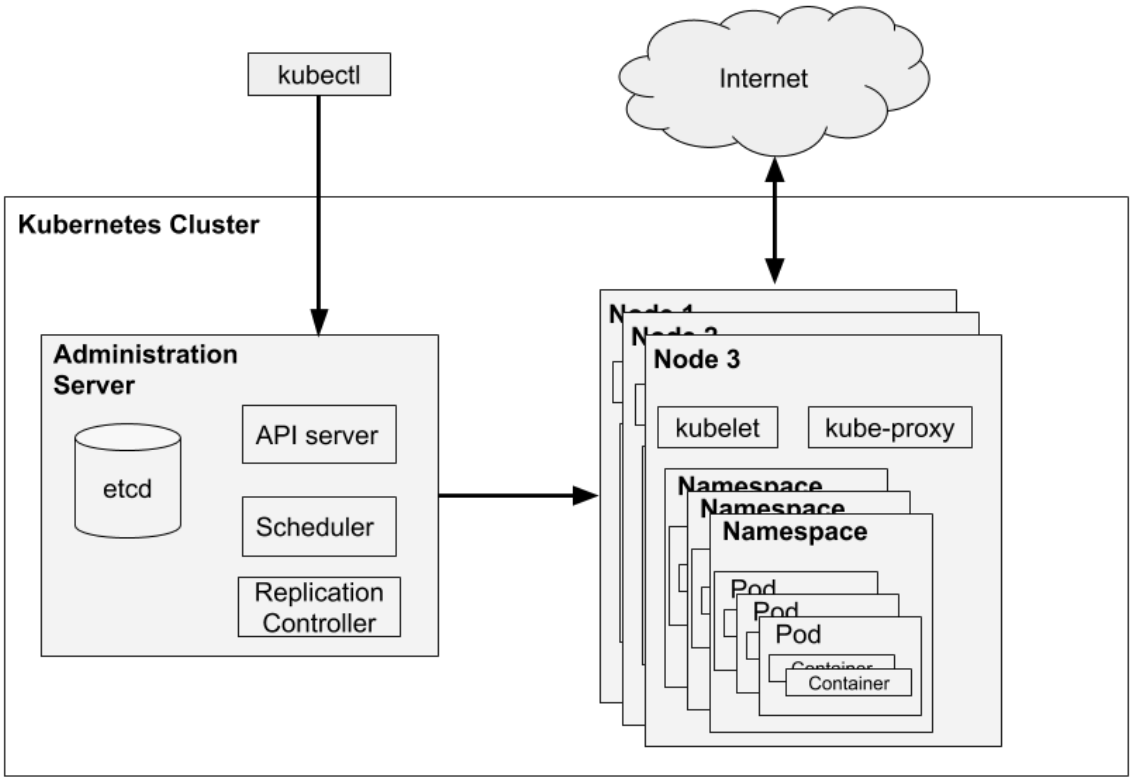


Figure 1.5 Kubernetes Architecture

- **Cluster.** A Kubernetes cluster abstracts hardware or virtual servers (Nodes) and presents them as a pool of resources. A cluster consists of one or more Administration ("Master") Server used to manage the cluster and any number of worker Nodes used to run workloads (pods). The administration server exposes an API server used by administration tools, like `kubectl`, to interact with the cluster. When a workload (Pod) is deployed to the cluster, the scheduler schedules the Pod to execute on a Node within the cluster.
- **Namespace.** A means to divide cluster resources between projects or teams. A Namespace can span multiple nodes in a cluster, so the diagram is a bit over-simplified for readability. Names defined within a namespace must be unique, but can be re-used across Namespaces.
- **Pod.** A Pod is one or more containers that share the same storage volumes, network,

Namespace, and lifecycle. Pods are atomic units, so deploying a pod deploys all containers within that pod to the same Node. For example, a microservice may use a local out-of-process cache service. It may make sense to place the microservice and the caching service in the same pod if there is a tight coupling. This ensures they are deployed to the same Node and have the same lifecycle. The pods in the exercises consist of one container per pod, so it will "feel" as if a pod is the same thing as a container, but that is not the case. A pod is ephemeral, meaning a pod's state is not maintained between destruction and any subsequent creation.

- **Replication controller.** Ensures the number of running pods matches the specified number of replicas. Specifying more than one replica improves availability and service throughput. If a pod is killed, then the replication controller will instantiate a new one to replace it. A Replication Controller can also conduct a rolling upgrade when a new container image version is specified.
- **Deployment** A deployment is a higher-level abstraction that describes thes state of a deployed application. For example, a deployment can specify the container image to be deployed, the number of replicas for that container image, health check probes used to check pod health, and more.
- **Service.** A stable endpoint used to access a group of like pods, and brings stability to a highly dynamic environment.

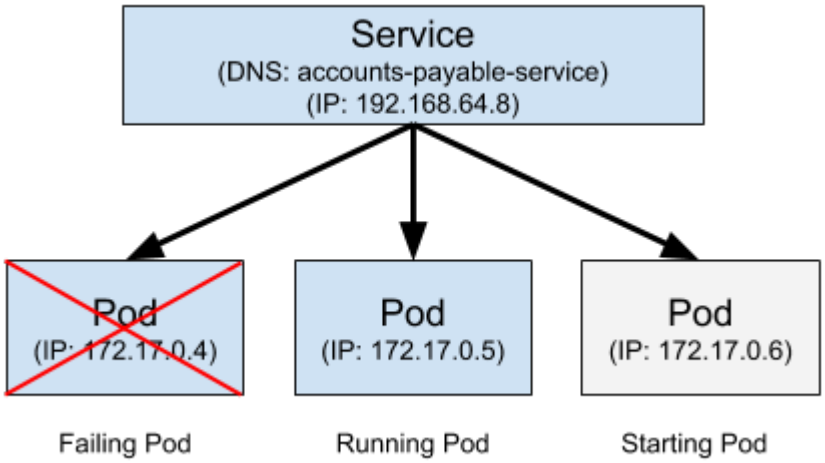


Figure 1.6 Kubernetes Service

Microservices are deployed within pods, and pods come and go, each with their own IP address. This is reflected in Figure 1.6. For example, the replication controller will scale the number of pods, either up or down, to meet the specified number of replicas (running pods). The Accounts Payable Service has three replicas. The pod at IP address 172.17.0.4 is failing, and will need to be replaced with a new pod. The pod at IP address 172.17.0.5 is running and receiving traffic. The pod at IP address 172.17.0.6 is starting and will be able to serve traffic once booted. There is quite a bit of instability in this example with pods, each with their own IP address, failing and starting. Any service, such as the Frontend Web UI microservice service described earlier, needs a stable IP address to connect to.

A Service will create a single IP address, and create a DNS name within the cluster so other microservices can access the service in a consistent manner, and requests will be proxied to one of the replicas.

- **ConfigMap.** Used to store microservice configuration, separating configuration from the

microservice itself. ConfigMaps are clear text. As an option, a Kubernetes Secret can be used to store confidential information.

With the exception of the cluster, each of these concepts are represented by Kubernetes objects. Kubernetes objects are persistent entities that collectively represent the *current* state of the cluster. We can manipulate the cluster by creating, manipulating, and deleting Kubernetes objects. By manipulating the state, we are defining what we want the *desired* state to be. Objects are manipulated by invoking APIs on the Kubernetes API server running on an administration server. The three most popular means of invoking the API server is by using a Web UI such as the Kubernetes Dashboard, by using the `kubectl` CLI to directly manipulate state, or by defining the state in YAML and applying the desired state with `kubectl apply`.

Once a desired state is defined, a Kubernetes cluster updates its *current* state to match the *desired* state. This is done by using the *controller pattern*. Controllers monitor the state of the cluster, and when a controller is notified of a state change, it reacts to that change by updating the *current* state to match the *desired* state. For example, if a Replication Controller sees a change to a ReplicationController object from a current state of 3 replicas to a desired state of 2 replicas, the Replication Controller will kill one of the pods.

Defining Kubernetes objects using YAML and applying it with `kubectl` is very popular among administrators, but not all Java developers have embraced YAML. Luckily, we'll be able to avoid YAML by using the Quarkus Kubernetes Extension that lets us define the desired state using a property file. When building the application the kubernetes deployment YAML is generated automatically. The YAML can be applied automatically as a part of the Quarkus build process, or it can be applied manually using `kubectl`.

1.5 Kubernetes native microservices

What does it mean to develop Kubernetes native microservices? It's developing a microservice with the understanding Kubernetes is the underlying deployment platform, and is facilitated by having a Kubernetes runtime like Quarkus. How is this different from any other microservice, or the frequently mentioned "Cloud Native Java"? Here are some differentiating characteristics:

- **Low Memory consumption.** A Kubernetes cluster is shared infrastructure, and organizations want to extract as much value out of their Kubernetes investment by consolidating as many services across as many departments on a Kubernetes cluster as possible. Reduced memory consumption is a gating factor. Until runtimes like Quarkus, organizations were considering leaving Java runtimes for Node.js or Golang in order to better utilize their Kubernetes clusters.
- **Fast startup.** Kubernetes can automatically create new microservice instances to meet demand. Without fast startup, existing instances can become overloaded and fail before new instances come online, impacting overall application stability. This potential complication can also impact rolling upgrades when a new version of a service is incrementally deployed to replace an existing one.

- **Minimize operating system threads.** A Kubernetes Node may be running hundreds of microservice instances, each of which may have up to hundreds of threads. It is not uncommon for a thread to consume a megabyte of memory. In addition, the operating system scheduler works increasingly harder as the number of threads increases. Quarkus runs its asynchronous, reactive, and (by default) traditionally thread-blocking imperative APIs on an event loop, which significantly reduces the number of threads.
- **Consume Kubernetes ConfigMaps.** Services deployed to Kubernetes can be configured using a Kubernetes ConfigMap. A ConfigMap is a file that is typically mounted to a Pod filesystem. However, Quarkus can seamlessly use the Kubernetes client API to access a ConfigMap without mounting the filesystem in the pod, simplifying configuration.
- **Expose Health Endpoints.** A service should always expose its health so Kubernetes can restart an unhealthy service or redirect traffic away from a pod that is temporarily unavailable. In addition to supporting custom health checks, Quarkus has built-in data source and messaging client (AMQ and Kafka) readiness health checks to automatically pause traffic when those backend services are unavailable.
- **Support CNCF projects.** CNCF is the "Cloud Native Computing Foundation", which is responsible for the evolution of Kubernetes and related projects like Prometheus monitoring (using the OpenMetrics format) and Jaeger (using OpenTracing/OpenTelemetry).
- **Inherent Kubernetes deployment support.** Quarkus has built-in support for deploying to Kubernetes. It enables a developer to compile, package, and deploy a microservice to Kubernetes using one line maven (or Gradle) command. In addition, Quarkus requires no Kubernetes YAML expertise. Kubernetes YAML is generated automatically and can be customized using Java properties.
- **Kubernetes Client API.** Quarkus includes a Java-friendly API for interacting with a Kubernetes cluster. This enables programmatic access to any Kubernetes capability and extend or tailor it for enterprises needs.

1.6 Summary

- A microservice models and implements a subset of business functionality called a bounded context.
- A microservices architecture is a collection of evolving, collaborating microservices.
- MicroProfile is a collection of microservices specifications that facilitate the creation of portable microservices across multiple implementations.
- Microservices have evolved from running in a shared environment like an application server to running on a single application stack.
- Kubernetes has replaced the application server as the shared application environment
- Quarkus is a Java single application stack that can efficiently run MicroProfile applications on Kubernetes.

2 Your first Quarkus application

This chapter covers:

- Creating a Quarkus project
- Developing with Quarkus Live Coding
- Writing tests for a Quarkus microservice
- Deploying and running a microservice to Kubernetes

Throughout the book we will be using the domain of banking to create microservice examples, highlighting key concepts from each chapter.

Throughout this chapter, the example will be an *Account Service*. The purpose of the *Account Service* is to manage bank accounts, holding information like customer name, balance, and overdraft status. In developing the *Account Service*, the chapter will cover the ways to create Quarkus projects, developing with *Live Coding* for real time feedback, writing tests, building native executables for an application, how to package an application for Kubernetes, and how to deploy to Kubernetes.

There's a lot to cover, let's dive into creating the *Account Service*!

2.1 Creating a project

There are several ways to create a microservice using Quarkus:

1. With the project generator at https://code.quarkus.io/
2. In a terminal with the Quarkus Maven plugin
3. Manually create the project and include the Quarkus dependencies and plugin configuration

Of the above options, option 3 is the more complicated and prone to errors, so we won't cover this option.

Option 2 would use a command such as:

```
mvn io.quarkus:quarkus-maven-plugin:1.11.3.Final:create \
    -DprojectGroupId=quarkus \
    -DprojectArtifactId=account-service \
    -DclassName="quarkus.accounts.AccountResource" \
    -Dpath="/accounts"
```

For the *Account Service* we will use option 1, using the project generator at https://code.quarkus.io/.

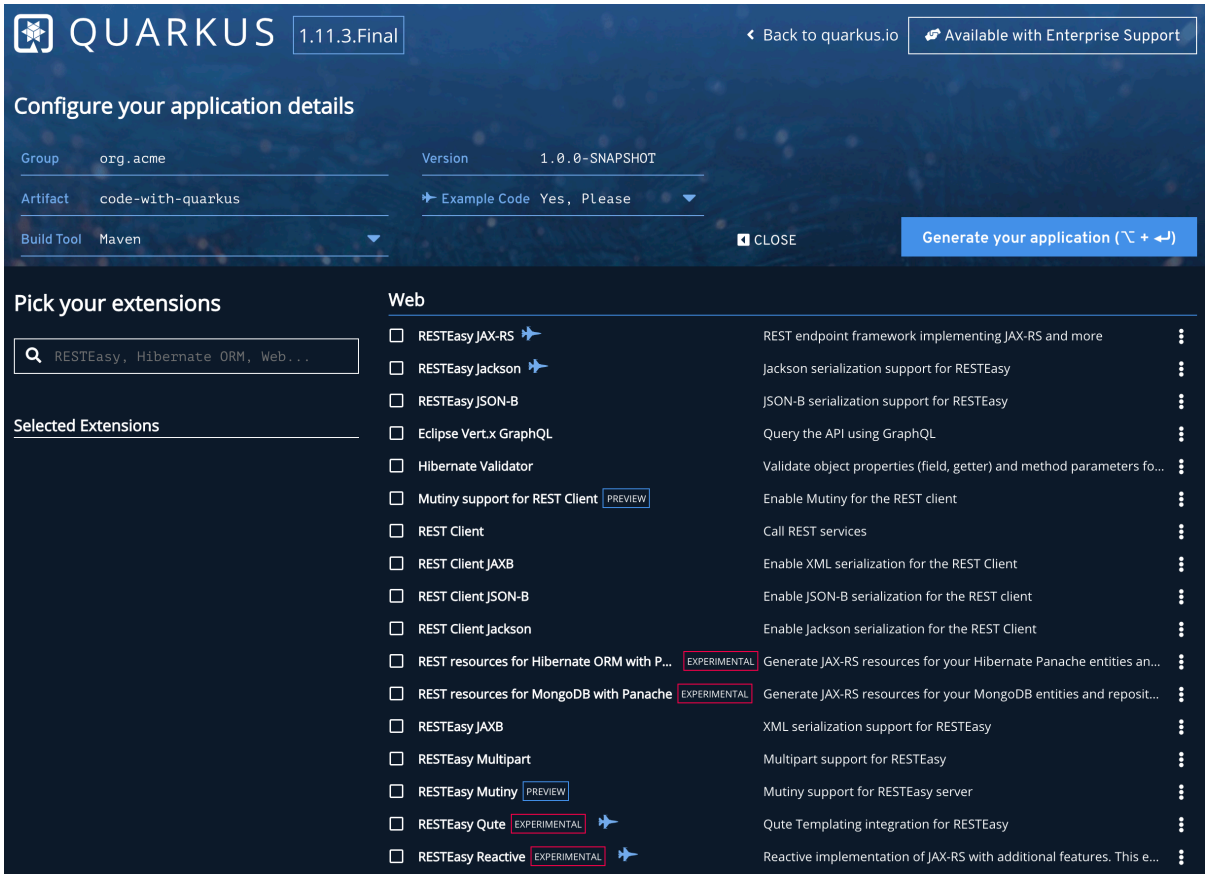


Figure 2.1 Quarkus project generator

Figure 2.1 is a view of the Quarkus project generator, at the time the screenshot was taken. The top left of the page contains fields for customizing project information, such as the group and artifact ids, and the build tool for the project.

The bottom right of the page shows all the possible extensions that can be selected for the application.

> **TIP** The Quarkus project generator lists hundreds of extensions. The search box can filter the list of available extensions to more quickly locate a particular set of extensions.

Select the **RESTEasy JAX-RS** extension, and leave *Example Code* set to *Yes, Please*:

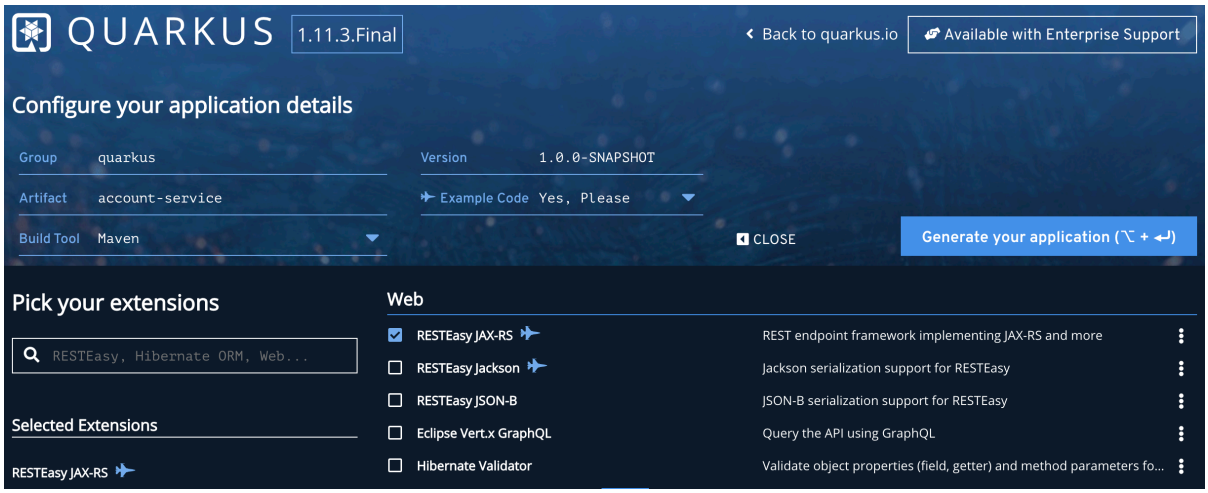


Figure 2.2 Quarkus project generator - more options [TODO: Update version]

Figure 2.2 has all the changes we've made to the generator for the *Account Service*. The group has been set to `quarkus`, the artifact to `account-service`, and **RESTEasy JAX-RS** extension selected.

Once the changes have been made, click on **Generate your application** as seen below:

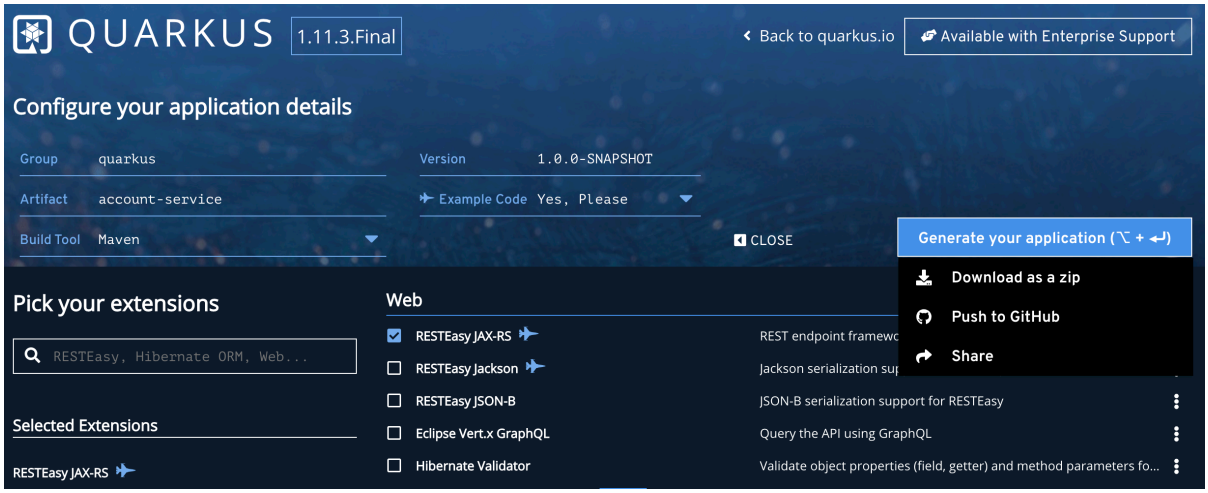


Figure 2.3 Quarkus project generator - generate application [TODO: Update version]

Figure 2.3 highlights the options we have for generating the project:

- Download as a zip

- Push to GitHub
- Share

Select **Download as a zip**, and a zip containing the project source will be created and downloaded. **Share** provides the option to share the configured project with others, or share the link to the zip that is generated from the configuration.

Once the zip has downloaded, extract the contents into a directory. The generated contents will be explained shortly, but first open a terminal window and change into the directory where the zip was extracted. In that directory, run the following command:

```
mvn quarkus:dev
```

Maven artifacts and their dependencies will have to be downloaded the first time a particular version of Quarkus is used.

Listing 2.1 Quarkus startup [TODO update version]

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (Quarkus Main Thread) account-service 1.0.0-SNAPSHOT on JVM
    (powered by Quarkus 1.11.3.Final) started in 1.201s.
    Listening on: http://localhost:8080
INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [cdi, resteasy]
```

Listing 2.1 contains the console output when Quarkus starts the project. The output includes the version used, in this case `1.11.3.Final`, and installed features include `cdi` and `resteasy`.

Once started, the application can be accessed at *http://localhost:8080*:

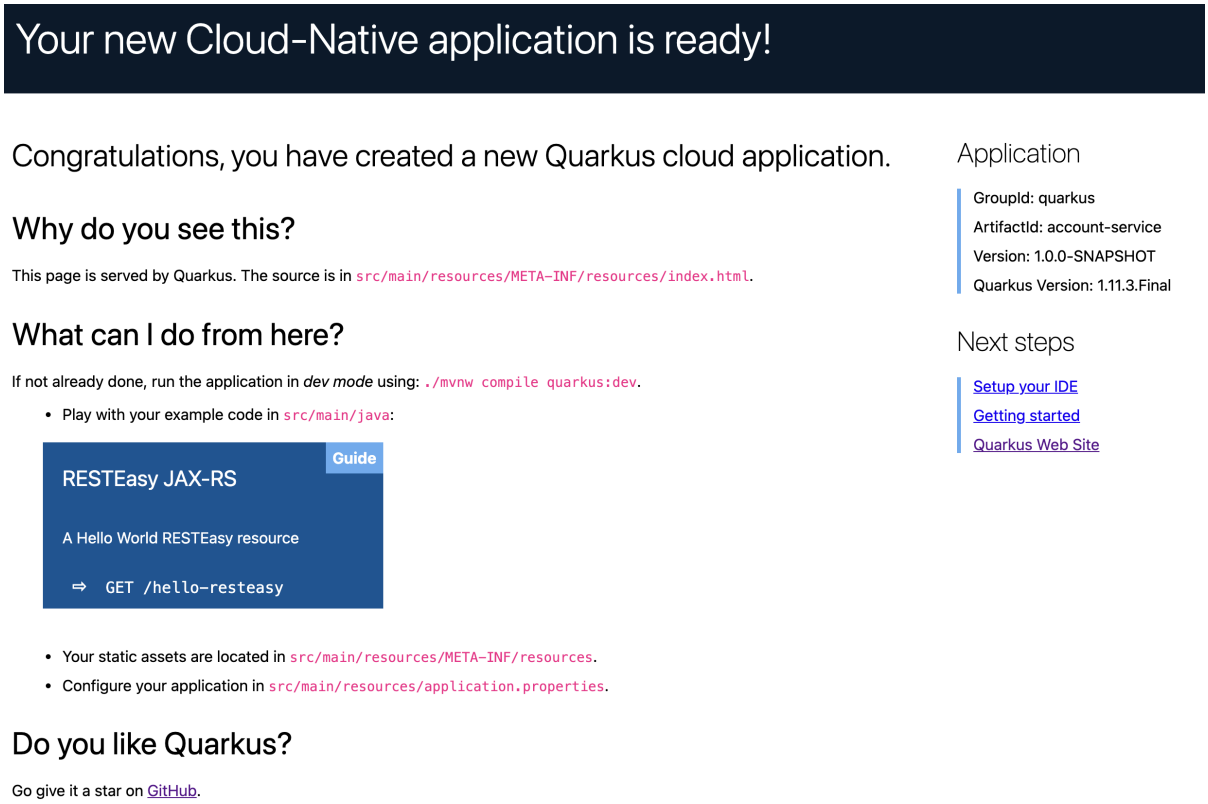


Figure 2.4 Quarkus default index page[TODO: Update version]

The default page of the generated application, shown in *Figure 2.4*, provides some pointers on what can be done next for creating REST endpoints, servlets, and static assets.

In addition to the default index page, open *http://localhost:8080/hello-resteasy* to be greeted by the generated JAX-RS resource!

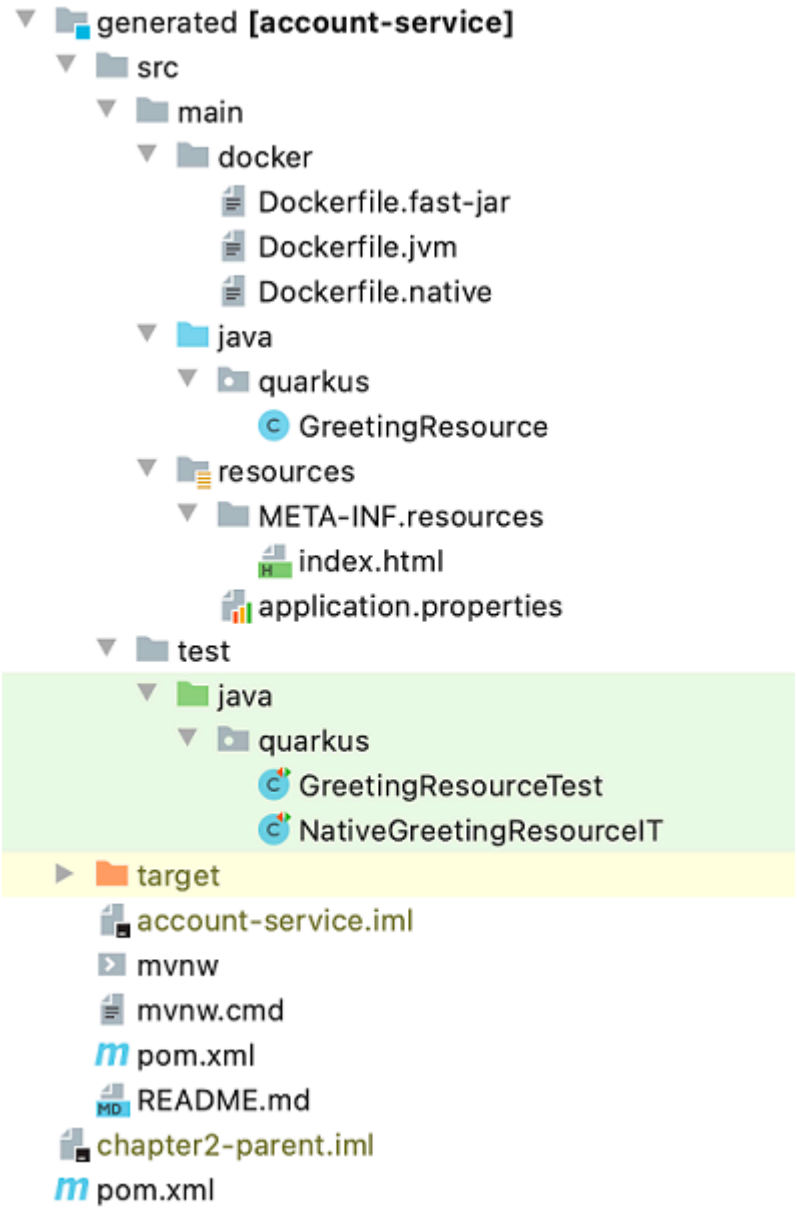


Figure 2.5 Quarkus generated project structure

With the generated application running, take a look through what the project includes from the generation process in 2.5. Open up the project in an editor, or whatever tool might be preferred.

The project root contains the build file, in this case *pom.xml*, a *README.md* with information on how to run the project, and Maven wrappers for those that may not have Maven installed already.

Looking in *src/main* there are directories for Docker files, Java source files, and other resources. In the *docker* directory there are Dockerfiles for the JVM, native executable, and fast-jar format. Native executables will be discussed in "Creating a native executable" in section 2.4.

Each of the Docker files use the Red Hat Universal Base Image (UBI) as their base. Full details on the image content can be found here:

https://catalog.redhat.com/software/containers/ubi8/ubi-minimal/5c359a62bed8bd75a2c3fba8

Within the Java source directory, *src/main/java*, there is the `quarkus` package. Inside the package is the `GreetingResource` class, containing a JAX-RS resource endpoint:

Listing 2.2 GreetingResource

```
@Path("/hello-resteasy")                    ❶
public class GreetingResource {

    @GET                                    ❷
    @Produces(MediaType.TEXT_PLAIN)         ❸
    public String hello() {
        return "Hello RESTEasy";            ❹
    }
}
```

❶ Defines the JAX-RS resource to respond at */hello-resteasy*

❷ Method responds to an HTTP GET request

❸ Response to the browser will set the content type to TEXT_PLAIN

❹ Return "Hello RESTEasy" as the HTTP GET response

Take a look at the next directory, *src/main/resources*. The first file is *application.properties*. This is where any configuration packaged within the application should be placed. Configurations can also reside outside the application, but these are restricted to aspects that can be configured at runtime.

> **NOTE** **The different types of configuration will be discussed in Chapter 3. Including the ability to use application.yaml instead of a properties file.**

Currently, there is no configuration in *application.properties*, but configuration will be added soon.

Also in *src/main/resources*, is the *META-INF/resources* directory. Any static assets for the application should be placed in this directory. Inside the directory is the static *index.html* that created the page seen in *Figure 2.4*.

Moving on from what was generated in *src/main/*, next is *src/test*. Here there are two classes, *GreetingResourceTest* and *NativeGreetingResourceIT*. The first uses `@QuarkusTest` to run a unit test, on the JVM, verifying the endpoint returns *hello* as expected:

Listing 2.3 GreetingResourceTest

```
@QuarkusTest                                    ❶
public class GreetingResourceTest {
    @Test                                       ❷
    public void testHelloEndpoint() {
        given()
          .when().get("/hello-resteasy")        ❸
          .then()
             .statusCode(200)
             .body(is("Hello RESTEasy"));       ❹
    }
}
```

❶ Tells JUnit to use the Quarkus extension, which starts the application for the test

❷ Regular JUnit test method marker

❸ Use RestAssured to access the */hello-resteasy* URL

❹ Verify the response had a body that contained *Hello RESTEasy*

`NativeGreetingResourceIT` runs the same tests, but with the native executable of the application:

Listing 2.4 NativeGreetingResourceIT

```
@NativeImageTest                                ❶
public class NativeGreetingResourceIT
    extends GreetingResourceTest {              ❷
    // Execute the same tests but in native mode.
}
```

❶ Tells JUnit to use the Quarkus native executable extension

❷ Extends from the JUnit unit tests to reuse them

NOTE **It's not required to run the same set of tests with a native executable and the JVM. However, it is a convenient means of testing on the JVM and a native executable with a single set of common tests.**

Having looked through what the project generator creates, all Java source files, and the *index.html* file, can be deleted. Don't modify the Dockerfiles, *application.properties*, or Java packages for now.

2.2 Developing with Live Coding

With a blank application, it's time to develop the *Account Service*. For developing the service, *Live Coding* functionality of Quarkus will be utilized.

Using *Live Coding* enables Java source, resources, and configuration of a running application to

be updated. All changes will be reflected in the running application automatically, enabling developers to improve the turnaround time when developing a new application.

Live Coding enables hot deployment via background compilation. Any changes to Java source, or resources, will be reflected as soon as the application receives a new request from the browser. Refreshing the browser, or issuing a new browser request, triggers a scan of the project for any changes to then recompile and redeploy the application. If there are issues with compilation or deployment, an error page will provide details of the problem.

To begin, create a minimal JAX-RS resource:

Listing 2.5 AccountResource

```
@Path("/accounts")
public class AccountResource {
}
```

There's not much there right now, just a JAX-RS resource that defines a URL path of */accounts*. There are no methods to respond to any requests, but restart *Live Coding* if it had been stopped:

```
mvn compile quarkus:dev
```

TIP **Live Coding will handle the deletion and creation of new files without issue while it's still running.**

In the terminal window there will be output similar to:

```
Listening for transport dt_socket at address: 5005
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2020-06-18 20:31:00,626 INFO  [io.quarkus] (Quarkus Main Thread) chapter2-accoun
t-service 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.5.1.Final) started in 0.84
8s. Listening on: http://0.0.0.0:8080
2020-06-18 20:31:00,643 INFO  [io.quarkus] (Quarkus Main Thread) Profile dev act
ivated. Live Coding activated.
2020-06-18 20:31:00,643 INFO  [io.quarkus] (Quarkus Main Thread) Installed featu
res: [cdi, resteasy]
```

Figure 2.6 Account Service startup

Notice the first line indicates that a debugger has been started on port *5005*. This is an added benefit to using *Live Coding*, Quarkus opens the default debug port for the application.

Opening a browser to *http://localhost:8080* will show:

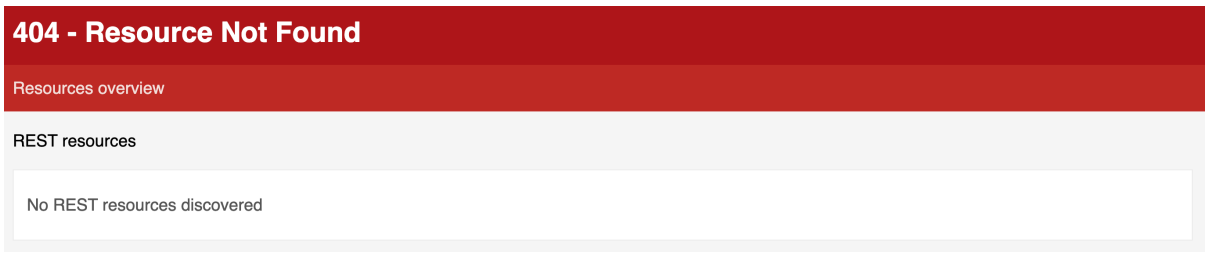


Figure 2.7 Account Service no Resources

Don't be concerned with the error, it makes sense as the JAX-RS resource defined a URL path and no methods to process HTTP requests. If accessing *http://localhost:8080/accounts*, the same error message is in the browser.

Now it's time to start developing some code. While *Live Coding* is still going, create the *Account* POJO to represent a bank account in the system:

Listing 2.6 Account

```
public class Account {
  public Long accountNumber;
  public Long customerNumber;
  public String customerName;
  public BigDecimal balance;
  public AccountStatus accountStatus = AccountStatus.OPEN;

  public Account() {
  }

  public Account(Long accountNumber, Long customerNumber, String customerName,
  BigDecimal balance) {
    this.accountNumber = accountNumber;
    this.customerNumber = customerNumber;
    this.customerName = customerName;
    this.balance = balance;
  }

  public void markOverdrawn() {
    accountStatus = AccountStatus.OVERDRAWN;
  }

  public void removeOverdrawnStatus() {
    accountStatus = AccountStatus.OPEN;
  }

  public void close() {
    accountStatus = AccountStatus.CLOSED;
    balance = BigDecimal.valueOf(0);
  }

  public void withdrawFunds(BigDecimal amount) {
    balance = balance.subtract(amount);
  }

  public void addFunds(BigDecimal amount) {
    balance = balance.add(amount);
  }

  public BigDecimal getBalance() {
    return balance;
  }

  public Long getAccountNumber() {
    return accountNumber;
  }

  public String getCustomerName() {
    return customerName;
  }

  public AccountStatus getStatus() {
    return accountStatus;
  }
}
```

Account has some fields to hold data about the account: account number, customer number, customer name, balance, and account status. It has a constructor that takes values to populate the fields with, except for the account status as that defaults to `OPEN`. After there are methods for setting and clearing the overdrawn status, closing the account, adding and withdrawing account funds, and lastly some getters for balance, account number, and customer name.

Not a lot to it, but it's a foundation to build from.

Right now it won't compile, as `AccountStatus` needs to be created:

Listing 2.7 AccountStatus

```
public enum AccountStatus {
  OPEN,
  CLOSED,
  OVERDRAWN
}
```

There's nothing there yet, but open up *http://localhost:8080/accounts* to show the error page. With *Live Coding* running, open up `pom.xml` and change the `quarkus-resteasy` dependency to be `quarkus-resteasy-jsonb`. Doing this adds support for returning JSON objects in the endpoints.

NOTE	**Instead of `quarkus-resteasy-jsonb`, `quarkus-resteasy-jackson` could also be used.**

IMPORTANT	**Modifying dependencies in `pom.xml` can be done with Live Coding, but there will be a longer delay before restarting is complete if new dependencies need to be downloaded.**

To begin creating the *Account Service*, open up `AccountResource` and add the following:

Listing 2.8 AccountResource

```
@Path("/accounts")
public class AccountResource {
  @GET
  @Produces(MediaType.APPLICATION_JSON)     ❶
  public Set<Account> allAccounts() {       ❷
    return Collections.emptySet();
  }
}
```

❶ Indicates the response is converted to JSON.

❷ Return a `Set` of `Account` objects.

To add some data, add the following to `AccountResource`:

Listing 2.9 AccountResource

```
@Path("/accounts")
public class AccountResource {
  Set<Account> accounts = new HashSet<>();      ❶

  @PostConstruct                                ❷
  public void setup() {                         ❸
    accounts.add(new Account(123456789L, 987654321L, "George Baird", new BigDecimal("354.23")));
    accounts.add(new Account(121212121L, 888777666L, "Mary Taylor", new BigDecimal("560.03")));
    accounts.add(new Account(545454545L, 222444999L, "Diana Rigg", new BigDecimal("422.00")));
  }
  ...
}
```

❶ Create a `Set` of `Account` objects to hold the state.

❷ `@PostConstruct` indicates the method should be called straight after creation of the CDI Bean.

❸ `setup()` pre populates some data into the list of accounts.

NOTE **Though the JAX-RS resource does not specify a CDI Scope annotation, Quarkus defaults JAX-RS resources to `@Singleton`. To ensure the JAX-RS resource is in the preferred CDI Scope, `@Singleton`, `@ApplicationScoped`, or `@RequestScoped` can be added to `AccountResource`.**

Right now `allAccounts()` returns an empty `Set`, change it to return the `accounts` field:

Listing 2.10 AccountResource

```
@Path("/accounts")
@ApplicationScoped
public class AccountResource {
  ...
  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public Set<Account> allAccounts() {
    return accounts;
  }
  ...
}
```

Refresh the browser window open to *http://localhost:8080/accounts*:

```
[
    {
        "accountNumber": 121212121,
        "customerNumber": 888777666,
        "customerName": "Mary Taylor",
        "balance": 560.03,
        "accountStatus": "OPEN"
    },
    {
        "accountNumber": 123456789,
        "customerNumber": 987654321,
        "customerName": "George Baird",
        "balance": 354.23,
        "accountStatus": "OPEN"
    },
    {
        "accountNumber": 545454545,
        "customerNumber": 222444999,
        "customerName": "Diana Rigg",
        "balance": 422,
        "accountStatus": "OPEN"
    }
]
```


Figure 2.8 Account Service - All Accounts

In *Figure 2.8* the page has reloaded to show all the accounts that are stored in the service.

NOTE	**Figure 2.8 uses the JSON Formatter extension for Chrome to format the JSON response. Such an extension provides a better means of viewing the structure of the JSON document.**

Next up is creating a method for retrieving a single *Account* instance:

Listing 2.11 AccountResource

```
@Path("/accounts")
@ApplicationScoped
public class AccountResource {
  ...
  @GET
  @Path("/{accountId}")                                    ❶
  @Produces(MediaType.APPLICATION_JSON)
  public Account getAccount(@PathParam("accountId") Long accountId) {    ❷
    Account response = null;
    for (Account acct : accounts) {
      if (acct.getAccountNumber().equals(accountId)) {                   ❸
        response = acct;
        break;
      }
    }

    if (response == null) {                                  ❹
      throw new NotFoundException("Account with id of " + accountId + " does not exist.");
    }

    return response;
  }
  ...
}
```

❶ Defines the name of the parameter on the URL path.

❷ `@PathParam` maps the `accountId` URL parameter into the `accountId` method parameter.

❸ Search through the accounts and find one with an account id that matches `accountId`.

❹ Return a `NotFoundException` if no matching account is found.

With the above changes, open *http://localhost:8080/accounts/121212121* in a browser to see the account details in a JSON document.

Quarkus has a nice feature with *Live Coding* for showing available URLs when accessing a URL that doesn't exist. Open *http://localhost:8080/accounts/5* in a browser:

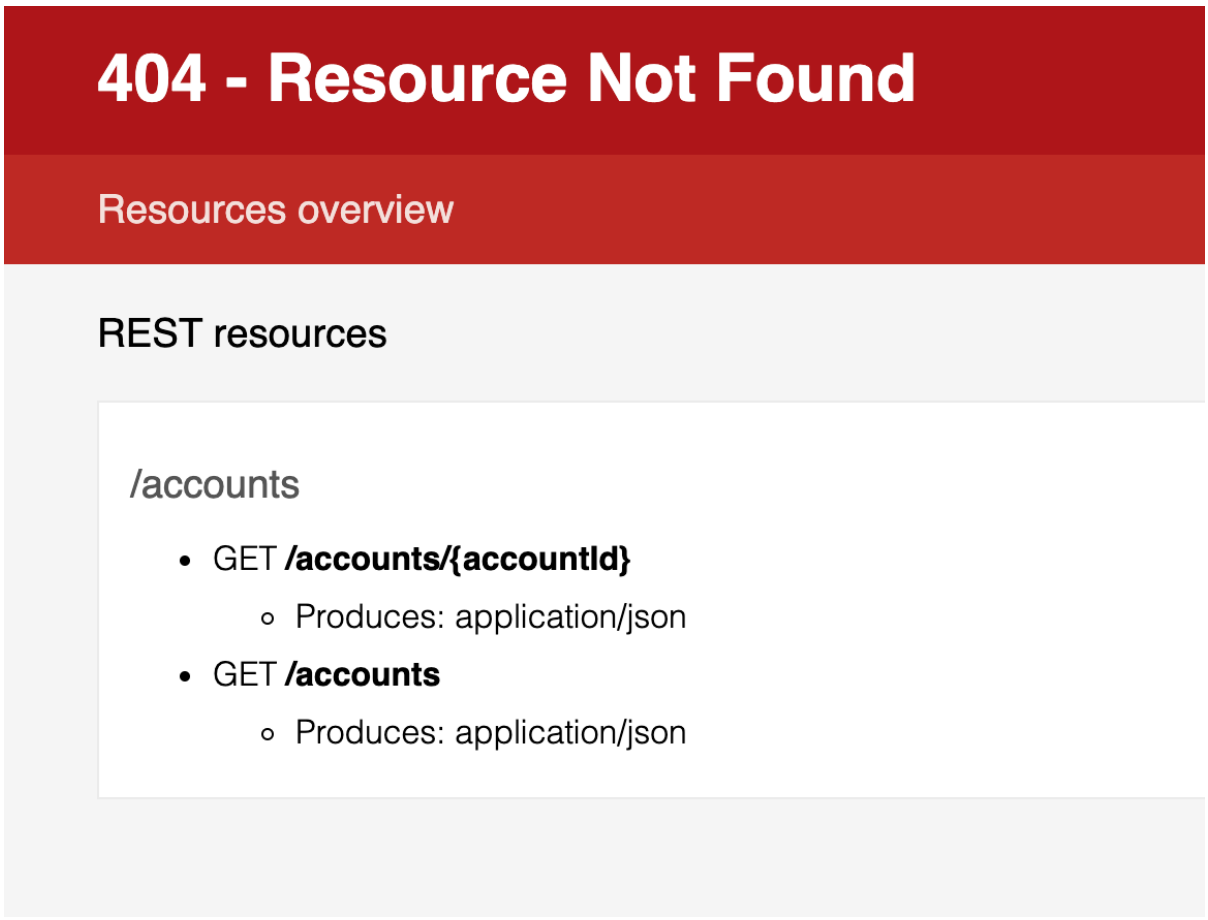


Figure 2.9 Quarkus Error page

Not finding an account number, the response is an HTTP 404 seen in *Figure 2.9*, but the page offers useful information about what endpoints are available. In this case, there is the main */accounts/* URL path, and the two URL paths within it that have been created.

As the endpoint we accessed was valid, but the requested record was not found, there is a nicer 404 response that can be created to provide more details. Instead of `getAccount()` throwing a `NotFoundException` when no record is found, change it to `WebApplicationException` and pass `404` as the response code:

Listing 2.12 AccountResource.getAccount()

```
if (response == null) {
  throw new WebApplicationException("Account with id of " + accountId + " does not exist.", 404);
}
```

To convert the exception into a meaningful response, create a JAX-RS exception mapper in `AccountResource`:

Listing 2.13 AccountResource

```
@Path("/accounts")
@ApplicationScoped
public class AccountResource {
  ...
  @Provider                                                    ❶
  public static class ErrorMapper implements ExceptionMapper<Exception> {     ❷

    @Override
    public Response toResponse(Exception exception) {          ❸

      int code = 500;
      if (exception instanceof WebApplicationException) {      ❹
        code = ((WebApplicationException) exception).getResponse().getStatus();
      }

      JsonObjectBuilder entityBuilder = Json.createObjectBuilder()      ❺
          .add("exceptionType", exception.getClass().getName())
          .add("code", code);

      if (exception.getMessage() != null) {                    ❻
        entityBuilder.add("error", exception.getMessage());
      }

      return Response.status(code)                             ❼
          .entity(entityBuilder.build())
          .build();
    }
  }
}
```

❶ `@Provider` indicates the class is an auto discovered JAX-RS Provider.

❷ Implements `ExceptionMapper` for all `Exception` types.

❸ Overrides the `toResponse` method for converting the exception to a `Response`.

❹ Check for `WebApplicationException` and extract the HTTP status code, otherwise defaults to 500.

❺ Use builder to construct JSON formatted data containing exception type and HTTP status code.

❻ If there is a message, add it to the JSON object.

❼ Return a `Response` with the HTTP status code and JSON object.

```
{
    "exceptionType": "javax.ws.rs.WebApplicationException",
    "code": 404,
    "error": "Account with id of 5 does not exist."
}
```

Figure 2.10 Account not found

As an exercise for the reader, add methods to `AccountResource` for creating, updating, and deleting accounts. The full code for `AccountResource` will be in */chapter2/account-service*.

2.3 Writing a Test

The *Account Service* has methods for:

- Retrieving all accounts
- Retrieving a single account
- Create a new account
- Update an account
- Delete an account

However, there is no verification that what has been coded actually works. Only retrieving all accounts and retrieving a single account have been verified, by accessing specific URLs from a browser to trigger HTTP GET requests. Even with the manual verification, any additional changes that might be made are not being verified. Unless manual verification follows every change.

It's important to ensure the developed code has been tested and verified appropriately against expected outcomes. For that, it is necessary to add, at a minimum, some level of tests for the code.

Quarkus supports running JUnit 5 tests with the addition of `@QuarkusTest` onto a test class. `@QuarkusTest` informs JUnit 5 of the extension to utilize during the test. The extension performs the necessary augmentation of the service being tested, equivalent to what happens during compilation with the Quarkus Maven or Gradle plugin. Prior to running the tests, the extension starts the constructed Quarkus service, just as if it was constructed with any build tool.

To begin adding tests to the *Account Service*, add the following dependencies in the `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
</dependency>
```

Generating the project from https://code.quarkus.io, the *Account Service* already includes the testing dependencies.

NOTE `rest-assured` **is not a required dependency for testing, but it offers a convenient means of testing HTTP endpoints. It would be possible to use different testing libraries for the same purpose, but the examples that follow all use** `rest-assured`**. In addition, using** `rest-assured` **has a dependency on Hamcrest for asserting and matching test data.**

The project generator also setup the Maven Surefire plugin for testing:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>${surefire-plugin.version}</version>                    ❶
  <configuration>
    <systemPropertyVariables>
      <java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>   ❷
    </systemPropertyVariables>
  </configuration>
</plugin>
```

❶ Set to a version of the Surefire plugin that works with JUnit 5. A minimum of 2.22.1 is required.

❷ Set a system property to ensure the tests use the correct log manager.

Here is a test case to verify retrieving all accounts returns the expected result:

Listing 2.14 AccountResourceTest

```
@QuarkusTest
public class AccountResourceTest {
  @Test                                          ❶
  void testRetrieveAll() {                       ❷
    Response result =
        given()
          .when().get("/accounts")               ❸
          .then()
            .statusCode(200)                     ❹
            .body(
                containsString("George Baird"),  ❺
                containsString("Mary Taylor"),
                containsString("Diana Rigg")
            )
            .extract()
            .response();                         ❻

    List<Account> accounts = result.jsonPath().getList("$");   ❼
    assertThat(accounts, not(empty()));                        ❽
    assertThat(accounts, hasSize(3));                          ❾
  }
}
```

❶ Declares the method as a test method.

❷ With JUnit 5 test methods don't need to be public.

❸ Issue an HTTP GET request to */accounts* URL.

❹ Verify the response had a 200 status code, meaning it returned without problem.

❺ Verify the body contains all customer names.

❻ Extract the response.

❼ Extract the JSON Array and convert to a `List` of `Account` objects.

❽ Assert the array of `Account` objects is not empty.

❾ Assert the array of `Account` objects has three items.

One test method is not sufficient to ensure prevention of future breakages. Next up is a test method for verifying retrieval of a single `Account`:

Listing 2.15 AccountResourceTest

```
@Test
void testGetAccount() {
  Account account =
      given()
          .when().get("/accounts/{accountId}", 545454545)          ❶
          .then()
            .statusCode(200)
            .extract()
            .as(Account.class);

  assertThat(account.getAccountNumber(), equalTo(545454545L));      ❷
  assertThat(account.getCustomerName(), equalTo("Diana Rigg"));
  assertThat(account.getBalance(), equalTo(new BigDecimal("422.00")));
  assertThat(account.getStatus(), equalTo(AccountStatus.OPEN));
}
```

❶ Pass the id of the account to be retrieved as a URL path parameter.

❷ Verify the account response object with expected values.

The tests written so far do not verify updating or adding data with the *Account Service*, they only verified that existing data returns with the correct values. Next, add a test to verify that creation of a new account succeeds.

There are multiple facets to testing creation. In addition to verifying creation of the new account, the test needs to ensure that the list of all accounts includes the new account. When including tests for mutating the state within a service, it becomes necessary to order the execution sequence of tests.

Why is it necessary to order the test execution? When there is a test to create, delete, or update the state within a service, it will impact any tests that read the state. For instance, in the above test to retrieve all accounts, *Listing 2.14*, the expectation is it returns three accounts. However, when the test method execution order is non-deterministic, i.e. not in a defined order, it's possible for the test creating an account to execute before *Listing 2.14*, causing it to fail by finding four accounts.

To define the test method execution order, add `@TestMethodOrder(OrderAnnotation.class)`

onto the test class definition. Above or below `@QuarkusTest` is fine. `@Order(x)` is added to each test method, where `x` is a number to indicate where in the execution sequence of all tests is this particular test. `testRetrieveAll()` and `testGetAccount()` can either be `Order(1)` or `Order(2)`, as they don't mutate data, so it does not matter.

Listing 2.16 AccountResourceTest

```
@Test
@Order(3)                                          ❶
void testCreateAccount() {
  Account newAccount = new Account(324324L, 112244L, "Sandy Holmes", new BigDecimal("154.55"));

  Account returnedAccount =
    given()
      .contentType(ContentType.JSON)               ❷
      .body(newAccount)                            ❸
      .when().post("/accounts")                    ❹
      .then()
        .statusCode(201)                           ❺
        .extract()
        .as(Account.class);

  assertThat(returnedAccount, notNullValue());                ❻
  assertThat(returnedAccount, equalTo(newAccount));

  Response result =
    given()
      .when().get("/accounts")                               ❼
      .then()
        .statusCode(200)
        .body(
            containsString("George Baird"),
            containsString("Mary Taylor"),
            containsString("Diana Rigg"),
            containsString("Sandy Holmes")                   ❽
        )
        .extract()
        .response();

  List<Account> accounts = result.jsonPath().getList("$");
  assertThat(accounts, not(empty()));
  assertThat(accounts, hasSize(4));                        ❾
}
```

❶ Define the test execution order to be third, after the retrieve all and get account tests.

❷ Set the content type to JSON for the HTTP POST.

❸ Set the new account object into the body of the HTTP POST.

❹ Send the HTTP POST request to */accounts* URL.

❺ Verify the HTTP status code returned is 201, indicating it was created successfully.

❻ Assert that the account from the response was not null and equals the account we posted.

❼ Send an HTTP GET request to */accounts* URL, for retrieving all accounts.

❽ Verify the response contains the name of the customer on the new account.

❾ Assert there are now four accounts.

Open a terminal window in the directory where the *Account Service* is located and run the tests:

```
mvn test
```

```
[ERROR] Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 3.477 s
<<< FAILURE! - in quarkus.accounts.AccountResourceTest
[ERROR] testCreateAccount  Time elapsed: 0.103 s  <<< FAILURE!
java.lang.AssertionError:
1 expectation failed.
Expected status code <201> but was <200>.

        at quarkus.accounts.AccountResourceTest.testCreateAccount(AccountResourc
eTest.java:72)
```

Figure 2.11 Create Account test failure

Figure 2.11 shows the error when running the tests. Though creating an account should have returned a *201* HTTP status code, the test received *200* instead. Though the request succeeded, it didn't return an expected HTTP status code.

To fix it, instead of returning the created `Account` instance, the method should return a `Response` to enable the appropriate HTTP status code to be set. Here is the updated create method:

Listing 2.17 AccountResource

```
@Path("/accounts")
@ApplicationScoped
public class AccountResource {
  ...
  @POST
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  public Response createAccount(Account account) {
    if (account.getAccountNumber() == null) {
      throw new WebApplicationException("No Account number specified.", 400);
    }

    accounts.add(account);
    return Response.status(201).entity(account).build();      ❶
  }
  ...
}
```

❶ Construct a `Response` with status code `201` containing the new account entity.

Running `mvn test` again shows a different error. Now it fails because the two accounts, the one sent in the HTTP POST request and the one returned, are not equal. The test failure is:

```
Expected: <quarkus.accounts.repository.Account@22361e23>
     but: was <quarkus.accounts.repository.Account@46994f26>
    at quarkus.accounts.activerecord.AccountResourceTest.testCreateAccount(AccountResourceTest.java:77)
```

`Account` doesn't have any equals or hashcode methods, making any equality check use the default object comparison. Which in this case means they are not the same object. To fix it, update `Account` with equals and hashcode methods:

Listing 2.18 Account

```
public class Account {
  ...
  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Account account = (Account) o;
    return accountNumber.equals(account.accountNumber) &&
        customerNumber.equals(account.customerNumber);
  }

  @Override
  public int hashCode() {
    return Objects.hash(accountNumber, customerNumber);
  }
  ...
}
```

NOTE **The equality check and hashcode creation only use the account and customer numbers. All the other data on Account can change, and it still represents the same instance. It's very important to ensure objects have an appropriately unique business key.**

Running `mvn test` again, all tests now pass.

In future sections, and chapters, additional aspects of testing will be discussed. These include running tests with Native executables, and defining required resources for tests.

2.4 Creating a Native executable

Java programs require a Java Virtual Machine (JVM), as their operating system, for execution. The JVM includes all the low level Java APIs wrapping operating system libraries, as well as convenience APIs to simplify Java programming. The JVM, including all the APIs it provides, is not small. It occupies large parts of memory, measured by its Resident Set Size (RSS), when running Java programs.

Native executables are files containing programs to be executed directly on an operating system, only relying on operating system libraries to be present. Embedded within them are all the necessary operating system instructions required by a particular program. The key difference between a *native executable* and Java programs is that there is no requirement for a JVM to be present during runtime execution.

The ability to compile a Java program down into a *native executable* significantly reduces the file

size of the program because the JVM is no longer required. It also significantly reduces the amount of RSS memory used while executing, and shortens the time required to start the program.

WARNING	The reduction in the program size is a result of the dead code elimination process. There are several aspects of this that impact how code can execute inside a native executable. A key difference is that dynamic class loading will not work, because non directly referenced code is removed from the native executable. Full details of what won't work in a native executable can be found on the GraalVM website: https://www.graalvm.org/docs/Native-Image/user/README.

Over the last couple of years a part of the GraalVM project offering compilation to *native executable* has become popular. GraalVM might sound familiar because of the Truffle compiler sub-project offering polyglot programming on the JVM, but the compilation of Java down to *native executable* is from a different sub project.

native executables are particularly beneficial in serverless environments where processes need to start promptly, and require as few resources as possible.

Quarkus offers first class support for *native executable* creation and optimisation. Such optimisation is possible through *Ahead of Time* (AOT) compilation, build time processing of framework metadata, and native image pre boot.

NOTE	Ahead of Time refers to the process of compiling Java bytecode to a native executable. The JVM only offers Just in Time compilation.

Metadata processing at build time ensures any classes required for initial application deployment are utilized during the build, and are no longer required during runtime execution. This reduces the number of classes needed at runtime, providing the dual benefits of reduced memory utilization and faster startup time.

NOTE	Examples of metadata processing include processing `persistence.xml`, and defining required processing based on annotations in the code.

Quarkus further reduces the number of classes needed at runtime in a *native executable* by performing a pre boot when building the native image. During this phase, Quarkus starts as much of the frameworks as possible within the application and stores the serialized state within the *native executable*. The resulting *native executable* has therefore already run most, if not all, of the necessary startup code for an application, resulting in further improvement to startup time.

In addition to what Quarkus does, GraalVM performs *dead code elimination* on the source and packaged libraries. This process will traverse all execution paths of the code to remove methods and classes that are never on the execution path. Doing so both reduces the size of the *native executable*, and the memory required to run the application.

How does a project create a *native executable*?

In the `pom.xml` for the project, add a profile for the *native executable* creation:

```
<profile>
  <id>native</id>                                                    ❶
  <activation>
    <property>
      <name>native</name>                                            ❷
    </property>
  </activation>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-failsafe-plugin</artifactId>               ❸
        <version>${surefire-plugin.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
            <configuration>
              <systemPropertyVariables>                              ❹
                <native.image.path>${project.build.directory}/${project.build.finalName}
                  -runner</native.image.path>
              </systemPropertyVariables>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  <properties>
    <quarkus.package.type>native</quarkus.package.type>              ❺
  </properties>
</profile>
```

❶ Specifies the id of the profile when activating with "-Pnative".

❷ Defines a flag that when present will activate the profile, "-Dnative".

❸ Include the Failsafe plugin to run integration tests with a native executable.

❹ Define the path to the native executable for use when testing.

❺ Setting this property tells the Quarkus Maven plugin to build a native executable in addition to the usual Java JAR runner.

Generating the project from https://code.quarkus.io, the *Account Service* already includes the **native** profile in the `pom.xml`.

> **NOTE** **Instead of a new profile a native executable can be created by passing `-Dquarkus.package.type=native` to `mvn clean install`. However, having a profile is more convenient and enables integration testing with a native executable.**

Before being able to create a native executable, it's necessary to install GraalVM for the JDK version and operating system in use, in this instance JDK 11. Follow the instructions on the Quarkus website for installing and configuring GraalVM: https://quarkus.io/guides/building-native-image#configuring-graalvm. Pre requisites for GraalVM here: https://quarkus.io/guides/building-native-image#prerequisites-for-oracle-graalvm-ceee

Once GraalVM is installed, to build the native executable, run:

```
mvn clean install -Pnative
```

The native build process can take a few minutes to complete, much slower than regular Java compilation, depending on the number of classes in the application, and the number of external libraries included.

Once complete, there will be a *-runner* executable in the */target* directory, which is the result of the native executable build process. The native executable will be specific to the operating system it was built on, as GraalVM uses native libraries to implement certain functionality.

> **TIP** **To create a native executable that is suitable for use within a Linux container, run `mvn package -Pnative -Dquarkus.native.container-build=true`.**

Try running the native executable version of the *Account Service*:

```
./target/chapter2-account-service-1.0.0-SNAPSHOT-runner
```

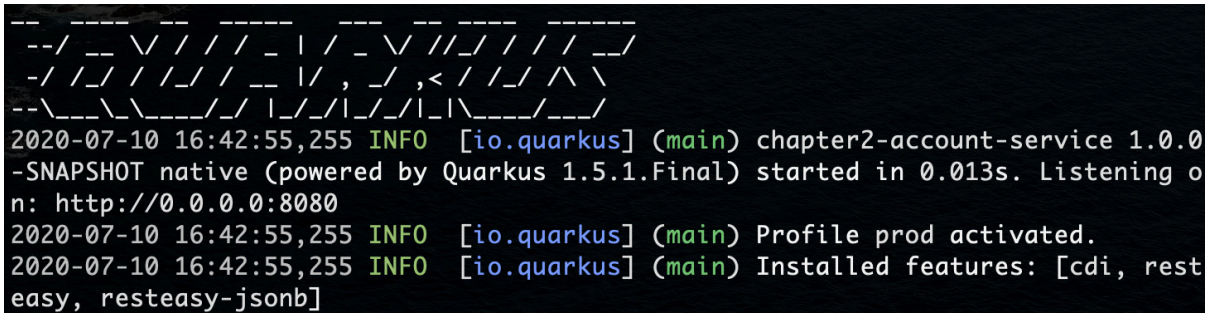


Figure 2.12 Quarkus native executable startup [TODO update version]

As with the earlier startup, *Figure 2.12* contains the console output when the native executable

starts. Notice the startup time for the *Account Service*? In this case it was only **0.013s**!

> **NOTE** **Within a native executable there is still garbage collection, though it uses different garbage collectors than the JVM. One impact of this is very long-running processes will see better memory performance over time with the JVM instead of native executable, due to the JVM continually optimizing memory utilization.**

In addition to the native executable build, it is now also possible to run native executable tests, as was seen with the generated project earlier. To run the current test with a native executable, create the test as follows:

Listing 2.19 Account

```
@NativeImageTest
public class NativeAccountResourceIT extends AccountResourceTest {
  // Execute the same tests but in native mode.
}
```

`mvn clean install -Pnative` will do the native executable build as before, but also run the above tests against that generated executable. If everything works as expected, the native executable will build, and the tests defined in `AccountResourceTest` will execute and all pass.

2.5 Running in Kubernetes

Quarkus focuses on Kubernetes-native, so it's time to put that to the test, packaging and deploying the *Account Service* to Kubernetes. There are several options when it comes to deploying Quarkus applications to Kubernetes, this section will cover some of them.

2.5.1 Generating Kubernetes YAML

When using Kubernetes everything is YAML, there's just no way around that. However, Quarkus provides some ways to alleviate the hassle of hand-crafting YAML, by offering extensions to generate it.

The first thing to do is add a dependency into the *Account Service* `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-kubernetes</artifactId>
</dependency>
```

The above dependency adds the Kubernetes extension for Quarkus, which offers the ability to generate, and customize, the necessary YAML for deploying to Kubernetes.

To see what it produces, run `mvn clean install` on the project, then look at the files produced in */target/kubernetes*. By default, it will produce a `.yml` and `.json` version of the required

configuration.

Here is an example of what can be seen for the *Account Service*:

Listing 2.20 kubernetes.yml

```
---
apiVersion: "v1"
kind: "Service"                                    ❶
metadata:
  annotations:
    app.quarkus.io/build-timestamp: "...."
    app.quarkus.io/commit-id: "...."
  labels:
    app.kubernetes.io/name: "chapter2-account-service"
    app.kubernetes.io/version: "1.0.0-SNAPSHOT"
  name: "chapter2-account-service"
spec:
  ports:
  - name: "http"                                   ❷
    port: 8080
    targetPort: 8080
  selector:
    app.kubernetes.io/name: "chapter2-account-service"
    app.kubernetes.io/version: "1.0.0-SNAPSHOT"
  type: "ClusterIP"
---
apiVersion: "apps/v1"
kind: "Deployment"                                 ❸
metadata:
  annotations:
    app.quarkus.io/build-timestamp: "...."
    app.quarkus.io/commit-id: "...."
  labels:
    app.kubernetes.io/name: "chapter2-account-service"
    app.kubernetes.io/version: "1.0.0-SNAPSHOT"
  name: "chapter2-account-service"
spec:
  replicas: 1                                      ❹
  selector:
    matchLabels:
      app.kubernetes.io/name: "chapter2-account-service"
      app.kubernetes.io/version: "1.0.0-SNAPSHOT"
  template:
    metadata:
      annotations:
        app.quarkus.io/build-timestamp: "...."
        app.quarkus.io/commit-id: "...."
      labels:
        app.kubernetes.io/name: "chapter2-account-service"
        app.kubernetes.io/version: "1.0.0-SNAPSHOT"
    spec:
      containers:
      - env:
        - name: "KUBERNETES_NAMESPACE"
          valueFrom:
            fieldRef:
              fieldPath: "metadata.namespace"
        image: "{docker-user}/chapter2-account-service:1.0.0-SNAPSHOT"       ❺
        imagePullPolicy: "Always"
        name: "chapter2-account-service"
        ports:
        - containerPort: 8080
          name: "http"
          protocol: "TCP"
```

❶ Defines the Kubernetes *Service*, *Account Service*, to be provisioned.

❷ Indicates the *Service* will expose port `8080`, and the application will be running on `8080`.

❸ Creates the Kubernetes *Deployment* of the *Service*.

❹ Tells Kubernetes to only create one instance. It's possible to set the value higher, but it's not necessary in this situation.

❺ Name of the Docker image to use for the *Deployment*.

With the default `kubernetes.yml` there are some customizations worth making:

- Change the name of the *Service* to `account-service`.
- Use a more meaningful name for the Docker image.

To make the above changes, modify `application.properties` in *src/main/resources* to include:

```
quarkus.container-image.group=quarkus-mp
quarkus.container-image.name=account-service
quarkus.kubernetes.name=account-service
```

Running `mvn clean install` again and looking at `kubernetes.yml` in */target/kubernetes*, notice that the name used is now `account-service`, and the Docker image is `quarkus-mp/account-service:1.0.0-SNAPSHOT`.

With Minikube as the deployment target, specific resource files can be generated. These resource files are required to expose the Kubernetes services to the local machine. Add the following dependency into the `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-minikube</artifactId>
</dependency>
```

TIP **Full details on how to deploy to Minikube can be found here: https://quarkus.io/guides/deploying-to-kubernetes#deploying-to-minikube**

Running `mvn clean install` will now generate Minikube specific resources into the *target/kubernetes* directory. Taking a look at the files, they're virtually identical. The only difference is with the `Service` definition:

```
spec:
  ports:
  - name: http
    nodePort: 30704                                    ❶
    port: 8080
    targetPort: 8080
```

```
  selector:
    app.kubernetes.io/name: account-service
    app.kubernetes.io/version: 1.0.0-SNAPSHOT
  type: NodePort                                    ❷
```

❶ For Kubernetes `nodePort` is not required, but when using Minikube the `nodePort` indicates which port on the local machine any traffic received on will be forwarded to the service.

❷ With Kubernetes the `type` is set to `ClusterIP`, but for Minikube `NodePort` is required.

IMPORTANT **It is not recommended to use Minikube specific Kubernetes resources when deploying to a Kubernetes environment for production. The examples will use the dependency, as it exposes the services to localhost.**

2.5.2 Packaging an application

With Quarkus there are several ways to package an application for deployment to Kubernetes:

- Jib (https://github.com/GoogleContainerTools/jib)
- Docker
- S2I (Source to Image) binary build

Each requires the addition of their respective dependency to the `pom.xml`. Either `quarkus-container-image-jib`, `quarkus-container-image-docker`, or `quarkus-container-image-s2i`.

To minimize the required dependencies for running the examples, Docker will not be required. The advantage with *jib* is that all requirements for producing container images is part of the dependency itself.

Container images with Docker utilize the contents of the *src/main/docker* directory, but require the Docker daemon to be installed.

With the dependency added, `quarkus-container-image-jib`, running the following will create the container image for JVM execution:

```
mvn clean package -Dquarkus.container-image.build=true
```

IMPORTANT **If there isn't a Docker daemon running locally, the container image creation will fail. The Docker daemon inside Minikube can be used instead. Run `minikube start`, and then expose the Minikube Docker daemon with `eval $(minikube -p minikube docker-env)`. It's necessary for the `eval` command to be run in each terminal window running the Maven commands to create a container, as the evaluation is specific to each terminal window.**

When successful, running `docker images` will show the `quarkus-mp:account-service` image:

```
→ docker images
REPOSITORY                          TAG                 IMAGE ID
  CREATED             SIZE
quarkus-mp/account-service          1.0.0-SNAPSHOT      62e48b07d6c0
  4 seconds ago       200MB
```

Figure 2.13 Docker images

2.5.3 Deploying and running an application

It's time to deploy to Minikube! If Minikube isn't already installed and running install Minikube using the instructions provided in Appendix A.

Once installed, open a new terminal window and run:

```
minikube start
```

WARNING **If it's the first time Minikube has been run, it could take some time to download the necessary container images.**

This will start Minikube with the default settings of 4GB RAM and 20GB HDD.

IMPORTANT **Run `eval $(minikube -p minikube docker-env)` in each terminal window that will be executing commands to build and deploy containers.**

Time to deploy! Run the following:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

The above command will generate the necessary container image, utilizing whichever container extension is installed, and deploy to the Kubernetes cluster specified in *.kube/config*. The Minikube cluster will be present in */[HOME]/.kube/config* if `minikube start` was executed.

If successful, the build should finish with messages similar to:

```
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Deploying to kubernetes
server: https://192.168.64.2:8443/ in namespace: default.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Service account-service.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Deployment account-service.
```

The log messages indicate each of the Kubernetes resources that were deployed, that were present within the `kubernetes.yml` generated earlier:

- *Service*
- *Deployment*

With *Account Service* deployed, run `minikube service list` to see the details of all services:

```
|-------------|-----------------|-------------|--------------------------|
|  NAMESPACE  |      NAME       | TARGET PORT |            URL           |
|-------------|-----------------|-------------|--------------------------|
| default     | account-service | http/8080   | http://192.168.64.2:30704 |
| default     | kubernetes      | No node port |
| kube-system | kube-dns        | No node port |
|-------------|-----------------|-------------|--------------------------|
```

For `account-service`, the URL for use locally is *http://192.168.64.2:30704*.

NOTE **As Minikube binds to the IP address of the machine, using http://localhost:30704 will not access the service in Minikube.**

To see the list of all accounts, open a browser to *http://192.168.64.2:30704/accounts*. Test out the other endpoints *Account Service* has to make sure they work as expected when deployed to Minikube.

That is a lot of information to digest. Let's recap the key tasks we covered during the chapter: how to generate a Quarkus project from https://code.quarkus.io/, using *Live Coding* to improve development speed, writing tests for a Quarkus microservice, building native executables to reduce image size and improve startup speed, and what's needed to deploy a Quarkus microservice to Kubernetes.

2.6 Summary

- You can open up https://code.quarkus.io/ in a browser and select the desired extensions for an application, choosing the name of the application, before generating the project code to download.
- Start a microservice with `mvn quarkus:dev` to begin *Live Coding* with Quarkus. Make changes to a JAX-RS resource in the IDE and see immediate changes to the running application when refreshing the browser.
- Add `@QuarkusTest` on a test class so that Quarkus packages the application for a test in the same manner as the Quarkus Maven plugin. This makes the test as near to an actual build as possible, improving the chances of catching any issues within a test early.
- Generate a *native executable* of a Quarkus application with `mvn clean install -Pnative`, with the "native" profile in `pom.xml`. The generated executable can optimize memory usage and startup time in constrained or FaaS type environments, where services aren't necessarily running for weeks on end.
- Kubernetes needs resource definitions to know what is being deployed. When adding the kubernetes extension to a Quarkus application, it automatically creates the JSON, and YAML, needed to deploy it to Kubernetes.
- Add `quarkus-container-image-jib` dependency to `pom.xml` for generating the necessary container images for deployment to Kubernetes. Running `mvn clean package -Dquarkus.container-image.build=true` will generate the image for Kubernetes.

3 Configuring Microservices

This chapter covers:

- Externalized configuration
- MicroProfile Config
- Accessing application configuration
- Configuration sources
- Quarkus configuration features
- Using Kubernetes ConfigMaps and Secrets

Chapter 2 introduced the *Account Service*, which ran both locally and in Kubernetes. The *Account Service* already runs in two contexts, locally and in Kubernetes, and can run in many more than two as shown below. Each context varies, having external services like a database, messaging system, and backend business microservices. The *Account Service* has to interact with the services in that same context, each with configuration requirements.

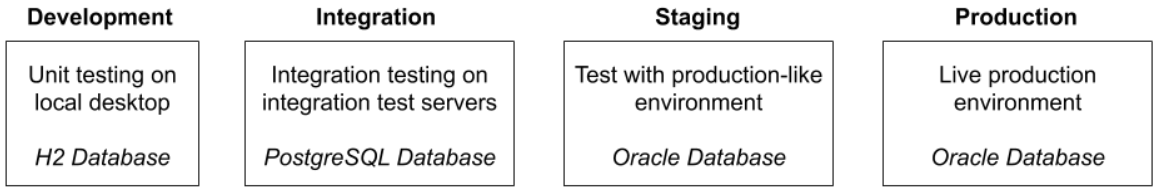


Figure 3.1 Example microservice contexts

Figure 3.1 represents how enterprises may use different databases depending on the context. The developer uses a local desktop database during development, like the H2 embedded database. Integration testing uses a low-cost database like PostgreSQL. Production uses a large-scale enterprise-grade database like Oracle, and staging mimics production as closely as possible, so it

is also using Oracle. The application needs a way to access and apply a configuration specific to each context without having to recompile, repackage, and redeploy for each context. What is required is *externalized configuration*, where the application accesses the configuration specific to the context it is running in.

3.1 MicroProfile Config Architecture Overview

MicroProfile Config enables externalized configuration, where the application can access and apply configuration properties without having to modify the application when a configuration changes. Quarkus also uses MicroProfile Config to configure itself and receives the same context-specific configuration benefits. For example, locally Quarkus may need to expose a web application on port 8080, while in staging and production Quarkus may need to be configured for port 80 without modifying application code.

Figure 3.2 outlines the MicroProfile Config architecture that enables externalized configuration.

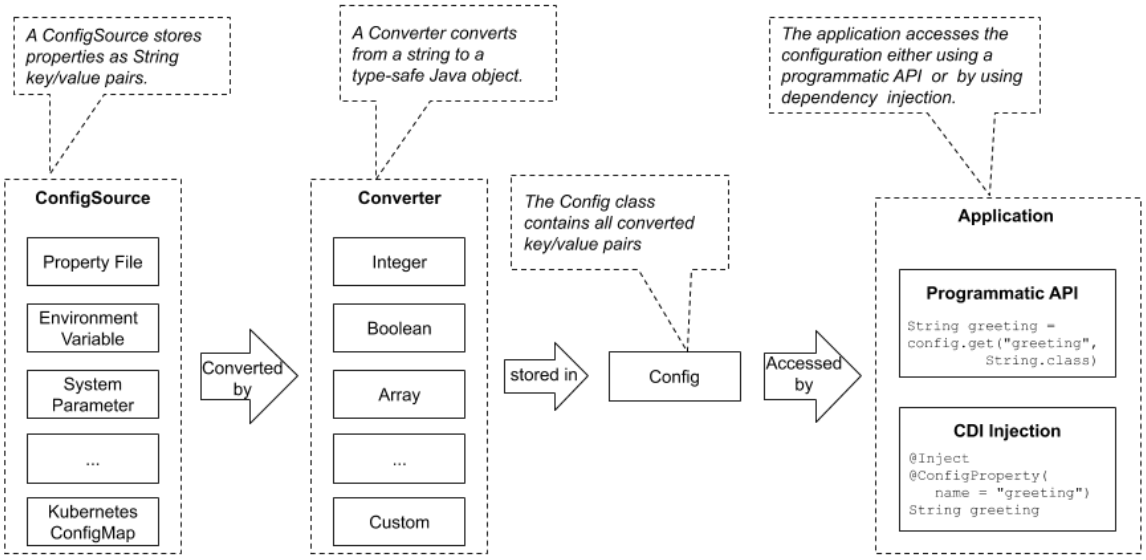


Figure 3.2 MicroProfile Config architecture

Properties are String key/value pairs defined in a configuration source. When the application starts, Quarkus uses MicroProfile Config to load properties from all available configuration sources. As the properties are loaded, they are converted from Strings to a Java data type and stored in a Config object. An application can then access the properties from the Config object using either a programmatic API or an annotation-based CDI injection API.

This section offers an overview of the MicroProfile Config architecture, and the remainder of this chapter details the components of this architecture most often used by developers. There are some advanced capabilities offered by MicroProfile Config like creating Converters for custom data types and building custom data sources, but these are beyond the scope of this book.

3.2 Accessing a configuration

Applications access a configuration through the `Config` object. MicroProfile Config includes two API styles for accessing the `Config` object. The following examples show the two API styles by retrieving the value of the *greeting* property from the Config object and storing it in a `greeting` variable.

- **Programmatic API.** The programmatic API is available for runtimes that do not have CDI injection available. Listing 3.1 shows a brief example:

Listing 3.1 Programmatic API

```
Config = ConfigProvider.getConfig();

String greeting = config.getValue("myapp.greeting", String.class);   ❶
```

❶ Directly look up the greeting using `config.getValue()` programmatic API.

- **CDI Injection.** Available to runtimes that support CDI injection. Because Quarkus supports CDI injection (see Listing 3.2), future examples will focus exclusively on CDI injection.

Listing 3.2 CDI Injection API example

```
@Inject
@ConfigProperty(name="myapp.greeting")   ❶
String greeting;
```

❶ Inject the value of `myapp.greeting` into `greeting` using CDI Injection

> **NOTE** **When injecting a property with @ConfigProperty, the MicroProfile Config specification requires the use of the @Inject annotation. Quarkus, with its focus on developer joy, makes the use of @Inject on @ConfigProperty annotations optional to simplify code. The remainder of this chapter will use the CDI approach exclusively.**

3.3 The Bank Service

With a background in externalized configuration and MicroProfile Config in place, the next step is to apply them. Let's begin by creating a microservice, the *Bank Service*, that uses the configuration APIs. The *Bank Service* is very basic, allowing the focus to remain on its configuration. It will consist of a few configurable fields that are accessed using MicroProfile Config and exposed through REST endpoints.

- **name**. String field containing the name of the bank.
- **mobileBanking**. Boolean indicating support for mobile banking.
- **supportConfig**. Java object with multiple configuration values for obtaining bank

support.

In later chapters, we will extend the *Bank Service* with additional capabilities, including those that extend across to the *Account Service* like invoking remote REST endpoints, propagating security tokens, and tracing requests.

3.3.1 Create Bank Service

Chapter 2 used `code.quarkus.io` to generate an application. The Quarkus maven plugin will be used here as an alternative approach. See Listing 3.3 for the maven command line to create the *Bank Service* Quarkus project.

Listing 3.3 Generate bank-service using maven

```
mvn io.quarkus:quarkus-maven-plugin:1.11.3.Final:create \     ❶
    -DprojectGroupId=quarkus \
    -DprojectArtifactId=bank-service \
    -DclassName="quarkus.bank.BankResource" \                 ❷
    -Dpath="/bank" \                                          ❸
    -Dextensions="resteasy-jsonb, \                           ❹
                  quarkus-kubernetes, \                       ❺
                  docker, \                                   ❻
                  minikube, \                                 ❼
                  kubernetes-config"                          ❽
cd bank-service
```

❶ Use the Quarkus maven plugin *create* goal to generate the project

❷ Name the Java class that implements a REST resource

❸ Specify the generated REST resource path

❹ Specify the list of Quarkus Extensions used in this chapter. To improve the developer experience, the `-Dextensions` property accepts shortened extension names. The shortened name must be specific enough to select only one extension or the command will fail. The quarkus-resteasy-jsonb extension, selected with the shortened "resteasy-jsonb" name adds JSON-B serialization support to RESTEasy (JAX-RS)

❺ The quarkus-kubernetes extension adds support for kubernetes deployment and Kubernetes YAML generation customizable through configuration properties.

❻ The quarkus-container-image-docker extension generates a container image using a docker registry

❼ The quarkus-minikube extension customizes Kuberntes YAML generation for minikube deployment

❽ The quarkus-kubernetes-config extension reads Kubernetes ConfigMaps and Secrets directly through the Kubernetes API server

TIP	**Extensions can be easily added at any time using the `add-extensions` maven goal and removed using the `remove-extension` goal. If running in development mode (`mvn compile quarkus:dev`), Quarkus will automatically reload the application with the extension changes included!** [6]

To prepare for this chapter's examples, execute the following steps:

- Remove the src/test directory and its subdirectories. This example will frequently break the generated tests by intentionally modifying the output.
- To prevent potential port conflicts, stop the *Account Service* started in the previous chapter if it is still running.

With the project created and prerequisite steps taken, start the application in developer mode with the command line shown in Listing 3.4:

Listing 3.4 Start Live Coding

```
mvn compile quarkus:dev
```

With developer mode enabled, it's time to start configuring the *Bank Service*.

3.3.2 Configuring the Bank Service *name* field

Beginning with the `bank.name` property, add the `getName()` method in *BankResource.java* shown in Listing 3.5:

Listing 3.5 Inject and use bank name property

```
@ConfigProperty(name="bank.name")                    ❶
String name;

@GET
@Path("/name")
@Produces(MediaType.TEXT_PLAIN)                      ❷
public String getName() {
    return name;                                     ❸
}
```

❶ Inject the value of the `bank.name` property into `name`

❷ The retun value will be in text format

❸ Return the injected `name`.

Load the `http://localhost:8080/bank/name` endpoint and notice the error page similar to 3.3.

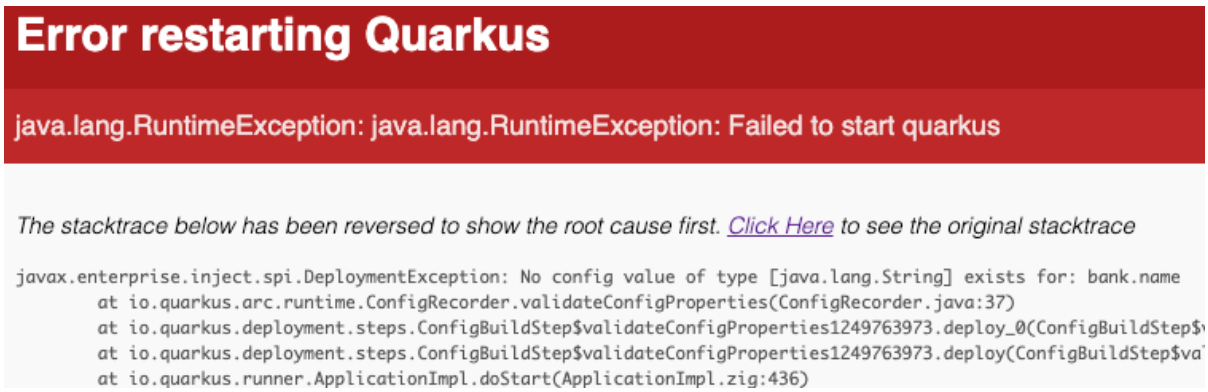


Figure 3.3 Browser output

The error identifies a shortcoming in the code, and Quarkus places the source of the error immediately at the top of the page. The code attempts to inject the value of the `bank.name` property, but `bank.name` has not been defined. Quarkus, as required by the MicroProfile Config specification, will throw a DeploymentException when attempting to inject an undefined property.

There are three ways to address missing property values, and all are commonly used depending on the need:

1. **Default value**. A fallback value that is general enough to apply in all situations when a property is missing
2. **Supply a value** Define the property and value within a property source
3. **Java Optional**. Use when a missing property value needs to be supplied by custom business logic.

Let's look at the first two in more detail, and the third shortly after that.

Assigning a default value is simple. Update the `@ConfigProperty` code as shown in Listing 3.6

Listing 3.6 Assigning a property a default value

```
@ConfigProperty(name="bank.name",
                defaultValue = "Bank of Default")   ❶
```

❶ Assign a default value, which is used when `bank.name` is undefined

Reloading the url will show the updated bank name in 3.7:

Listing 3.7 Output: Bank of Default

```
Bank of Default
```

The second option, assigning the property a value, can be easily accomplished by adding the `bank.name` property to the `application.properties` file as shown in 3.8:

Listing 3.8 Define bank.name property in application.properties

```
bank.name=Bank of Quarkus
```

Reloading the url will show the updated bank name in Listing 3.9:

Listing 3.9 Output: Bank of Quarkus

```
Bank of Quarkus
```

3.4 Configuration Sources

A configuration source is a source of configuration values defined as key/value pairs. `application.properties` is a configuration source, and Quarkus supports nearly a dozen more. It is common for a microservice to consume its configuration from more than one source. Figure 3.4 shows configuration sources and sample values used throughout this chapter.

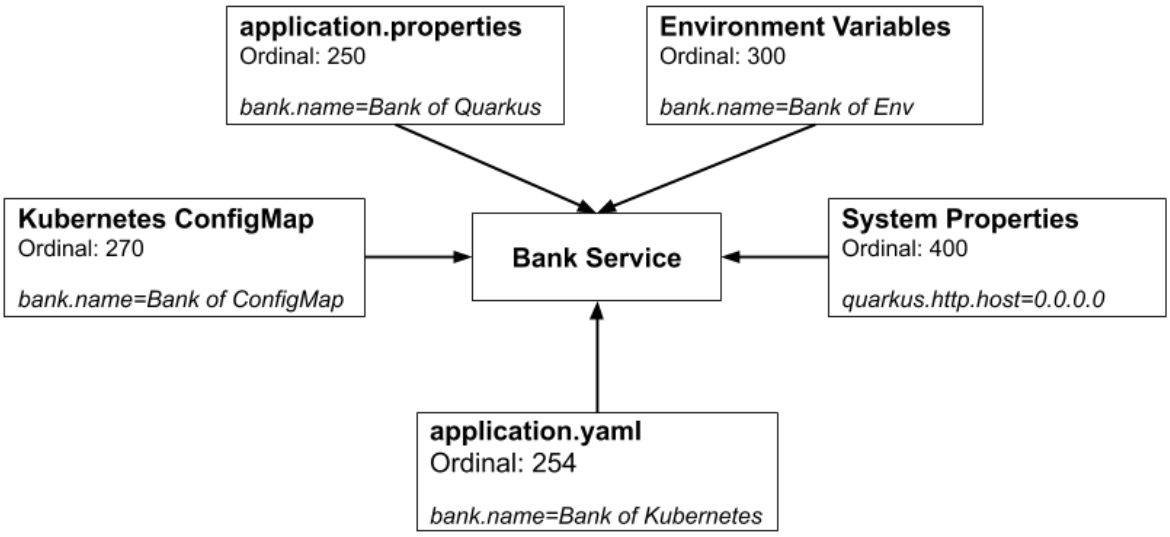


Figure 3.4 Configuration sources

The same property is often intentionally defined in more than one configuration source. If this is the case, which one takes precedence? MicroProfile Config uses a simple but effective approach for property conflict resolution. Each configuration source is assigned an *ordinal*. The properties defined in a configuration source with a higher ordinal take precedence over properties defined in a configuration source with a lower ordinal. MicroProfile Config requires support for three configuration sources, each with its own ordinal. Table 3.1 outlines the required MicroProfile Config configuration sources and additional Quarkus-supported configuration sources used in this chapter and their ordinals.

Table 3.1 Example MicroProfile Config sources

Source	Ordinal	Description
System properties	400	Required by MicroProfile Config. These are JVM properties that override nearly all property sources by using -Dproperty=value on the Java command line
Environment variables	300	Required by MicroProfile Config. Overrides most property settings. Linux containers use environment variables as a form of parameter passing
Kubernetes ConfigMap client	270	Directly access a Kubernetes ConfigMap. Overrides values in application.properties
application.yaml	254	Store properties in YAML format in files with a .yaml or .yml file extension
application.properties	250	The default property file used by most Quarkus applications
microprofile-config.properties	240	Required by MicroProfile Config. Useful for MicroProfile-centric applications that prefer application portability across MicroProfile implementations.

NOTE MicroProfile Config requires support for META-INF/microprofile-config.properties file for application portability. Quarkus supports microprofile-config.properties, but defaults to application.properties. This book uses application.properties, although microprofile-config.properties works equally well.

Let's put the configuration source ordinal values to the test and start with environment variables. Environment variables are a special case. Property names can contain dots, dashes, and forward slash characters, but some operating systems do not support them in environment variables. For this reason, these characters are mapped to characters that are broadly supported by operating systems.

MicroProfile Config searches for environment variables in the following order (Ex: `bank.mobileBanking`):

1. **Exact Match.** Search for `bank.mobileBanking`. If not found, move to next rule.
2. **Replace each non alpha-numeric with _.** Search for `bank_mobileBanking`. If not found, move to the next rule.
3. **Replace each non alpha-numberic with _ ; convert to upper-case.** Search for `BANK_MOBILEBANKING`

Define a BANK_NAME environment variable as seen in Listing 3.10:

Listing 3.10 Define bank.name environment variable

```
export BANK_NAME="Bank of Env"
```

Start Quarkus in developer mode (`mvn quarkus:dev`) to verify that the environment variable

overrides application.properties. Reloading the `http://localhost:8080/bank/name` URL will result in the output in Listing 3.11.

Listing 3.11 Output: Bank of Env

```
Bank of Env
```

Next, start Quarkus as a runnable jar to test two outcomes at once. The first is to test the system property configuration source, and the second is to test externalized configuration with a different packaging format.

Restart the application with the system property as seen in Listings 3.12 and 3.13:

Listing 3.12 Run the Bank Service as a runnable jar file

```
mvn -Dquarkus.package.type=uber-jar package   ❶

java "-Dbank.name=Bank of System" \
     -jar target/bank-service-1.0-SNAPSHOT-runner.jar   ❷
```

❶ Package the application into a runnable uber jar. Only the jar file and the JVM are needed to run the application.

❷ Run the application, specifying `bank.name` as a system property.

Listing 3.13 Startup Output

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2020-07-27 14:27:25,976 INFO  [io.quarkus] (main) bank-service 1.0-SNAPSHOT on JVM
(powered by Quarkus 1.6.1.Final) started in 0.587s. Listening on: http://0.0.0.0:8080   ❶
2020-07-27 14:27:25,993 INFO  [io.quarkus] (main) Profile prod activated.
2020-07-27 14:27:25,994 INFO  [io.quarkus] (main) Installed features: [cdi, resteasy]
```

❶ Quarkus running as an uber jar started in 0.587 seconds!

Reload the `http://localhost:8080/bank/name` endpoint, with output shown in Listing 3.14.

Listing 3.14 Output: Bank of System

```
Bank of System
```

There are a couple of items to point out in this approach. First, Quarkus applications can run as a uber jar file like many popular Java runtimes. Second, it started in just over .5 seconds! While uber jars have become a popular package format in recent years, it is not container-friendly. For this reason, Quarkus applications are rarely packaged as uber jars. More on this later.

Stop Quarkus and remove the BANK_NAME environment variable (see Listing 3.15)

Listing 3.15 Remove environment variable

```
unset BANK_NAME
```

3.5 Configuring the mobileBanking field

To begin coding mobileBanking configuration, start Quarkus in developer mode. To avoid exceptions, this example is using a different approach to the earlier `defaultValue` by introducing the use of the Java `Optional` type. Add the code in 3.16 to BankResource.java.

Listing 3.16 Add mobileBanking support to BankResource

```
@ConfigProperty(name="app.mobileBanking")      ❶
Optional<Boolean>  mobileBanking;              ❷

@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/mobilebanking")
public Boolean getMobileBanking() {
    return mobileBanking.orElse(false);        ❸
}
```

❶ Inject the value of `app.mobileBanking` into the `mobileBanking` field

❷ With Optional types, MicroProfile Config will not throw an exception if a propert is not defined

❸ If the mobileBanking field is undefined, return false

`mobileBanking` is a boolean, and properties are stored as strings. The string needs to be converted to a boolean data type for proper injection. As shown in Figure 3.2, converters in the MicroProfile Config architecture convert properties from strings to primitive data types, including booleans.

NOTE **There is also an API for creating converters for custom data types.**

MicroProfile Config supports the Java Optional data type for working with undefined properties while avoiding a DeploymentException. In the `getMobileBanking()` method, `mobileBanking` returns the configured value if defined, or *false* if left undefined.

To test the code, load the */bank/mobilebanking* endpoint to see an HTTP response of *false*, this time without the need for exception handling. The value of `app.mobileBanking` in `application.properties`, either *true* or *false*, will be returned value at the endpoint.

3.6 Grouping properties with @ConfigurationProperties

An alternative approach to individually injecting each property is to inject a group of related properties into fields of a single class. Annotating a class with @ConfigurationProperties, as shown in Listing 3.17, makes every field in the class a property. Every field will have its value injected from a property source.

Listing 3.17 Defining @ConfigurationProperties

```
@ConfigProperties                                          ❶
public class BankSupportConfig {                           ❷
    @Size(min=12, max = 12)   // xxx-xxx-xxxx format       ❸
    private String phone;

    public String email;                                   ❹

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

❶ Annotating a class with @ConfigProperties makes every field a property.

❷ A configuration class should be a Plain Old Java Object (POJO) with no business logic.

❸ Bean Validation [7] constraints are supported. With a `min` and `max` setting both set to 12, the phone number must contain twelve characters.

❹ Fields become properties regardless access modifiers. For example BankSupportConfig contains both private and public fields.

Listing 3.18 adds code to `BankResource.java` to inject the configuration and to return the injected property values at a JAX-RS endpoint.

Listing 3.18 Using @ConfigurationProperties

```
BankSupportConfig supportConfig;

public BankResource(BankSupportConfig config) {            ❶
    this.supportConfig = config;
}

@GET
@Produces(MediaType.APPLICATION_JSON)                      ❷
@Path("/support")
public Map<String, String> getSupport() {
    HashMap<String,String> map = new HashMap<>();

    map.put("email", supportConfig.email);                 ❸
    map.put("phone", supportConfig.getPhone());

    return map;
}
```

❶ Inject BankSupportConfig into BankingResource using constructor injection. Alternatively, directly inject using `@Inject BankSupportConfig supportConfig`.

❷ The return value (map) will be converted to a JSON representation.

❸ Add the properties to the map. `supportConfig.email` can be added directly because `email` is a public field, while `supportConfig.phone` is accessed through the `getPhone()` accessor method because `phone` is a private field. A best practice is to choose a consistent approach for better readability.

With the BankSupportConfig class and JAX-RS endpoint defined, the last step is defining the properties themselves. When using @ConfigProperties, the grouped properties share a common prefix. By default, the prefix is the name of the class. Table 3.2 shows the prefix conversion rules, using the `email` property as an example:

Table 3.2 @ConfigProperties property prefix rules

Class name	Resulting Property Name	Description
`Support`	**support.email**	**The prefix becomes the class name with a lower case leading letter**
`BankSupport`	**bank-support.email**	**The prefix becomes each camel-cased word, separated by a '-', with each word leading with a lower-case letter.**
`BankSupportConfig` **or BankSupportConfiguration`**	**bank-support.email**	**Same rules as the above** `BankSupport` **example, but drop the trailing Config or Configuration.**

With these rules in mind for the `BankSupportConfig`, Listing 3.19 shows the field properties are referenced as `bank-support.email` and `bank-support.phone` in `application.properties`.

Listing 3.19.

Listing 3.19 Defining support properties in application.properties

```
bank-support.email=support@bankofquarkus.com      ❶
bank-support.phone=555-555-5555
```

❶ With a class name of `BankSupportConfig`, the prefix becomes `bank-support`

When accessing the http://localhost:8080/bank/support REST endpoint, the result should look the same as in Listing 3.20:

Listing 3.20 Support endpoint JSON output

```
{"phone":"555-555-5555","email":"support@bankofquarkus.com"}
```

3.7 Quarkus-specific configuration features

The focus so far has been on featurees defined by the MicroProfile Config specification. Quarkus goes beyond the specification by adding Quarkus-specific configuration features.

3.7.1 Quarkus Configuration Profiles

With *profiles*, Quarkus enables multiple configurations within a single configuration source. Quarkus defines three built-in profiles:

- **dev.** Activated when in developer mode (ex: `mvn quarkus:dev`)
- **test.** Activated when running tests
- **prod.** Activated when not in development or test modes. Chapter 4 will use profiles to differentiate between production and development database configuration properties.

Listing 3.21 Example application.properties with profiles

```
bank.name=Bank of Quarkus                ❶
%dev.bank.name=Bank of Development       ❷
%prod.bank.name=Bank of Production       ❸
```

❶ This is the default property definition.

❷ This property definition is used when running quarkus in developer mode.

❸ This property definition is used whe application is startd with `java -jar` or when running a natively-compiled binary.

As shown in 3.21, the syntax for specifying a profile is `%profile.key=value`, so the application.properties file defines the `bank.name` property three times. When running Quarkus in development mode, like `mvn quarkus:dev`, the value of bank.name will be *Bank of Development*. When running in production, like `java -jar target/bank-service.jar`, the value of bank.name will be *Bank of Production*. `bank.name`, with no profile prefix, is a fallback value used when a profile value is not defined. For example, when running `mvn quarkus:test` in this example, neither %dev nor %prod properties apply. A `%test.bank.name` property is not defined. So, the fallback value of *Bank of Quarkus* is used.

Custom profiles can also be defined. Earlier in the chapter four contexts were covered - development, integration, staging, and production. Since Quarkus inherently supports development and production profiles, let's create a custom *staging* profile and update application.properties:

Listing 3.22 Add a staging profile `bank.name` property value

```
%staging.bank.name=Bank of Staging
```

Custom profiles can be activated by either setting the name of the `quarkus.profile` system property (ex: `java -Dquarkus.profile=staging -jar myapp.jar`) or by setting the

QUARKUS_PROFILE environment variable.

Start Quarkus in developer mode with `mvn compile quarkus:dev` and access the endpoint at `http://localhost:8080/bank/name`. The output is shown in Listing 3.23:

Listing 3.23 Quarkus developer mode output

```
Bank of Development
```

To see the production profile output, see Listings 3.24 and 3.25

Listing 3.24 Run application in production mode

```
java -jar target/banking-service-1.0-SNAPSHOT-runner.jar
```

Listing 3.25 Production mode output from `http://localhost:8080/bank/name`

```
Bank of Production
```

3.7.2 Property Expressions

Quarkus supports property expressions in application.properties, where an expression follows the `${my-expression}` format. Quarkus will resolve properties as it reads them. Let's modify Listing 3.19 to use property expressions:

Listing 3.26 Property Expression example

```
support.email=support@bankofquarkus.com      ❶
bank-support.email=${support.email}          ❷
bank-support.phone=555-555-5555
```

❶ Added `support.email` property for the support email address.

❷ Updated `bank-support.email` to use a property expression

Reload the */bank/support* endpoint to validate that the support email address matches 3.27.

Listing 3.27 Support endpoint JSON output

```
{"phone":"555-555-5555","email":"support@bankofquarkus.com"}
```

While `support.email` and `bank-support.${support-email}` are in the same configuration source in this example, they do not have to be. Chapter 4 will use property expressions for database credentials. The databases credentials and the property expression that refers to the credentials will be defined in different configuration sources.

3.7.3 Runtime vs build-time properties

As a MicroProfile Config implementation, Quarkus optimizes configuration for containers in general and Kubernetes in particular. Kubernetes is considered immutable infrastructure, where it restarts pods with a new application configuration instead of modifying an application's configuration within a running pod.

Let's do a quick Quarkus and traditional Java runtime configuration comparison. Most Java runtimes scan the classpath while an application is starting. The runtime scanning creates a very dynamic deployment capability at the cost of increased RAM utilization and increased startup time. It can take a significant amount of resources to conduct a classpath scan to build an in-memory model (meta-model) of what it has found. Also, the application pays this resource penalty every time it starts. In a highly dynamic environment like Kubernetes that encourages frequent incremental application updates, this is quite often!

Quarkus, on the other hand, considers its primary target environment being containers in general and Kubernetes in particular. Quarkus allows extensions to define two types of properties - build time and run time.

Quarkus pre-scans and compiles as much code as possible when the application is compiled (built), so it is static in nature when loaded and run. Build-time properties influence compilation, and how the metamodel (like annotation processing) is pre-wired. Changing build-time properties at runtime have no effect, like when running `java -jar myapp.jar`. Their values, or the effect of their values, are already compiled into `myapp.jar`. An example is a JDBC driver because developers typically know ahead of time which drivers will be required.

Runtime properties do not impact how code is pre-scanned and generated, but do influence runtime execution. Examples include port numbers like `quarkus.http.port=80` and database connection strings like `quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/mydatabase`.

The result of pre-scanning build-time is lower runtime memory utilization - consuming only 10's of MB of RAM - and faster startup time - starting in 10's of milliseconds as a native binary and 100's of milliseconds on the JVM.

Each Quarkus extension guide [8] lists its configurable properties. The *Quarkus - All Configuration Guide* [9] lists all configuration properties for all Quarkus extensions. In both cases, a lock icon identifies properties fixed at build time.

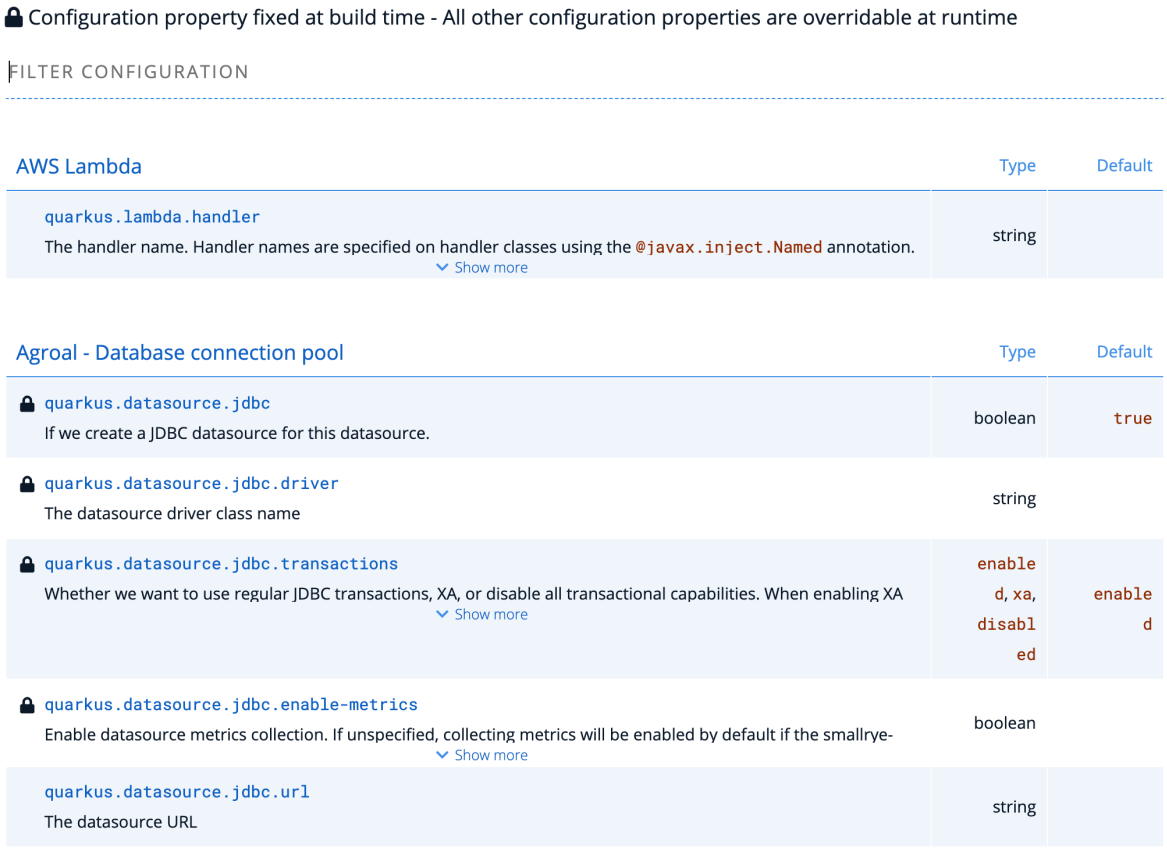


Figure 3.5 Buildtime properties identified by lock icon

Figure 3.5 shows a mix of fixed properties and runtime-configurable properties from the *Quarkus - All Configuration Guide*. For example, the Agroal database connection pooling extension "fixes" the `quarkus.datasource.jdbc.driver` property at build time, but allows the `quarkus.datasource.jdbc.url` propeerty to change after compilation.

3.8 Configuration on Kubernetes

We have been configuring the *Bank Service* throughout the chapter, and application configuration for a Kubernetes deployment is nearly the same. The primary difference is the available configuration sources and how to utilize them.

3.8.1 Common Kubernetes configuration sources

Table 3.1 covers configuration sources used in this chapter, but let's look at how they are most commonly used in Kubernetes.

- **System properties.** Container images often start a runtime with pre-defined parameters. A good example is requiring the use of team or corporate standards. The corporate standard in this case is using the JBoss LogManager:

```
java -Djava.util.logging.manager=org.jboss.logmanager.LogManager \
     -jar /deployment/app.jar
```

- **Environment variables.** A container is a self-contained runnable software package. Environment variables are a formalized and popular parameter-passing technique to configure an application packaged in a container. For example, the Postgres official container image uses environment variables like `POSTGRES_USER` to define a database user [10]. This approach to container parameter passing is popular in Kubernetes as well.
- **Kubernetes ConfigMap.** A ConfigMap is a first-class externalized configuration concept for Kubernetes. A ConfigMap stores non-confidential data as key-value pairs. Think of a ConfigMap as an interface for accessing key-value pairs, and there is more than one interface implementation. The most common implementation is mounting a ConfigMap as a storage volume within a Pod and is therefore accessible to all containers within the Pod. Quarkus uses a different ConfigMap implementation. Instead of mounting the configuration file within the container, the Quarkus ConfigMap extension takes a simpler approach by directly accessing the properties from Etcd using the Kubernetes REST-based API server. Figure 3.6 compares the two different approaches.

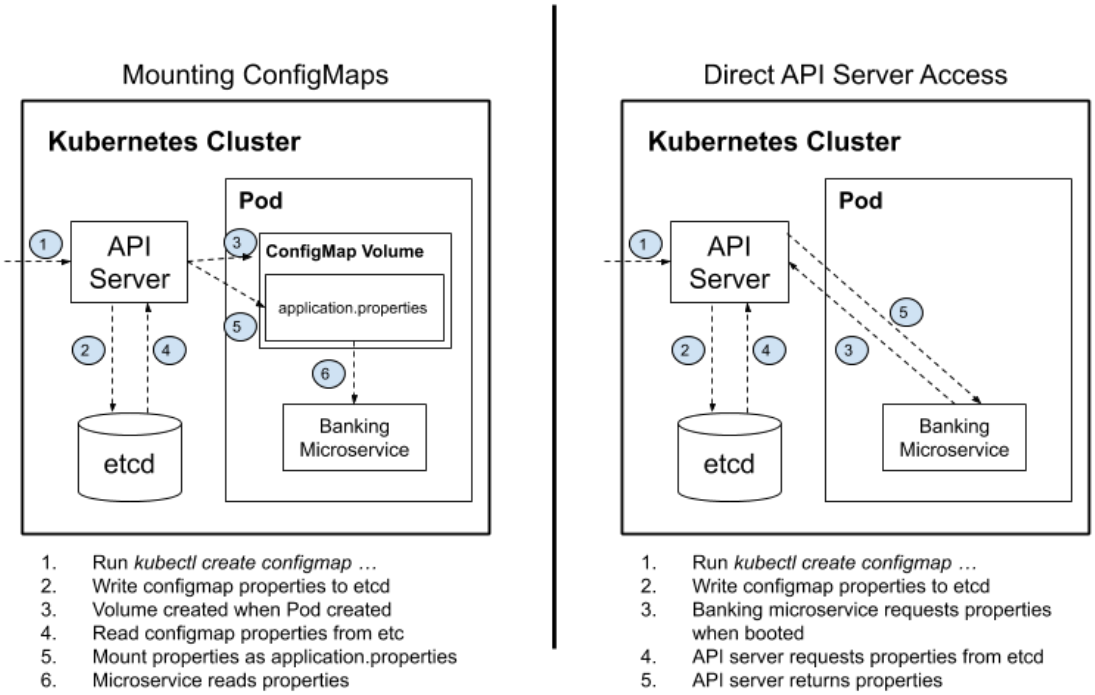


Figure 3.6 ConfigMaps - Mounting vs API Server direct access

- **application.properties.** Quarkus applications can still include an application.properties file for sensible default values.
- **3rd Party configuration sources.** Quarkus supports popular 3rd party configuration sources that can run in Kubernetes like the Spring Cloud Config Server, Vault, and Consul.

3.8.2 Using a ConfigMap for Quarkus applications

Quarkus can recognize ConfigMap files created from `application.properties`, `application.yaml`, and `application.yml` files. Let's create a ConfigMap out of an application.yaml file to not confuse it with the existing application.properties file. Create the application.yaml as seen in Listing 3.28 in the top-level project directory:

Listing 3.28 Create application.yaml

```
bank:
  name: Bank of ConfigMap
```

TIP **Make sure there are two spaces before the `name` property since YAML is space-sensitive.**

Next, create the Kubernetes ConfigMap as seen in Listing 3.29:

Listing 3.29 Create Kubernetes ConfigMap

```
kubectl create configmap banking \          ❶
        --from-file=application.yaml        ❷
```

❶ Create a ConfigMap named *banking*

❷ Populate the ConfigMap with the contents of the application.yaml file.

With the ConfigMap created in Kubernetes, the next step is to configure the banking service to access it, as shown in the application.properties file in Listing 3.30.

Listing 3.30 Configure Quarkus to use the banking ConfigMap

```
%prod.quarkus.kubernetes-config.enabled=true          ❶
%prod.quarkus.kubernetes-config.config-maps=banking   ❷
```

❶ Enable Kubernetes ConfigMap support. `%prod` specifies that it only applies when running in production.

❷ The comma-separated list of ConfigMaps to use

TIP **A ConfigMap can be viewed with `kubectl get cm/banking -oyaml` edited with `kubectl edit cm/banking`, and deleted with `kubectl delete cm/banking`.**

With the ConfigMap created and the banking service configured to use it, deploy the banking service to Kubernetes as shown in Listing 3.31:

Listing 3.31 Deploy updated application to Kubernetes

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

To verify the output, run `minikube service list` to obtain the base URL as in Listing 3.32

Listing 3.32 Example output of `minikube svc list`

```
|-------------|-----------------|--------------|---------------------------|
|  NAMESPACE  |      NAME       | TARGET PORT  |            URL            |
|-------------|-----------------|--------------|---------------------------|
| default     | banking-service | http/8080    | http://192.168.64.8:31763 |    ❶
| default     | kubernetes      | No node port |
| kube-system | kube-dns        | No node port |
|-------------|-----------------|--------------|---------------------------|
```

❶ The base URL, although the IP address and port will likely differ from what is shown

Load the URL in the browser, appending `/bank/name`. The full URL in this example would be http://192.168.64.8:31763/bank/name

The output should be the contents of bank.name defined in the ConfigMap as shown in Listing 3.33

Listing 3.33 Output obtained from ConfigMap

```
Bank of ConfigMap
```

3.8.3 Editing a ConfigMap

Changing a ConfigMap requires a Pod re-start. This boots a new *Bank Service* instance that reloads property values from its configuration sources. The first step, of course, is to edit the ConfigMap. Type `kubectl edit cm/banking`. See Listing 3.34 for editing the ConfigMap:

Listing 3.34 ConfigMap Contents while editing

```
apiVersion: v1
data:
  application.yaml: |-                          ❶
    bank:
      name: Bank of Quarkus (ConfigMap)         ❷
kind: ConfigMap
metadata:                                       ❸
  creationTimestamp: "2020-08-04T06:08:56Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:application.yaml: {}
    manager: kubectl
    operation: Update
    time: "2020-08-04T07:09:30Z"
  name: banking
  namespace: default
  resourceVersion: "863163"
  selfLink: /api/v1/namespaces/default/configmaps/banking
  uid: 3eba39df-336d-4a83-b50f-24ff8b767660
```

❶ Contents of application.yaml

❷ Edit this line to reflect new value of bank.name

❸ Ignore all the other content that is automatically added by Kubernetes when creating the ConfigMap. Do not modify content outside of application.yaml because the results will vary depending on the edits.

Kubernetes offers various ways to restart the pod, however the simplest is to redeploy the application as shown in Listing 3.35:

Listing 3.35 Redeploy updated application to Kubernetes

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

3.8.4 Kubernetes Secrets

ConfigMaps are ideal for general property storage and access. However, there are cases where working with confidential properties is required, like using usernames, passwords, and OAuth tokens. The Kubernetes' solution for storing sensitive information is the *Kubernetes Secret*. By default Secrets store data in base64-encoded format. While this makes sensitive data unreadable to the eye, it can be easily decoded. From an application perspective, secrets look and feel a lot like ConfigMaps.

WARNING **Like ConfigMaps, secrets are stored in Etcd. Any administrator with access to Etcd can decode base64-encoded secrets. Kubernetes can encrypt secret data at rest as well.** [11]

Up to this point, properties have been stored mostly in files including application.properties and application.yaml. ConfigMaps and Secrets can also store *literals*, meaning key-value pairs without having to define them within a file. See Listing 3.36 to create a database username and password using Secrets.

Listing 3.36 Create Kubernetes Secrets from literals

```
kubectl create secret generic db-credentials \     ❶
        --from-literal=username=admin \            ❷
        --from-literal=password=secret             ❸
```

❶ Create a Kubernetes Secret named `db-credentials`

❷ Store `password=secret` as a base64-encoded property

❸ Store `username=admin` as a base64-encoded property

Next, run the command in Listing 3.37 and view the output in Listing 3.38 to check it is encoded.

Listing 3.37 Get the Secret contents

```
kubectl get secret db-credentials -oyaml
```

Listing 3.38 Kubectl output

```
- apiVersion: v1
  data:
    password: c2VjcmV0    ❶
    username: YWRtaW4=    ❷
  kind: Secret
  metadata:
...
```

❶ Encoded password

❷ Encoded username

With a Kubernetes Secret containing `username` and `password` properties, the next step is to verify that these properties can be injected and used within the application. Extend BankResource as shown in Listing 3.39.

Listing 3.39 Access Secret from BankResource.java

```
@ConfigProperty(name="username")                    ❶
String  username;

@ConfigProperty(name="password")
String password;

@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/secrets")
public Map<String, String> getSecrets() {
    HashMap<String,String> map = new HashMap<>();

    map.put("username", username);                  ❷
    map.put("password", password);

    return map;
}
```

❶ Inject username and password into BankResource fields

❷ Insert username and password into a HashMap and return as a JSON string

Like a ConfigMap, applications need two Quarkus properties defined to access a Secret. See Listing 3.40.

Listing 3.40 Enable Secret access in application.properties

```
%prod.quarkus.kubernetes-config.secrets.enabled=true     ❶
%prod.quarkus.kubernetes-config.secrets=db-credentials   ❷
```

❶ Enable access to secrets. `%prod` specifies that it only applies when running in production.

❷ The comma-separated list of Secrets to include for property lookup

Redeploy the application and open the `/bank/secrets` endpoint. The output should look like Listing 3.41

Listing 3.41 Browser output

```
{"password":"secret","username":"admin"}
```

3.9 Summary

This chapter covers a lot of ground. It introduces externalized configuration, MicroProfile Config, Quarkus-specific configuration features and Kubernetes ConfigMaps. The top two take-away's from this chapter is that externalized configuration is a microservice deployment necessity and Quarkus uses MicroProfile Config and custom configuration features to make Kubernetes deployments both practical and seamless.

Here are key detailed points:

- Quarkus uses the MicroProfile Config API for application configuration and to configure itself.
- MicroProfile Config uses Configuration Sources to abstract wheres configuration values are stored.
- There is an order of precedence when loading property values from configuration sources.
- Properties can be loaded individually using @ConfigProperty or in bulk using @ConfigProperties
- Quarkus supports configuration profiles for loading context-dependent configuration values, like for development, test, and production.
- Not all Quarkus properties can be modified at runtime.
- Quarkus supports ConfigMaps by reading ConfigMap key/value pairs using the Kubernetes API server.
- Applications can store and access sensitive information in Kubernetes Secrets.

4 Database access with Panache

This chapter covers:

- What is Panache
- Simplifying JPA development with Panache
- Database testing with Panache and `@QuarkusTest`

In chapter 2, creating the *Account Service* showed how to develop JAX-RS endpoints with Quarkus. This chapter will take that *Account Service* and add database storage for the account data, instead of the data only being held in memory.

As most microservices will need to store some type of data, or interact with data stored by another microservice, being able to read and store data to a database is a key feature to learn and understand. Though stateless microservices are "a thing", and certainly a goal, if appropriate, for a microservice, there are also times when denying the need to store data leads to unnecessary mental gymnastics. Leading to a significantly more complex distributed system.

To simplify the development of microservices requiring storage, Quarkus created *Hibernate ORM with Panache (Panache)*. An opinionated means of storing and retrieving state from a database, heavily inspired from the Play framework, Ruby on Rails, and JPA experience. *Panache* offers two different paths, depending on developer preference, *Active Record* and *Data Repository*. The *Data Repository* approach will be familiar to those with experience in Spring Data JPA.

Before getting into how *Panache* can simplify database development in a microservice, the *Account Service* will be altered to store data with the known JPA approach. Showing how to store data with JPA will make it easier to compare the approaches, both in terms of the amount of code, and the different coding styles they facilitate.

4.1 Datasources

Before delving into modeling objects for persistence, there needs to be a datasource defined for JPA, or Panache, to communicate with the database.

The *Agroal* extension from Quarkus handles datasource configuration and setup. However, adding a dependency for the extension is unnecessary when being used for JPA or Panache, as those dependencies have a dependency on *Agroal*. At a minimum, the type of datasource and database URL must be specified in configuration, and usually a username and password:

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=database-user
quarkus.datasource.password=database-pwd
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/my_database
```

NOTE **The username and password values above are just examples, what they need to be set to will be dependent on the database being connected to.**

In this particular example, the configuration tells Quarkus that the application will be connecting to a PostgreSQL database. The JDBC configuration indicates the URL of the database.

With the above configuration, there is no datasource name mentioned. That is because the configuration is defining the *default* datasource that should be used by anything needing a JDBC Datasource. Multiple datasources are created by setting a specific name in the configuration. For instance, the below configuration creates a datasource called "orders":

```
quarkus.datasource.orders.db-kind=postgresql
quarkus.datasource.orders.username=order-user
quarkus.datasource.orders.password=order-pwd
quarkus.datasource.orders.jdbc.url=jdbc:postgresql://localhost:5432/orders_db
```

There are many kinds of databases that datasources can be created for, but the more popular ones are `h2` (mostly for testing), `mysql`, `mariadb`, and `postgresql`.

In addition to defining the datasource configuration, there needs to be a JDBC driver present for Quarkus to create the datasource, and to communicate with the database! For that, a dependency such as:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-jdbc-postgresql</artifactId>
</dependency>
```

The above dependency matches the configuration from earlier that specified the database type as `postgresql`. If a different database is used, the application would require a different dependency where the artifact is prefixed with `quarkus-jdbc-`, and suffixed with the database type name.

While it is possible to use the regular JDBC driver dependencies directly with Quarkus. Using the Quarkus provided JBDC driver extensions allows them to be automatically configured with Quarkus, but also means they are guaranteed to work as part of a native executable. At present most JDBC driver dependencies won't work inside a native executable.

Quarkus has a fantastic feature to help with testing when using a database. Adding `@QuarkusTestResource(H2DatabaseTestResource.class)` onto a test class will start an H2 database as part of the test startup! All it needs is a dependency:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-test-h2</artifactId>
  <scope>test</scope>
</dependency>
```

Most applications don't need to interact with a datasource directly, they use another layer on top to simplify the code. Now it's time to modify the *Account Service* from chapter 2 to use JPA to store its data instead of in memory.

4.2 JPA

Before delving into what a Quarkus microservice that uses JPA looks like, *Figure 4.1* shows the components involved and their interaction:

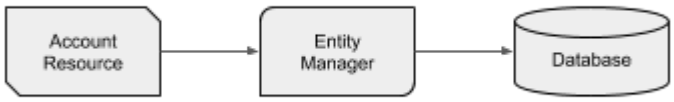


Figure 4.1 Account Service - JPA

Though JPA may not be the favored approach to database interactions by many developers, it provides an easy migration path for anyone familiar with Java EE and Jakarta EE development with JPA. In addition, it provides a good basis for comparison with the Panache approaches later in the chapter.

As seen in *Figure 4.1*, the `AccountResource` uses an `EntityManager` to interact with the database. Whether it's finding entities, creating new ones, or updating existing ones, it all happens through the `EntityManager` instance.

Let's begin converting the *Account Service* from chapter to use JPA for data storage. To add JPA to the *Account Service*, the following dependencies need to be added:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-hibernate-orm</artifactId>
</dependency>
```

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-jdbc-postgresql</artifactId>
</dependency>
```

`quarkus-hibernate-orm` adds the Hibernate implementation of JPA to the project, and `quarkus-jdbc-postgresql` adds the JDBC driver for PostgreSQL discussed in 4.1.

The updated code for the *Account Service* from chapter 2 can be found in the *chapter4/jpa/* directory of the book source.

Next is to modify the *Account* class to be a JPA Entity:

Listing 4.1 Account

```
@Entity                                                          ❶
@NamedQuery(name = "Accounts.findAll",                           ❷
  query = "SELECT a FROM Account a ORDER BY a.accountNumber")
@NamedQuery(name = "Accounts.findByAccountNumber",               ❸
  query = "SELECT a FROM Account a WHERE a.accountNumber = :accountNumber
  ORDER BY a.accountNumber")
public class Account {
  @Id                                                            ❹
  @SequenceGenerator(name = "accountsSequence", sequenceName = "accounts_id_seq",
    allocationSize = 1, initialValue = 10)                       ❺
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
    generator = "accountsSequence")                              ❻
  private Long id;

  private Long accountNumber;                                    ❼
  private Long customerNumber;
  private String customerName;
  private BigDecimal balance;
  private AccountStatus accountStatus = AccountStatus.OPEN;

  ...
}
```

❶ Indicates the Pojo is a JPA Entity.

❷ Define a named query to retrieve all accounts, and order the result by `accountNumber`.

❸ Another named query, this one finding accounts that match `accountNumber`.

❹ Tells JPA that the `id` field is the primary key of the database table.

❺ Create a sequence generator for the `id` field, starting with the number 10. Starting at 10 provides space to import some records on startup for testing.

❻ Uses the sequence generator from (5) to specify where the generated value comes from for the primary key.

❼ When using JPA, the fields can be marked `private` instead of `public`.

NOTE **Getter and setter methods, general object methods, and equals and hashcode methods present from chapter 2 are excluded from the listing for clarity.**

All constructors were removed from the `Account` class, because construction of instances directly is not needed when using JPA.

With the JPA Entity defined, it's now possible to use an `EntityManager` to interact with the database for that entity. The first change to `AccountResource` is to inject an instance of the `EntityManager`:

```
@Inject
EntityManager entityManager;
```

Now the `entityManager` instance can be used for retrieving all the accounts:

Listing 4.2 AccountResource

```
@GET
public List<Account> allAccounts() {
  return entityManager
    .createNamedQuery("Accounts.findAll", Account.class)          ❶
    .getResultList();                                             ❷
}
```

❶ Tells the `entityManager` to use the named query "Accounts.findAll" defined on `Account` in 4.1, and that the expected results will be of the `Account` type.

❷ Convert the results from the database into a `List` of `Account` instances.

There was another named query for finding accounts by their number on `Account`, let's see what that looks like:

Listing 4.3 AccountResource

```
public Account getAccount(@PathParam("acctNumber") Long accountNumber) {
  try {
    return entityManager
      .createNamedQuery("Accounts.findByAccountNumber", Account.class)      ❶
      .setParameter("accountNumber", accountNumber)                          ❷
      .getSingleResult();                                                    ❸
  } catch (NoResultException nre) {                                          ❹
    throw new WebApplicationException("Account with " + accountNumber + " does not exist.", 404);
  }
}
```

❶ Use the "Accounts.findByAccountNumber" named query.

❷ Pass the parameter into the query, setting the name of the parameter in the query and passing the value.

❸ For a given `accountNumber` there should only be one account, so request the return of a single `Account` instance.

❹ To retain the exception handling added in chapter 2, catch any `NoResultException` thrown when there is no account and convert it to a `WebApplicationException`.

Now for a look at how to add a record to the database with an `EntityManager`:

Listing 4.4 AccountResource

```
@Transactional                                          ❶
public Response createAccount(Account account) {
  ...
  entityManager.persist(account);                       ❷
  return Response.status(201).entity(account).build();
}
```

❶ Tells Quarkus that a transaction should be created for this operation. A transaction is necessary here because any exception from within the method needs to result in a "rollback" of any proposed database changes before they're committed. In this case it's a new `Account`.

❷ Call `persist` with the `Account` instance, adding it to the persistent context for committing to the database at the completion of the transaction, in this case `createAccount()`.

Having shown how to use named queries and persist a new entity instance, how do I update an entity that already exists? Calling `entityManager.persist()` will throw an exception if it's already persisted. What is used instead?

Listing 4.5 AccountResource

```
@Transactional                                          ❶
public Account withdrawal(@PathParam("accountNumber") Long accountNumber, String amount) {
  Account entity = getAccount(accountNumber);           ❷
  entity.withdrawFunds(new BigDecimal(amount));         ❸
  return entity;
}
```

❶ Require a transaction during method execution.

❷ Retrieve an `Account` instance using `accountNumber`.

❸ Withdraw the funds from the account, modifying the state of the entity.

For those that noticed, in *Listing <<account-resource-jpa-withdrawal* `entityManager` was not used. It wasn't necessary to call any methods on `entityManager` because retrieving the account instance had already happened. Retrieving the account puts the instance into the persistence context as a *managed* object. Managed objects can be updated at will and persisted in the database when the transaction commits.

If the method had a parameter of `Account`, instead of `accountNumber` and `amount`, the instance would be *unmanaged* as it does not exist in the current persistence context. Updating the balance would require:

```
@Transactional
public Account updateBalance(Account account) {
  entityManager.merge(account);                         ❶
```

```
  return account;
}
```

❶ Merges the unmanaged instance into the persistence context, making it managed.

IMPORTANT **When using unmanaged instances to update state in a database, it's necessary to ensure that state hasn't been updated in the meantime. For example, the above method updating the balance requires the account to have been retrieved previously. An update to the balance could have occurred in another request between retrieval of the account, and a call to update the balance. There are means to mitigate against this problem, such as versioning JPA entities, but the use of `entityManager.merge()` needs to be carefully considered.**

It's time to write some tests! To be able to test with an H2 database, but use PostgreSQL in a production deployment, configuration profiles need to be used. Here's a snippet of the needed `application.properties`:

```
quarkus.datasource.db-kind=postgresql                ❶
quarkus.datasource.username=quarkus_banking
quarkus.datasource.password=quarkus_banking
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost/quarkus_banking

%test.quarkus.datasource.db-kind=h2                  ❷
%test.quarkus.datasource.username=username-default
%test.quarkus.datasource.password=                   ❸
%test.quarkus.datasource.jdbc.url=jdbc:h2:tcp://localhost/mem:default
```

❶ Defines the datasource configuration for production, when building the application, and for *Live Coding*.

❷ Datasource configuration for tests.

❸ Overrides the password to empty, as H2 does not require a password.

`%test.` is one of the configuration profiles introduced in Chapter 3. Using the test profile for H2 configuration enables a separate configuration for production and *Live Coding* modes.

Here's a test using the H2 database described in 4.1:

Listing 4.6 AccountResourceTest

```
@QuarkusTest
@QuarkusTestResource(H2DatabaseTestResource.class)          ❶
@TestMethodOrder(OrderAnnotation.class)
public class AccountResourceTest {
  @Test
  @Order(1)
  void testRetrieveAll() {
    Response result =
        given()
          .when().get("/accounts")
          .then()
            .statusCode(200)
            .body(
                containsString("Debbie Hall"),
                containsString("David Tennant"),
                containsString("Alex Kingston")
            )
            .extract()
            .response();

    List<Account> accounts = result.jsonPath().getList("$");
    assertThat(accounts, not(empty()));
    assertThat(accounts, hasSize(8));
  }
}
```

❶ Tells Quarkus to start an H2 database prior to the tests being executed.

The above test may look familiar from chapter 2. The only change between the same test in chapter 2 and this one, is that `@QuarkusTestResource` was added to the test class. Another change is when verifying the customer names, why are they different? In chapter 2 all the data was in memory only, but now it's within a database.

To add records for testing, define an `import.sql` in the *chapter4/jpa/src/main/resources* directory:

```
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName, customerNumber)
VALUES (1, 123456789, 0, 550.78, 'Debbie Hall', 12345);
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName, customerNumber)
 VALUES (2, 111222333, 0, 2389.32, 'David Tennant', 112211);
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName, customerNumber)
 VALUES (3, 444666, 0, 3499.12, 'Billie Piper', 332233);
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName, customerNumber)
 VALUES (4, 87878787, 0, 890.54, 'Matt Smith', 444434);
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName, customerNumber)
 VALUES (5, 990880221, 0, 1298.34, 'Alex Kingston', 778877);
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName, customerNumber)
 VALUES (6, 987654321, 0, 781.82, 'Tom Baker', 908990);
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName, customerNumber)
 VALUES (7, 5465, 0, 239.33, 'Alex Trebek', 776868);
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName, customerNumber)
 VALUES (8, 78790, 0, 439.01, 'Vanna White', 444222);
```

Then inform Quarkus to use it by adding the below to `application.properties`:

```
quarkus.hibernate-orm.sql-load-script=import.sql
```

With all that in place, from the *chapter4/jpa/* directory, run the following:

```
mvn clean install
```

The test will execute using the H2 database, and if everything went well, the test passed!

In developing the *Account Service* to use JPA, there was no mention of a `persistence.xml` file, why is that? Everyone familiar with developing JPA code in Java EE and Jakarta EE will know about creating a `persistence.xml` file to configure the driver, datasource name, and other JPA configuration elements.

With Quarkus, there's no need for a `persistence.xml`. What is present in that file is either performed automatically based on dependencies, has sensible defaults, or can be customized with `application.properties` instead. Though it is possible to use a `persistence.xml` file with Quarkus, it will not be demonstrated.

As an exercise for the reader, add additional test methods for:

- Creating an account
- Closing an account
- Withdrawing funds from an account
- Depositing funds into an account

By no means were all aspects of how to use JPA covered in this section, that was not the intention. While using JPA with Quarkus is an option, the purpose of this section was to outline some key usages of JPA to provide a means of comparing how data access with Panache differs.

4.3 Simplifying database development

Using JPA for accessing a database is only one approach, and there are many. Quarkus also includes the ability to choose the *Active Record* or *Data Repository* approaches to managing state. Both of these approaches are part of the *Panache* extensions to Quarkus. Panache seeks to make writing entities trivial and fun with Quarkus.

Though the following sections talk about data, Panache is emerging as a mini brand within Quarkus for simplification. In addition to simplifying entity development, Panache also has experimental functionality for generating RESTful CRUD endpoints, saving the time it takes to churn out the boilerplate JAX-RS definitions. See https://quarkus.io/guides/rest-data-panache for all the details.

4.3.1 Active record approach

Let's take a look at how the *Active record* pattern differs to JPA:

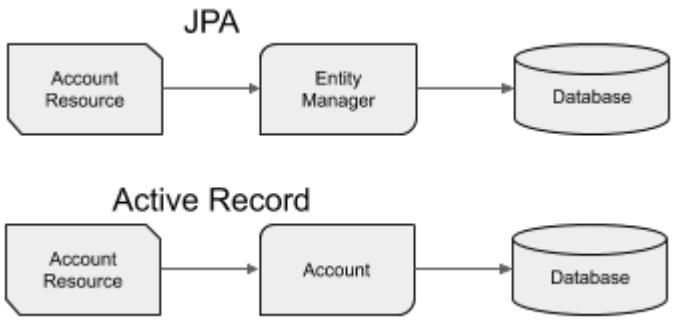


Figure 4.2 Account Service - Active Record

As seen in *Figure 4.2* above, all interactions occur through the entity itself. As objects usually hold data that needs to be stored, *Active record* puts the data access logic into the domain object directly. The *Active record* approach rose to popularity with Ruby on Rails and the Play framework.

In 2002 Martin Fowler outlined the approach in his book "Patterns of Enterprise Architecture". [12] His definition can be found on his site.[13]

Now to work on the implementation! All the code for this section can be found in the _/chapter4/active-record/ directory of the book source code.

Dependencies need to be different, as regular Hibernate is not what is needed, but the Panache version instead. For that, add the following dependency:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-hibernate-orm-panache</artifactId>
</dependency>
```

As a JDBC driver is needed, the same PostgreSQL dependency used with JPA can be added.

For the entity, that's different as Panache is used:

Listing 4.7 Account

```
@Entity
public class Account extends PanacheEntity {                    ❶
  public Long accountNumber;                                     ❷
  public Long customerNumber;
  public String customerName;
  public BigDecimal balance;
  public AccountStatus accountStatus = AccountStatus.OPEN;

  public static long totalAccountsForCustomer(Long customerNumber) {   ❸
    return find("customerNumber", customerNumber).count();
  }

  public static Account findByAccountNumber(Long accountNumber) {
    return find("accountNumber", accountNumber).firstResult();
  }
  ...
}
```

❶ `Account` extends `PanacheEntity`, which provides the data access helper methods like `persist()`.

❷ The fields on `Account` need to be public.

❸ Custom static methods can be added to enhance those provided with `PanacheEntity`.

NOTE **`equals()` and `hashCode()` methods excluded for brevity. The full code can be viewed in the chapter 4 book source code.**

There's a couple of key points to note with *Listing 4.7*:

- `@Entity` is still used to indicate the class is a JPA Entity.
- Getter and setter methods for the fields are not required. During build time Panache generates the necessary getter and setter methods, replacing field access in code to use the generated getter and setter methods.
- Definition of `id`, the primary key, is handled by `PanacheEntity`. If there was a need to customize the `id` configuration, it could be done with the usual JPA annotations.

Given the data access methods are present on `Account`, interacting with it must be quite different:

Listing 4.8 AccountResource

```
public class AccountResource {

  @GET
  public List<Account> allAccounts() {
    return Account.listAll();                                    ❶
  }

  @GET
  @Path("/{acctNumber}")
  public Account getAccount(@PathParam("acctNumber") Long accountNumber) {
    return Account.findByAccountNumber(accountNumber);           ❷
  }

  @POST
  @Transactional
  public Response createAccount(Account account) {
    account.persist();                                           ❸
    return Response.status(201).entity(account).build();
  }

  @PUT
  @Path("{accountNumber}/withdrawal")
  @Transactional
  public Account withdrawal(@PathParam("accountNumber") Long accountNumber, String amount) {
    Account entity = Account.findByAccountNumber(accountNumber);
    entity.withdrawFunds(new BigDecimal(amount));
    return entity;                                               ❹
  }
}
```

❶ Uses the static `listAll()` from `PanacheEntity` superclass of `Account` to retrieve all the accounts.

❷ Calls the custom static method from *Listing 4.7*, retrieving an `Account` instance by the `accountNumber`.

❸ Adds a new `Account` instance into the persistence context. On transaction commit the record will be added to the database.

❹ When modifying an existing instance, it will be persisted on transaction completion.

For testing, `AccountResourceTest` from the JPA example earlier can be copied for use with Active record. The only necessary changes are because `Account` no longer has methods for retrieving or setting values, it needs direct field usage.

As before, the tests will use an in memory H2 database, import data on startup from `import.sql`, and the `application-properties` doesn't need to change compared with the JPA version.

In the */chapter4/active-record/* directory run:

```
mvn clean install
```

If all went well, all the tests will pass.

To briefly recap, the *Active record* approach with Panache integrates all data access into the JPA Entity, while taking care of boilerplate such as defining the primary key. `PanacheEntity` provides simplified methods that don't require deep SQL knowledge to construct queries, enabling developers to focus on the necessary business logic.

4.3.2 Data Repository approach

Now on to the last approach, *Data Repository*:

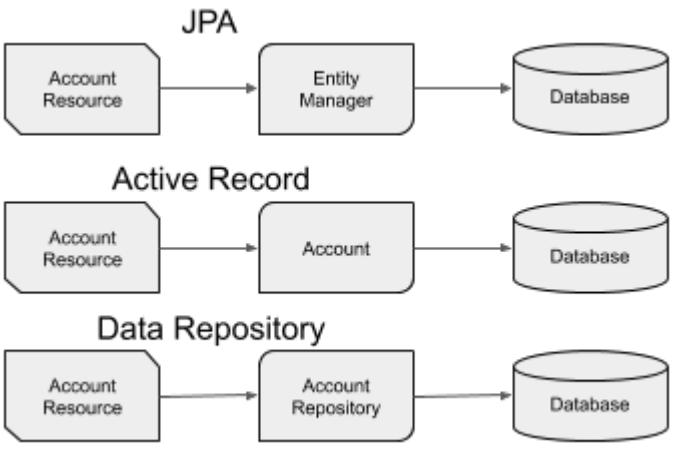


Figure 4.3 Account Service - Data Repository

Figure 4.3 uses `AccountRepository` as the intermediary for data access methods. There are some similarities with `EntityManager` from JPA, but also key differences. The Spring Framework popularized the *Data Repository* approach over the last decade or more.

Martin Fowler also outlined this approach, with *Active record*, in "Patterns of Enterprise Architecture".[14] On his website Martin Fowler explains the approach.[15]

So what's needed to implement the *Data Repository* approach?

Exactly the same as for the *Active Record* approach:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-hibernate-orm-panache</artifactId>
</dependency>
```

One benefit of both approaches being in the same dependency, it's quick to switch between either approach, or even use each approach in different situations in the same application.

The `Account` entity for the *Data Repository* approach:

Listing 4.9 Account

```
@Entity
public class Account {
  @Id
  @GeneratedValue
  private Long id;

  private Long accountNumber;
  private Long customerNumber;
  private String customerName;
  private BigDecimal balance;
  private AccountStatus accountStatus = AccountStatus.OPEN;

  ...
}
```

NOTE **Getter and setter methods, general object methods, and equals and hashcode methods are excluded from the listing for brevity.**

Listing 4.9 is very similar to *Listing 4.1* for JPA. The main difference is no `@NamedQuery` annotations, and the default id generation process for the primary key.

Time to take a look at the repository class:

Listing 4.10 AccountRepository

```
@ApplicationScoped                                                    ❶
public class AccountRepository implements PanacheRepository<Account> {  ❷
  public Account findByAccountNumber(Long accountNumber) {            ❸
    return find("accountNumber = ?1", accountNumber).firstResult();
  }
}
```

❶ `@ApplicationScoped` tells the container that only one instance should exist.

❷ Implement `PanacheRepository` for all the data access methods.

❸ Define a custom data access method.

As with the *Active Record* approach, a parent class includes the convenience methods for finding and retrieving instances.

But how different is the JAX-RS resource:

Listing 4.11 AccountResource

```
public class AccountResource {

  @Inject
  AccountRepository accountRepository;                ❶

  @GET
  public List<Account> allAccounts() {
    return accountRepository.listAll();               ❷
  }

  @GET
  @Path("/{acctNumber}")
  public Account getAccount(@PathParam("acctNumber") Long accountNumber) {
    Account account = accountRepository.findByAccountNumber(accountNumber);   ❸
    return account;
  }

  @POST
  @Transactional
  public Response createAccount(Account account) {
    accountRepository.persist(account);                                       ❹
    return Response.status(201).entity(account).build();
  }

  @PUT
  @Path("{accountNumber}/withdrawal")
  @Transactional
  public Account withdrawal(@PathParam("accountNumber") Long accountNumber, String amount) {
    Account entity = accountRepository.findByAccountNumber(accountNumber);
    entity.withdrawFunds(new BigDecimal(amount));                             ❺
    return entity;
  }
}
```

❶ Inject an `AccountRepository` instance for data access operations.

❷ Retrieve all the accounts with `listAll()`.

❸ Use the custom data access method on `AccountRepository`.

❹ Persist a new `Account` instance into the database.

❺ Update the balance on the account without needing to call `accountRepository.persist()`, it's done automatically when the transaction completes.

The `AccountResourceTest` class can be copied from the JPA example, as both approaches use entities that have getters and setters.

With the same testing approach as the chapter so far, the tests can be run from the */chapter4/data-repository/* directory with:

```
mvn clean install
```

4.3.3 Which approach to use?

Through the previous sections different approaches were outlined for JPA, Active Record, and Data Repository. Which one is the best?

As with most things dealing with software, it depends.

Let's cover the key points of each approach:

- JPA
  - Easy migration for existing Java EE and Jakarta EE applications.
  - Requires creation of primary key field, not provided by default.
  - `@NamedQuery` annotations must be placed on an entity, or super class.
  - Queries require actual SQL, as opposed to shortcut versions that are used in *Active Record* or *Data Repository*.
  - Non primary key search requires SQL or `@NamedQuery`.
- Active Record
  - Doesn't require getters and setters for all fields.
  - Coupling the data access layer into an object makes testing it without a database difficult. The flip side is testing with a database is a lot easier than the past.
  - Another aspect of coupling, it breaks the *single responsibility principle* and *separation of concerns*.
- Data Repository
  - Requires creation of primary key field, not provided by default.
  - Clearly separates data access and business logic. Enabling them to be tested independently.
  - Without custom methods, it's an empty class. For some, this can seem unusual.

The above are some key differences shown through the previous sections, there are likely many more. When it comes down to it, the chosen approach will depend on the requirements of an

application and personal choice of the developer based on their previous experience.

It's worth noting that no one approach is wrong, or right, it all depends on personal perspective and preference.

4.4 Deployment to Kubernetes

Now the *Account Service* has a database, it's time to deploy it to Kubernetes to see it in action. First though, it's necessary to deploy a PostgreSQL instance that can be used by it.

4.4.1 Deploying PostgreSQL

There are a couple of pieces that need to be deployed to Kubernetes for setting up a PostgreSQL database:

1. Kubernetes Secret with encoded username and password. This secret will be used in creating the PostgreSQL database and in the Datasource configuration in Quarkus.
2. PostgreSQL database deployment.

First, verify Minikube is already running, and if it isn't, run:

```
minikube start
```

IMPORTANT **As mentioned in previous chapters, ensure that `eval $(minikube -p minikube docker-env)` is run in each terminal window that will be pushing a deployment to Minikube, as it uses Docker inside Minikube for building the image.**

Once Minikube is running, create the secret:

```
kubectl create secret generic db-credentials \          ❶
--from-literal=username=quarkus_banking \               ❷
--from-literal=password=quarkus_banking                 ❸
```

❶ Create a new secret with the name `db-credentials`.

❷ Add `username` with plain text value of `quarkus_banking`. The value will be encoded as part of creating the secret.

❸ Also set a `password` in the secret.

NOTE **If the Minikube instance being used is the same as in chapter 3, `kubectl delete secret generic db-credentials` will need to be executed first.**

With the secret created, the PostgreSQL database instance can be started. To do that, it requires a Kubernetes Deployment and Service.

Change into the directory */chapter4/* and run:

```
kubectl apply -f postgresql_kubernetes.yml
```

If successful, the terminal will contain messages stating the Deployment and Service were created. With a PostgreSQL database running, it's time to package and deploy a service to use it.

4.4.2 Package and Deploy

Any of the examples from the chapter could be used to show it working in Kubernetes, but in this instance the *Active Record* example will be used.

Before packaging the application, there are a few changes needed as it will be reading Kubernetes secrets for database configuration.

Add a new dependency in `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-kubernetes-config</artifactId>
</dependency>
```

This dependency enables an application to read Kubernetes ConfigMaps and Secrets. For it to know where the information is to read, some additional properties are needed:

```
%prod.quarkus.kubernetes-config.enabled=true                ❶
%prod.quarkus.kubernetes-config.secrets.enabled=true        ❷
%prod.quarkus.kubernetes-config.secrets=db-credentials      ❸
```

❶ Enable the extension.

❷ Tell the extension that secrets will be read.

❸ List the secrets to be read. In this case it's just `db-credentials`.

NOTE **The `%prod.` prefix ensures the settings are not used during development and testing.**

As well as the above additions to `application.properties`, the datasource information needs to be modified for Kubernetes:

```
%prod.quarkus.datasource.username=${username}               ❶
%prod.quarkus.datasource.password=${password}
%prod.quarkus.datasource.jdbc.url=jdbc:postgresql://postgres.default:5432/quarkus_banking      ❷
```

❶ Use variables for the username and password as they will be retrieved from the secret.

❷ Update the URL to be in the format of "<service-name>.<namespace>:<port>/<database>". In this example, `postgres` is the service name, and the namespace is `default`.

Those are the only changes needed to have the database credentials read from a secret, and the PostgreSQL database used within Kubernetes. With the changes made, time to build the image and deploy it to Kubernetes:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

With *Account Service* deployed, run `minikube service list` to see the details of all services:

```
|---------------------|--------------------------|-------------|--------------------------|
|      NAMESPACE      |           NAME           | TARGET PORT |            URL           |
|---------------------|--------------------------|-------------|--------------------------|
| default             | account-service          | http/8080   | http://192.168.64.2:30704 |
| default             | kubernetes               | No node port |
| default             | postgres                 | http/5432   | http://192.168.64.2:31615 |
| kube-system         | kube-dns                 | No node port |
| kubernetes-dashboard | dashboard-metrics-scraper | No node port |
| kubernetes-dashboard | kubernetes-dashboard     | No node port |
|---------------------|--------------------------|-------------|--------------------------|
```

Accessing *http://192.168.64.2:30704/accounts* in a browser will now retrieve all the accounts in the PostgreSQL database running in Kubernetes!

Throughout the chapter each example has shown the different approaches that can be taken for writing database code with Quarkus. Beginning with JPA for easy migration to Quarkus, before progressing to cover the enhancements to Hibernate ORM that Panache brings through use of *Active Record* or *Data Repository* approaches.

4.5 Summary

- By adding the datasource properties for `db-kind`, `username`, `password`, and `jdbc.url`, along with the `quarkus-jdbc-postgresql` dependency, a Quarkus application is able to connect with a PostgreSQL database.
- Using `@NamedQuery` on a JPA entity class to define custom queries for use with the `EntityManager`.
- Hibernate ORM with Panache offers simplified approaches to JPA with either the *Active Record* or *Data Repository* approaches.
- Add the `quarkus-hibernate-orm-panache` dependency, and use `PanacheEntity` as a super class for JPA entity classes to use the *Active Record* approach. The *Active Record* approach provides common methods for use when interacting with a database, simplifying the data access layer in an application.
- When using the *Data Repository* approach, create a repository class implementing `PanacheRepository` to hold custom data access methods, such as queries equivalent to `@NamedQuery` on JPA entities.
- Define a PostgreSQL deployment and service in Kubernetes, and create the resources in a Kubernetes environment using Minikube.

5 Clients for consuming other microservices

This chapter covers:

- MicroProfile REST Client specification
- Using Type Safe interfaces to consume external microservices
- Customizing the content of headers on the request

While many microservices only require a database, alternative data services, or process a request within its own process, there will be times when a microservice needs to communicate with other microservices to fulfill a request. When shifting from monoliths to microservices, the tendency is towards smaller and leaner microservices which necessitates more of them. More importantly, many of those smaller microservices will need to communicate with each other to complete a task previously achieved with a single method calling other services inside a monolith. All those previous "in process" method calls are now external microservice invocations.

This chapter introduces the MicroProfile REST Client, and how Quarkus implements the specification to provide a Type safe means of interacting with external services. Though there are many possible approaches, including Javas' networking library, or third party libraries like okhttp and Apache HttpClient. Quarkus abstracts away the underlying HTTP transport construction from the developer, enabling them to focus on the task of defining the external service, and interacting with the service as if it were a local method invocation.

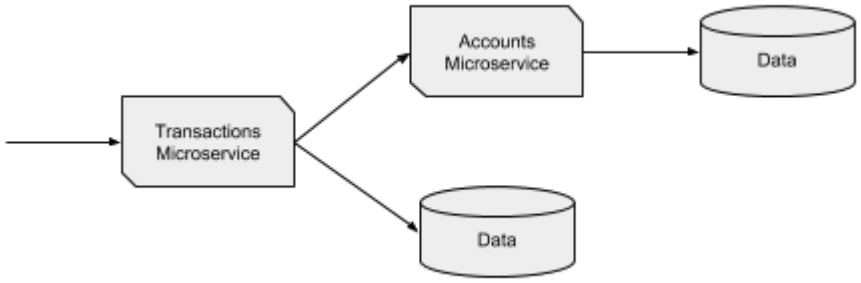


Figure 5.1 Banking Microservice consumption

Figure 5.1 represents a microservice calling another microservice, but is also the basis of the examples used throughout the chapter. The examples for this chapter follow on from the previous banking domain examples. The *Transaction Service* will call the *Account Service* to retrieve the current balance to ensure the requested transaction doesn't result in an overdrawn account.

The *Account Service* in the chapter is not the same as the version seen in earlier chapters. As the chapter's focus is on the calling microservice, and not the called microservice, the *Account Service* API will only expose methods to retrieve the current balance and update it.

> **NOTE** The Account Service for the chapter won't be shown as it's a derivative of earlier chapters. Take a look at the code for Chapter 5, in the /chapter5/account-service/ directory, to see how it was implemented.

5.1 What is MicroProfile REST Client?

MicroProfile REST Client is one of the specifications from Eclipse MicroProfile[16]. The specification defines how a representation of an external service with Java Interfaces ensures interactions with that service occur in a type-safe manner. What that means is utilizing the Java language and compilation process to ensure the code to interact with an external service is free of obvious errors.

When interacting with services utilizing one of the many HTTP libraries, or the JAX-RS Client library, it's necessary to perform a lot of casting between objects, transformations from JSON to POJOs, and many other steps that don't rely on the Java language to ensure its correctness. Though workable, it leaves code susceptible to failures due to problems not being discovered during compilation, but only through testing, or even, production usage. Using a type safe approach, with MicroProfile REST Client, allows these types of problems to be discovered during compilation and not much later during execution.

For many years the RESTEasy project[17] had a custom means of defining external services with

Java Interfaces. However, as it was included within only one JAX-RS implementation, it meant other JAX-RS implementations didn't have such a feature. Building on the RESTEasy projects work, the Thorntail project[18] added a CDI layer on top of the programmatic builder from RESTEasy.

MicroProfile REST Client defines a specification to combine the ideas from RESTEasy and Thorntail for the Eclipse MicroProfile platform. There are many aspects with the specification that align it with how JAX-RS defines RESTFul endpoints.

Some of the more important features of the specification include:

- Including additional client headers onto any external request.
- Following responses redirecting to another URL.
- Calling external services through an HTTP proxy.
- Registering custom providers for filtering, message body manipulation, interceptors, and exception mappers.
- Automatic registration of JSON-P and JSON-B providers.
- Configuring SSL for REST Client endpoints.

Having covered the origins of the specification, and its purpose, it's now time to begin using it with Quarkus.

5.2 Service Interface definition

For the *Transaction Service* to be able to communicate with the *Account Service* it needs to know what methods are available, their parameters, and their return types. Without that information the *Transaction Service* doesn't know the API contract of the *Account Service*.

There are many libraries available that support communicating with other services via HTTP and other protocols, including classes within the JDK itself. However, taking such an approach requires more complex code to handle setting the appropriate content type, setting any headers, and handling response codes for different situations. Let's take a look at a service definition for *Account Service*:

Listing 5.1 AccountService

```
@Path("/accounts")                                                          ❶
@RegisterRestClient                                                         ❷
@Produces(MediaType.APPLICATION_JSON)                                       ❸
public interface AccountService {
  @GET
  @Path("/{acctNumber}/balance")
  BigDecimal getBalance(@PathParam("acctNumber") Long accountNumber);       ❹

  @POST
  @Path("{accountNumber}/transaction")
  void transact(@PathParam("accountNumber") Long accountNumber, BigDecimal amount);    ❺
}
```

❶ Define the path of the service, excluding the base url portion.

❷ Indicates that the interface should have a CDI bean created that can be injected into classes.

❸ Sets all methods of the service to return JSON.

❹ Method for retrieving the account balance, with HTTP method and Path annotations.

❺ Same as (4), but for making a transaction on the account.

Looking at the interface definition it likely seemed very familiar, and there's a very good reason for that. The way in which a Java Interface defines the service deliberately uses the well-known JAX-RS annotations for a class and its methods. Utilizing the same JAX-RS annotations on a Java Interface to define a remote service as is used for creating a JAX-RS resource class means all the annotations used are already familiar to developers. If defining a service with a Java Interface used completely different annotations, or an entirely different way to define the service, developers would find it much more difficult to learn and use.

The only difference in the Java Interface compared to what a JAX-RS resource would contain is the `@RegisterRestClient` annotation. This annotation tells Quarkus that a CDI bean needs to be created containing the methods on the interface. Quarkus wires up the CDI bean such that calls onto the interface methods result in HTTP calls to the external service. Let's take a look at how the execution flow works:

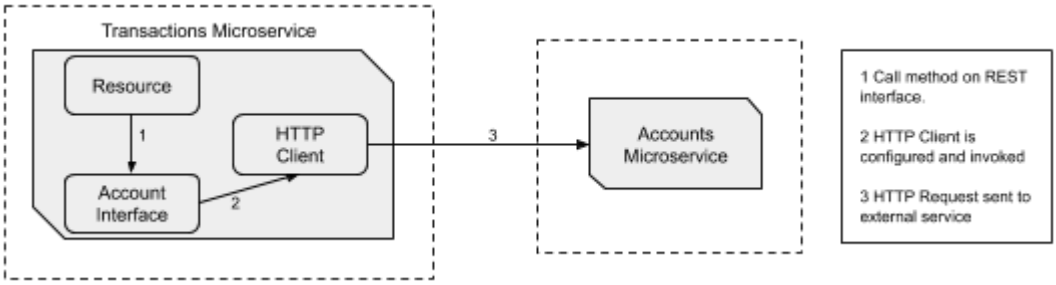


Figure 5.2 Account Transaction Service - REST Client call

In *Figure 5.2* the dotted boxes represent separate process boundaries. It doesn't matter whether it is a physical machine, or Kubernetes pod. The flow of execution when calling the *Account Service* in *Figure 5.2* is:

1. JAX-RS resource calls a method on the `AccountService` interface, which executes the call on the CDI Bean that implements the interface.
2. The CDI Bean, representing the `AccountService` interface, configures the HTTP client with the URL, HTTP method type, content types, headers, or anything else needing to be set onto the HTTP request.
3. The HTTP client issues the HTTP request to the external *Account Service*, and handles the response that's returned.

One thing that has not been mentioned so far, is how to define the URL of where the external service exists! As with many things, there are several ways to achieve it.

When `@RegisterRestClient` is present the URL can be set directly on the annotation with the `baseUri` parameter. Though that is not a great way to set it for production, as URLs can change, it's an easy way to configure it to get started. With the `baseUri` parameter set on the annotation, it's still possible to override the value with configuration. The configuration key for setting the URL is `{packageName}.{interfaceName}/mp-rest/url`, which can be added to the `application.properties` file with the URL of the external service.

For *Listing 5.1* above, the configuration key is `io.quarkus.transactions.AccountService/mp-rest/url`. Such a long key can be difficult to remember and is open to mistakes. To simplify the configuration key, set the `configKey` parameter of `@RegisterRestClient`. For instance, defining the `configKey` on the interface:

```
@RegisterRestClient(configKey = "account-service")
public interface AccountService {
}
```

The above makes the configuration key `account-service/mp-rest/url`, making it less prone to errors.

Having covered how to create a service definition for any external service, let's actually use it in the *Transaction Service*.

5.2.1 CDI REST Client

The previous section discussed how Quarkus automatically creates a CDI bean from the Java Interface when `@RegisterRestClient` is present. Now it's time to see how it can be used:

Listing 5.2 TransactionResource

```
public class TransactionResource {
  @Inject                                          ❶
  @RestClient                                      ❷
  AccountService accountService;                   ❸

  @POST
  @Path("/{acctNumber}")
  public Response newTransaction(@PathParam("acctNumber") Long accountNumber, BigDecimal amount)
  {
    accountService.transact(accountNumber, amount);        ❹
    return Response.ok().build();
  }
}
```

❶ To inject the CDI bean for the interface, it is necessary to explicitly use `@Inject`. Though it's not required for other situations, it is when injecting a rest client interface.

❷ CDI Qualifier telling Quarkus to inject a Type Safe REST Client bean matching the interface.

❸ REST Client interface representing the external service.

❹ Call the external service method.

Being able to call an external service with a single method call, as if it was a local service, is extremely powerful and simplifies making HTTP calls in the code. For developers familiar with Java EE, this looks very similar to Remote EJBs. In many respects it is very similar, except that instead of communicating with Remote Method Invocation (RMI), it uses HTTP.

With the interface defined, and a JAX-RS resource method that uses it, now it's time to test it.

MOCK THE EXTERNAL SERVICE

When unit testing, setting up and running the service to be called is far from ideal. To verify some basic operation of the *Transaction Service*, it's necessary to use a server to mock the responses that would be received from the *Account Service*. One option is to create a server that handles a request and provides an appropriate response, but thankfully there's a handy library that offers that exact functionality, Wiremock.

First step is to add the required dependency:

```
<dependency>
  <groupId>com.github.tomakehurst</groupId>
  <artifactId>wiremock-jre8</artifactId>
  <scope>test</scope>
</dependency>
```

To assist in setting up an environment for testing, Quarkus provides `QuarkusTestResourceLifecycleManager`. Implementing `QuarkusTestResourceLifecycleManager` enables customization of what happens during `start()` and `stop()` during the lifecycle of a test. Any implementation is applied to a test with `@QuarkusTestResource`. One is needed to interact with the Wiremock server:

Listing 5.3 WiremockAccountService

```
public class WiremockAccountService implements QuarkusTestResourceLifecycleManager {    ❶
  private WireMockServer wireMockServer;                                                 ❷

  @Override
  public Map<String, String> start() {
    wireMockServer = new WireMockServer();                                               ❸
    wireMockServer.start();

    stubFor(get(urlEqualTo("/accounts/121212/balance"))                                  ❹
        .willReturn(aResponse()
            .withHeader("Content-Type", "application/json")
            .withBody("435.76")
        ));

    stubFor(post(urlEqualTo("/accounts/121212/transaction"))                             ❺
        .willReturn(noContent())
    );

    return Collections.singletonMap("io.quarkus.transactions.AccountService/mp-rest/url",
        wireMockServer.baseUrl());                                                       ❻
  }

  @Override
  public void stop() {
    if (null != wireMockServer) {
      wireMockServer.stop();                                                             ❼
    }
  }
}
```

❶ Implement `QuarkusTestResourceLifecycleManager` to respond to the start and stop events of the test.

❷ Store the `WireMockServer` instance to enable stopping it during test shutdown.

❸ Create the `WireMockServer` and start it.

❹ Provide a stub for responding to the HTTP GET method for retrieving an account balance. As it's a mock server, the account number the server responds to needs to be hard coded and used in the request from a test.

❺ Create another stub, this one for responding to the HTTP POST method to create a transaction.

❻ Set an environment variable named `io.quarkus.transactions.AccountService/mp-rest/url` to the URL of the Wiremock server. The variable name matches the expected name of the configuration key for defining the URL.

❼ Stop the Wiremock server during test shutdown processing.

Lastly a test needs to be written to utilize the *Transaction Service* which will call the mock server:

Listing 5.4 TransactionServiceTest

```
@QuarkusTest
@QuarkusTestResource(WiremockAccountService.class)          ❶
public class TransactionServiceTest {
  @Test
  void testTransaction() {
        given()
            .body("142.12")
            .contentType(ContentType.JSON)
            .when().post("/transactions/{accountNumber}", 121212)       ❷
            .then()
            .statusCode(200);                                           ❸
  }
}
```

❶ Add the lifecycle manager for Wiremock to the test.

❷ Issue an HTTP POST request using the account number defined in the Wiremock stub.

❸ Verify a response code of 200 is returned.

With the test written, open up the */chapter5/transaction-service/* directory and run:

```
mvn clean install
```

The test should pass without issue.

Running the test using a mock server doesn't provide much confidence it's correct, so let's deploy all the services to Kubernetes to verify it.

DEPLOYING TO KUBERNETES

If Minikube is already running, great, if it isn't, run:

```
minikube start
```

With Minikube running, the PostgreSQL database instance can be started. To do that, install the Kubernetes Deployment and Service for PostgreSQL.

Change into the */chapter5/* directory and run:

```
kubectl apply -f postgresql_kubernetes.yml
```

WARNING **This PostgreSQL instance doesn't use secrets for the username and password, unlike Chapter 4. For this reason, this setup is not recommended for production usage.**

Change into the */chapter5/account-service/* directory to build and deploy the *Account Service* to Kubernetes:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

NOTE **Run `eval $(minikube -p minikube docker-env)` before the above command to ensure the container image build uses Docker inside Minikube.**

Verify the service has started properly by running `kubectl get pods`:

```
NAME                                  READY   STATUS    RESTARTS   AGE
account-service-6d6d7655cf-ktmhv      1/1     Running   0          6m55s
postgres-775d4d9dd5-b9v42             1/1     Running   0          13m
```

If there are errors, indicated by the `STATUS` column containing "Error", run `kubectl logs account-service-6d6d7655cf-ktmhv`, using the actual pod name, to show the logs of the container for diagnosing the error.

Find the URL of the *Account Service* by running `minikube service list`, and then verify it's working by running:

```
curl http://192.168.64.4:30704/accounts/444666/balance
```

The terminal will show the balance returned, which should be `3499.12` if everything worked.

With the *Account Service* deployed and working, time to do the same for the *Transaction Service* . Remember, the URL needs to be set so that the *Account Service* can be found. Do that by modifying `application.properties` to include:

```
%prod.io.quarkus.transactions.AccountService/mp-rest/url=http://account-service:8080
```

It uses the production profile (`%prod`) as the URL only applies when deployed to Kubernetes, and it's using the Kubernetes service name for the *Account Service* that is returned from `minikube service list`.

Change into the */chapter5/transaction-service/* directory and deploy the service:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

Verify the service has started without error and issue a request to withdraw funds from an account:

```
curl -H "Content-Type: application/json" -X POST -d "-143.43"
http://192.168.64.4:31692/transactions/444666
```

If it completes with no errors and messages, run the curl command from earlier to check the account balance. If everything worked as intended, the balance returned should now be `3355.69`!

Have some fun exploring by depositing and withdrawing different amounts from various

accounts, and see how the balance changes after each request.

Though they haven't been used so far, there are many other configuration options when using REST Client with CDI. With *Listing 5.1* as an interface, here is a list of different configurations that could be used:

```
io.quarkus.transactions.AccountService/mp-rest/url=http://localhost:8080                    ❶
io.quarkus.transactions.AccountService/mp-rest/scope=javax.inject.Singleton                 ❷
io.quarkus.transactions.AccountService/mp-rest/providers=io.quarkus.transactions.MyProvider ❸
io.quarkus.transactions.AccountService/mp-rest/connectTimeout=400                           ❹
io.quarkus.transactions.AccountService/mp-rest/readTimeout=1000                             ❺
io.quarkus.transactions.AccountService/mp-rest/followRedirects=true                         ❻
io.quarkus.transactions.AccountService/mp-rest/proxyAddress=http://myproxy:9100             ❼
```

❶ The URL where the external service is available, as seen in examples earlier.

❷ By default, the scope of a CDI Bean for a REST Client is `@Dependent`. This would change it to be `@Singleton` instead.

❸ Comma separated list of JAX-RS providers that should be used with the client.

❹ Timeout for connecting to the remote endpoint in milliseconds.

❺ How long to wait for a response from a remote endpoint in milliseconds.

❻ Determines if HTTP Redirect responses are followed or an error is returned.

❼ HTTP Proxy to be used for all HTTP requests from the client.

The above configuration can also be achieved with the programmatic API, which will be seen in the next section. If `configKey` on `@RegisterRestClient` had been used, all the above configuration keys could replace `io.quarkus.transactions.AccountService/mp-rest/` with `account-service/mp-rest/`.

Using CDI isn't the only way to use REST Client for connecting to external services, let's take a look at doing the same as above with the programmatic API.

5.2.2 Programmatic REST Client

In addition to utilizing CDI for injecting and calling REST Client beans for external interfaces, a programmatic builder API can be used instead. This API provides more control over the various settings of the REST Client without needing to manipulate configuration values.

Listing 5.5 AccountService

```
@Path("/accounts")
@Produces(MediaType.APPLICATION_JSON)
public interface AccountServiceProgrammatic {
  @GET
  @Path("/{acctNumber}/balance")
  BigDecimal getBalance(@PathParam("acctNumber") Long accountNumber);

  @POST
  @Path("{accountNumber}/transaction")
  void transact(@PathParam("accountNumber") Long accountNumber, BigDecimal amount);
}
```

The only difference between the above interface and *Listing 5.1* is the removal of `@RegisterRestClient`. Though the same interface can be used for CDI and programmatic API usage, it's important to show that `@RegisterRestClient` is not required for programmatic API usage.

Listing 5.6 TransactionResource

```
@Path("/transactions")
public class TransactionResource {
  @ConfigProperty(name = "account.service", defaultValue = "http://localhost:8080")        ❶
  String accountServiceUrl;

  ...

  @POST
  @Path("/api/{acctNumber}")                                          ❷
  public Response newTransactionWithApi(@PathParam("acctNumber") Long accountNumber,
  BigDecimal amount) throws MalformedURLException {
    AccountServiceProgrammatic acctService =
        RestClientBuilder.newBuilder()                                ❸
            .baseUrl(new URL(accountServiceUrl))                      ❹
            .connectTimeout(500, TimeUnit.MILLISECONDS)               ❺
            .readTimeout(1200, TimeUnit.MILLISECONDS)                 ❻
            .build(AccountServiceProgrammatic.class);                 ❼

    acctService.transact(accountNumber, amount);                      ❽
    return Response.ok().build();
  }
}
```

❶ Inject a value for the configuration key `account.service`, defaulting it to `http://localhost:8080` if it's not found.

❷ Add the new programmatic API to the */transactions/api/* URL path.

❸ Use `RestClientBuilder` to create a builder instance for setting features programmatically.

❹ Set the URL for any requests with the REST Client, which is equivalent to `baseUrl` on `@RegisterRestClient`. Use the configuration value of `account.service` to create a new URL.

❺ The maximum amount of wait time allowed when connecting to an external service.

6. How long to wait for a response before triggering an exception.
7. Build a proxy of the `AccountServiceProgrammatic` interface for calling the external service.
8. Call the service in the same way as done previously with a CDI Bean.

Add the following configuration into `application.properties`:

```
%prod.account.service=http://account-service:8080
```

Now build the *Transaction Service* and re-deploy it to Kubernetes:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

Verify the service has started without error and issue a request to deposit funds into an account:

```
curl -H "Content-Type: application/json" -X POST -d "2.03"
http://192.168.64.4:31692/transactions/api/444666
```

Running `curl http://192.168.64.4:30704/accounts/444666/balance` will return a balance that should be $2.03 more than it was previously. Try out different combinations of depositing and withdrawing funds from accounts with the programmatic API. Both the CDI Bean and programmatic API endpoints shouldn't result in any different outcomes.

Utilizing the programmatic API with `RestClientBuilder` provides greater control over the configuration of the client. Whether specifying the URL of the external service, registering JAX-RS providers, setting connection and read timeouts, or any other setting, it can all be done with the `RestClientBuilder`.

5.2.3 Choosing between CDI and Programmatic API

There is no right or wrong answer here, it all comes down to preference.

Some developers are more comfortable dealing with CDI Beans, while others prefer using programmatic APIs to fully control the process. There is one caveat when using CDI Beans for REST Client, it does require more configuration in `application.properties` compared to the programmatic API approach. However, whether that is a problem very much depends on what aspects the developer wants control over. If they require more control, then it's likely easier to do that with the programmatic API and not configuration properties.

Whichever approach is chosen, it doesn't impact the Type safe guarantees of the REST Client. It only impacts the interaction with the interface.

In addition, both approaches provide a Thread safe means of communicating with external resources. In Listing 5.6, the `acctService` could be stored in a variable on the class. In this case

it wasn't to simplify the code.

5.2.4 Asynchronous response types

In recent years there's more desire to write reactive code, and ideally not blocking threads while waiting. As calling an external service could be a slow operation, depending on network latency, network load, and many other factors, it would be a good idea to utilize asynchronous types when using the REST Client.

First thing to do is to update the `AccountService` and `AccountServiceProgrammatic` interfaces with the following method:

```
@POST
@Path("{accountNumber}/transaction")
CompletionStage<Void> transactAsync(@PathParam("accountNumber") Long accountNumber,
 BigDecimal amount);
```

The only change from the original `transact()` method on the interface is the return type. Instead of returning `void`, the method now returns `CompletionStage<Void>`. In essence the method is still returning a void response, but wrapping it in a `CompletionStage` allows the method to complete and handling the response from the HTTP request will happen later once received. While the response has not been received, the method execution completes and processing of the response waits. Doing so frees up the thread that was processing the request to handle other requests, while waiting for an asynchronous response.

With the interfaces updated, how different are the JAX-RS resource methods?

Listing 5.7 TransactionResource

```
@POST
@Path("/async/{acctNumber}")                                    ❶
public CompletionStage<Void> newTransactionAsync(@PathParam("acctNumber")
Long accountNumber, BigDecimal amount) {
  return accountService.transactAsync(accountNumber, amount);   ❷
}

@POST
@Path("/api/async/{acctNumber}")
public CompletionStage<Void> newTransactionWithApiAsync(@PathParam("acctNumber")
Long accountNumber, BigDecimal amount) throws MalformedURLException {
  AccountServiceProgrammatic acctService =
      RestClientBuilder.newBuilder()
          .baseUrl(new URL(accountServiceUrl))
          .build(AccountServiceProgrammatic.class);

  return acctService.transactAsync(accountNumber, amount);      ❸
}
```

❶ Use a different URL path for the asynchronous version. Return type is now `CompletionStage<Void>` instead of `Response`.

❷ Method body modified to returning the result of REST Client call.

❸ As with (2), instead of returning a `Response` to indicate everything is ok, return the `CompletionStage` returned from the REST Client call instead.

With the *Transaction Service* updated, re-deploy the changes with:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

Once verified that the service started successfully with `kubectl get pods`, retrieve an account balance and then deposit an amount using the new asynchronous API URL:

```
curl http://192.168.64.4:30704/accounts/444666/balance
curl -H "Content-Type: application/json" -X POST -d "5.63"
http://192.168.64.4:31692/transactions/async/444666
```

Though the new methods use asynchronous return types, they achieve the same outcome as the synchronous ones. Take some time to experiment with the asynchronous methods, see what the upper limit of the number of parallel requests might be?

5.3 Customizing REST Clients

So far the examples have focused on normal usage, defining an interface and then executing methods on that interface to execute HTTP requests. There are many additional features offered through REST Client, some of which will be covered in the following sections.

5.3.1 Client request headers

All requests that are received or sent from applications will contain many headers within them. Some everyone is familiar with, such as *Content-Type* and *Authorization*, but many headers are passed down a call chain. With REST Client, it's possible to add custom request headers into the outgoing client request or ensure that headers from an incoming JAX-RS request propagate to a subsequent client request.

To see the headers received in the *Account Service*, modify `AccountResource` to return them from the request. Another option is to log out the headers inside the service.

Listing 5.8 AccountResource

```
public class AccountResource {
  @POST
  @Path("{accountNumber}/transaction")
  @Transactional
  public Map<String, List<String>> transact(@Context HttpHeaders headers,          ❶
      @PathParam("accountNumber") Long accountNumber, BigDecimal amount) {
    ...

    return headers.getRequestHeaders();            ❷
  }
}
```

❶ Inject the `HttpHeaders` of the HTTP Request into the method. `@Context` is specific to JAX-RS, but acts in a manner similar to `@Inject` with CDI.

❷ Return the `Map` containing the HTTP Request headers.

With the *Account Service* modified, it's also necessary to modify the `AccountService` interface the *Transaction Service* uses.

Listing 5.9 AccountService

```
@Path("/accounts")
@RegisterRestClient
@ClientHeaderParam(name = "class-level-param", value = "AccountService interface")        ❶
@RegisterClientHeaders                                                                    ❷
@Produces(MediaType.APPLICATION_JSON)
public interface AccountService {
  ...

  @POST
  @Path("{accountNumber}/transaction")
  Map<String, List<String>> transact(@PathParam("accountNumber") Long accountNumber,
  BigDecimal amount);     ❸

  @POST
  @Path("{accountNumber}/transaction")
  @ClientHeaderParam(name = "method-level-param", value = "{generateValue}")        ❹
  CompletionStage<Map<String, List<String>>> transactAsync(@PathParam("accountNumber")
  Long accountNumber,
        BigDecimal amount);        ❺

  default String generateValue() {          ❻
    return "Value generated in method for async call";
  }
}
```

❶ Adds `class-level-param` to the outgoing HTTP Request header. Adding it on the interface means all methods will have the header added.

❷ Indicates the default `ClientHeadersFactory` should be used. The default factory will propagate any headers from an inbound JAX-RS request onto the outbound client request, where the headers are added as a comma separated list into the configuration key named `org.eclipse.microprofile.rest.client.propagateHeaders`.

❸ Modify the return type to be a `Map` of the headers.

❹ Similar to the usage of `@ClientHeaderParam` in (1), it adds the `method-level-param` header to the outbound HTTP request. This usage is different as it doesn't use a static value, but instead calls a method named `generateValue()`. Adding it to a method cause the header to only be added to any requests created when calling the `transactAsync` method. If the method is in a different class, the value needs to include the package and class name.

❺ As with (3), return a `CompletionStage` of the `Map` of headers.

❻ Default method on the interface used to create a value for the header from (4).

There are new features included in *Listing 5.9* to cover in detail.

`@ClientHeaderParam` is a convenient way to add request headers onto the HTTP request that is sent to the external service. As seen above, the annotation can define a `value` that is a constant string, or it can use a method, either on the interface itself or in another class, by using curly braces to surround the name of the method. Calling a method to add a header is useful for setting an authentication token on the request, which would be necessary to call secured services, and a token isn't present on the incoming request.

Is there an advantage to using `@ClientHeaderParam`? Another option is adding header parameters using `@HeaderParam` on a method parameter. The problem with the `@HeaderParam` approach is it requires additional parameters on any interface method. Maybe if adding one or two parameters, it's not too bad, but what about three, four, or even six parameters! Not only does it clutter up the method definition in the interface, whenever making a call to the method, all those parameters need to be passed as well. This is where `@ClientHeaderParam` is helpful, keeping the interface methods uncluttered and simplifying the method invocation.

`@RegisterClientHeaders` is similar to `@ClientHeaderParam`, but for propagating headers and not adding new ones. The default behavior when there's an incoming JAX-RS request is for no headers to be passed onto any subsequent REST Client call. Using `@RegisterClientHeaders` allows specific headers to be propagated from an incoming JAX-RS request.

Which headers should be propagated is specified in the configuration with the `org.eclipse.microprofile.rest.client.propagateHeaders` key, where the value is a comma separated list of header names to propagate. This feature is especially useful for propagating authentication headers from incoming requests onto REST Client calls, but do make sure it makes sense for them to be passed. Sometimes passing authentication headers from incoming to outgoing can have unintended consequences, such as performing operations on a service with an unexpected user identity.

If the default header propagation isn't sufficient, maybe it's needed to modify the content of a particular header, `@RegisterClientHeaders` allows the usage of a custom implementation. For example, `@RegisterClientHeaders(MyHeaderClass.class)` says use a custom implementation, where `MyHeaderClass` extends `ClientHeadersFactory`. The only method on `ClientHeadersFactory` to implement is `update()`, which has method arguments for the `MultiMap` containing the headers from the incoming JAX-RS request, and a `MultiMap` with the headers to be used on the outgoing REST Client call. Updating the headers on the outgoing headers will alter what is set on the HTTP request to the external service.

The changes to be made in the `TransactionResource` is modifying the return types of `newTransaction` and `newTransactionAsync` to use a `Map` for the headers.

The last thing needed is to specify which headers need to be automatically propagated, without

doing that `@RegisterClientHeaders` will not propagate anything. Add the following to `application.properties` of the *Transaction Service*:

```
org.eclipse.microprofile.rest.client.propagateHeaders=Special-Header          ❶
```

❶ The header name to be propagated is *Special-Header*.

With all those changes made, deploy the updated *Account Service* and *Transaction Service* to Kubernetes:

```
/chapter5/account-service > mvn clean package -Dquarkus.kubernetes.deploy=true
/chapter5/transaction-service > mvn clean package -Dquarkus.kubernetes.deploy=true
```

With both services updated, it's time to see the headers being passed. Let's run the synchronous transaction method first, which should only have the class level header added:

```
curl -H "Content-Type: application/json" -X POST -d "7.89"
http://192.168.64.4:31692/transactions/444666
```

The terminal output should contain:

```
{
  "class-level-param":["AccountService-interface"],            ❶
  "Accept":["application/json"],
  "Connection":["Keep-Alive"],
  "User-Agent":["Apache-HttpClient/4.5.12 (Java/11.0.5)"],
  "Host":["account-service:8080"],
  "Content-Length":["4"],
  "Content-Type":["application/json"]
}
```

❶ Header passed via `@ClientHeaderParam` on `AccountService`.

Now let's do the same with the asynchronous transaction. This time both the class level and method level headers should be present:

```
curl -H "Content-Type: application/json" -X POST -d "6.12"
http://192.168.64.4:31692/transactions/async/444666
```

The output should now include:

```
{
  "class-level-param":["AccountService-interface"],                    ❶
  "method-level-param":["Value generated in method for async call"],   ❷
  "Accept":["application/json"],
  "Connection":["Keep-Alive"],
  "User-Agent":["Apache-HttpClient/4.5.12 (Java/11.0.5)"],
  "Host":["account-service:8080"],
  "Content-Length":["4"],
  "Content-Type":["application/json"]
}
```

❶ Class level header from `AccountService`.

❷ Header passed via `@ClientHeaderParam` on `transactAsync` method of `AccountService`.

How about the propagation of headers? For that it's necessary to pass a header with curl:

```
curl -H "Special-Header: specialValue" -H "Content-Type: application/json"
-X POST -d "10.32" http://192.168.64.4:31692/transactions/444666
curl -H "Special-Header: specialValue" -H "Content-Type: application/json"
-X POST -d "9.21" http://192.168.64.4:31692/transactions/async/444666
```

If it worked as expected, the terminal output will contain the headers for each of the previous examples, in addition to the *Special-Header* that was passed into the initial call.

SIDEBAR **Exercise for the reader**

Modify the programmatic API versions in AccountServiceProgrammatic and TransactionResource, and try out the /api endpoints to see the headers.

In this section, the different approaches to including additional headers on the client request were covered. `@ClientHeaderParam` can be added to a REST Client interface for applying to all methods, or added to specific methods only. `@ClientHeaderParam` allows setting a static value as the header, or calling a method to retrieve a necessary value for the header.

5.3.2 Declaring providers

There are many JAX-RS providers that can be written to adjust a request or response, such as `ClientRequestFilter`, `ClientResponseFilter`, `MessageBodyReader`, `MessageBodyWriter`, `ParamConverter`, `ReaderInterceptor`, and `WriterInterceptor`. There is also the `ResponseExceptionMapper` from REST Client.

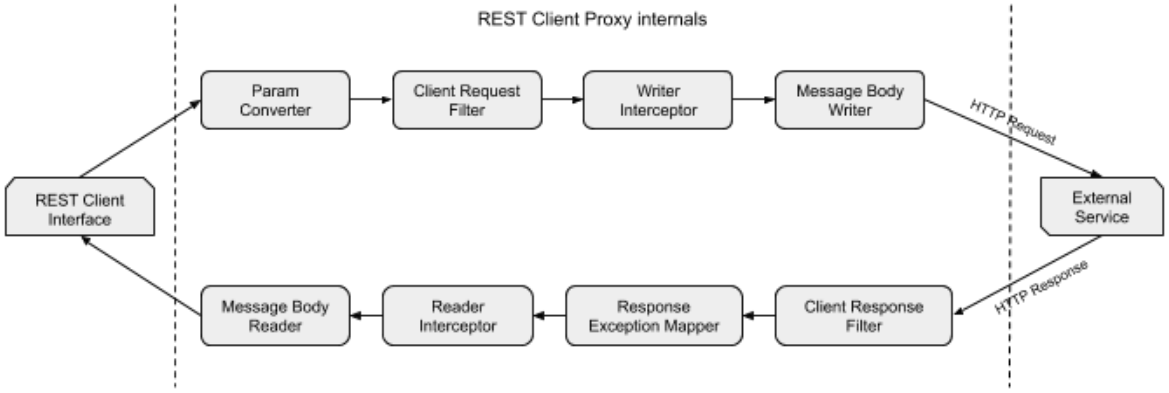


Figure 5.3 Provider processing sequence of REST Client Proxy

Figure 5.3 highlights the sequence of JAX-RS and REST Client provider execution in preparing the HTTP Request and handling the HTTP Response.

Any provider classes that implement the above interfaces can register them for use in a few ways:

1. Add `@Provider` onto the class itself. This is the least flexible as it means any JAX-RS interaction will include the provider, irrespective of whether it's an incoming JAX-RS request, or an outgoing REST Client call.
2. Associate a provider class with a specific REST Client interface by adding `@RegisterProvider(MyProvider.class)` to the interface.
3. When using the programmatic API, call `builder.register(MyProvider.class)` to use the provider with a particular REST Client call.
4. Implement either `RestClientBuilderListener` or `RestClientListener` and register the provider directly onto the `RestClientBuilder`.

The following sections will cover in detail how to use client filters and exception mappers.

CLIENT REQUEST FILTER

This section will show how to write and apply a `ClientRequestFilter` to REST Client calls. There's not much to writing a request filter, so let's write one to add a new header onto the request containing the name of the invoked method:

Listing 5.10 AccountRequestFilter

```
public class AccountRequestFilter implements ClientRequestFilter {        ❶
  @Override
  public void filter(ClientRequestContext requestContext) throws IOException {        ❷
    String invokedMethod =
      (String) requestContext.getProperty("org.eclipse.microprofile
      .rest.client.invokedMethod");        ❸
    requestContext.getHeaders().add("Invoked-Client-Method", invokedMethod);        ❹
  }
}
```

❶ The class needs to implement `ClientRequestFilter`.

❷ Override the `filter` method to perform whatever filtering is needed. The method has access to the `ClientRequestContext` to amend what is sent in the request.

❸ REST Client adds a property named `org.eclipse.microprofile.rest.client.invokedMethod` with the value of the interface method that is being invoked. In this case it is retrieved.

❹ Add a new request header named `Invoked-Client-Method` with the value from (3).

For the above filter to be used during invocation of a client request, register it as a provider onto `AccountService`:

```
@RegisterProvider(AccountRequestFilter.class)
public interface AccountService { ... }
```

Time to re-deploy the updated *Transaction Service* to Kubernetes:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

With the service updated, let's see the additional header added to the request by the filter. Verification can be done with either the synchronous or asynchronous version of the method, as it's applied directly on the interface it works on both executions.

```
curl -H "Content-Type: application/json" -X POST -d "15.64"
http://192.168.64.4:31692/transactions/444666
```

With the returned headers, the below should be present in the terminal:

```
{
  "class-level-param":["AccountService-interface"],
  "method-level-param":["Value generated in method for async call"],
  "Accept":["application/json"],
  "Invoked-Client-Method":["transact"],                    (1)
  "Connection":["Keep-Alive"],
  "User-Agent":["Apache-HttpClient/4.5.12 (Java/11.0.5)"],
  "Host":["account-service:8080"],
  "Content-Length":["4"],
  "Content-Type":["application/json"]
}
```

(1) Header added by the `AccountRequestFilter` showing the `transact` method of `AccountService` was called.

The returned results should be the same as the CDI Bean version seen above.

SIDEBAR **Exercise for the reader**

Modify the methods on `TransactionResource` that use the REST Client programmatic APIs to register the filter and run the tests on the URLs that use them.

This section covered how to register a `ClientRequestFilter`, or `ClientResponseFilter`, for a REST Client by adding `@RegisterProvider` with the name of the class.

MAPPING EXCEPTIONS

Another JAX-RS provider type that can be implemented is the `ResponseExceptionMapper`. This provider is specific to REST Client and will not work with JAX-RS endpoints. The purpose of the mapper is to convert the `Response` that is received from an external service into a `Throwable` that can be more easily handled.

IMPORTANT **The exception type of the mapper must be present on the "throws" clause of the method on the Interface for it to work.**

As with other JAX-RS providers, a specific `@Priority` can be set to indicate the precedence of

an exception mapper compared to others. The lower a priority number, the higher, or earlier, in the ordering it is executed.

Implementations of REST Client will provide a default exception mapper designed to handle any `Response` where the status code is greater than or equal to 400. With such a response, the default mapper returns a `WebApplicationException`. The priority of the default exception mapper is the maximum value for an `Integer`, so it can be bypassed with a lower priority.

If the default exception mapper isn't wanted at all, it can be disabled by setting the `microprofile.rest.client.disable.default.mapper` configuration property to `true`.

Let's write an exception mapper to handle any errors related to an account not being found. For that there needs to be an exception to be thrown from the mapper:

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

There's nothing special about the exception, as the string parameter constructor is sufficient.

Now to write the mapper:

Listing 5.11 AccountRequestFilter

```
public class AccountExceptionMapper implements ResponseExceptionMapper
<AccountNotFoundException> {      ❶
  @Override
  public AccountNotFoundException toThrowable(Response response) {           ❷
    return new AccountNotFoundException("Failed to retrieve account");       ❸
  }

  @Override
  public boolean handles(int status, MultivaluedMap<String, Object> headers) {     ❹
    return status == 404;           ❺
  }
}
```

❶ Implement `ResponseExceptionMapper` for the `AccountNotFoundException` type.

❷ `toThrowable` takes the `Response` and converts it the appropriate exception type, in this case `AccountNotFoundException`.

❸ Create an instance of `AccountNotFoundException`.

❹ `handles` method provides a way to say whether the mapper is responsible for producing a `Throwable` based on the `Response`, or whether it shouldn't be called for it.

❺ Only handles a `Response` when the status code is `404`.

Without adding `@Priority` onto `AccountExceptionMapper`, the default priority of `5000` is

used.

To see the effect of the exception mapper, modify `TransactionResource` to capture the exception from the REST Client call:

Listing 5.12 TransactionResource

```
public class TransactionResource {
  ...
  @POST
  @Path("/{acctNumber}")
  public Map<String, List<String>> newTransaction(@PathParam("acctNumber")
  Long accountNumber, BigDecimal amount) {
    try {
      return accountService.transact(accountNumber, amount);       ❶
    } catch (Throwable t) {
      t.printStackTrace();
      Map<String, List<String>> response = new HashMap<>();
      response.put("EXCEPTION - " + t.getClass(), Collections.singletonList
      (t.getMessage()));       ❷
      return response;
    }
  }
}
```

❶ Wrap the REST Client call in a try-catch.

❷ Create a `Map` with information about the received exception to return as the response. This is to show the captured exception only, a production service should handle the exception in a more appropriate manner.

Rebuild and deploy the *Transaction Service* to Kubernetes:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

Now call the service with an account number that doesn't exist:

```
curl -H "Content-Type: application/json" -X POST -d "15.64"
http://192.168.64.4:31692/transactions/11
```

The headers returned should provide details of the exception:

```
{
  "EXCEPTION - class javax.ws.rs.WebApplicationException":["Unknown error,
    status code 404"]      ❶
}
```

❶ By default, the exception type received from the REST Client call is `WebApplicationException`. This is a result of the default exception mapper being active.

Let's modify `AccountService` to register the custom exception mapper:

Listing 5.13 AccountService

```
@RegisterProvider(AccountExceptionMapper.class)          ❶
public interface AccountService {
  ...

  @POST
  @Path("{accountNumber}/transaction")
  Map<String, List<String>> transact(@PathParam("accountNumber") Long accountNumber,
      BigDecimal amount) throws AccountNotFoundException;       ❷
}
```

❶ Registers the `AccountExceptionMapper` for handling exceptions.

❷ `transact` indicates it can return an `AccountNotFoundException`, enabling the exception mapper to work.

Time to see how the exception changes with the mapper registered. Re-deploy the *Transaction Service*:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

Now run the same request again:

```
curl -H "Content-Type: application/json" -X POST -d "15.64"
http://192.168.64.4:31692/transactions/11
```

The terminal should contain the response:

```
{
  "EXCEPTION - class io.quarkus.transactions.AccountNotFoundException":
  ["Failed to retrieve account"]      ❶
}
```

❶ The exception type received is now `AccountNotFoundException`.

SIDEBAR **Exercise for the reader**

Try out the different methods on `TransactionResource` to see when the exception mapper is, and is not, applied. Another exercise is to store the transactions into a local database for auditing.

5.4 Summary

- By adding `@RegisterRestClient` onto an interface that defines an external service, a CDI Bean representing the interface is available for injection with `@RestClient` to execute REST Client calls.
- Customization of interface behavior can be achieved with configuration keys that start with `[packageName].[className]/mp-rest/`. URL of the external service, which CDI Bean scope to use, what JAX-RS providers to register, and connection or read timeouts, are all items available for customization.
- When executing REST Client calls with services that may require time to execute, it is worth switching the return types to `CompletionStage` to enable asynchronous execution.
- Adding `@RegisterProvider`, with a JAX-RS provider class name as the value, onto an interface of an external service indicates the provider should be used with any REST Client calls that involve the interface.
- Implementing `ResponseExceptionMapper` to handle specific HTTP status codes and return a custom exception, making executing REST Client calls more like local method execution.

6 Application Health

This chapter covers:

- Application health, or lack thereof, in a traditional 3-tier Java monolithic application architecture
- MicroProfile Health and exposing application health
- Exposing Account Service and Transaction Service application health
- Using Kubernetes probes to address application health issues

The combination of Kubernetes and the microservices architecture have caused a fundamental shift in how developers create applications. What used to be dozens of large monolithic applications are now becoming hundreds (or thousands) of smaller, more nimble microservice instances. The more application instances that are running, the larger the odds of an individual application instance failing. The increased odds of failure could be a significant challenge in production if application health is not a first-class concern in Kubernetes.

Let's begin with a quick review of how monolithic applications running in application servers react to unhealthy applications.

6.1 Growing role of developers in application health

Many enterprise Java developers have experience with Java application servers, dating back to the late 1990s. During most of that time, developers created monolithic 3-tier applications with little awareness of exposing an application's health. From a developer's perspective, monitoring an application's health is the responsibility of system administrators whose livelihoods are to keep applications up and running in production.

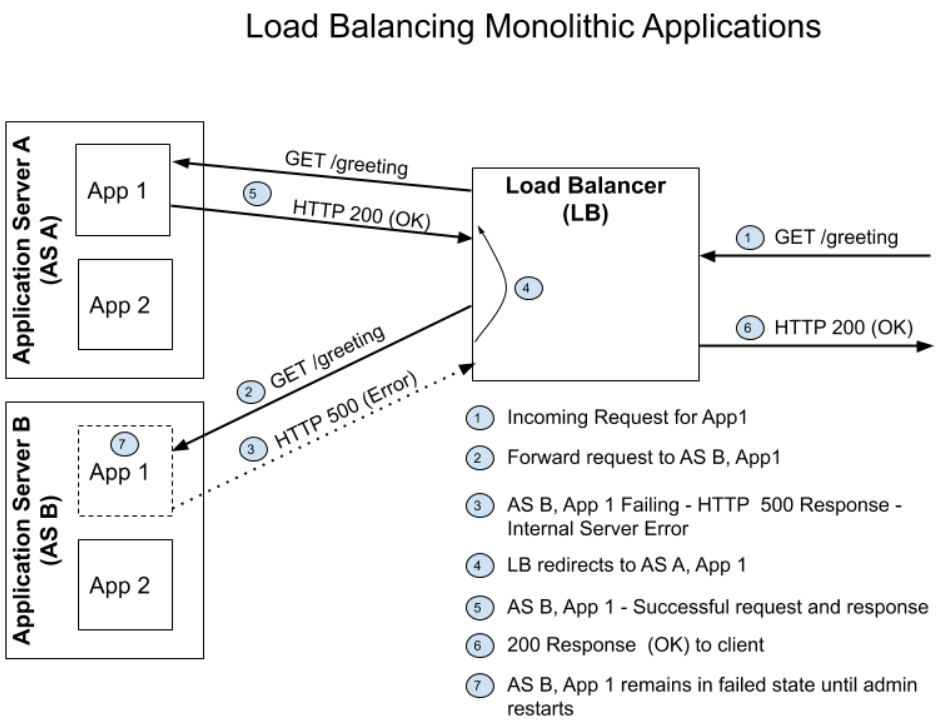


Figure 6.1 Traditional application server high availability architecture

Figure 6.1 shows the typical high-availability architecture of monolithic applications running in a traditional horizontally scaled application server configuration. There are some notable points to make about this architecture:

- **Load Balancer**. The load balancer's primary role is to balance load across multiple application instances for application scalability and availability. The load balancer will redirect traffic from a failing instance to a properly running instance. It is the responsibility of an administrator to ensure proper load balancer configuration.
- **Handling failed applications.** Addressing failed application was commonly dealt with manually. Because dealing with a failing application is a manual process, administrators prefer to conduct a root cause analysis to find the cause and address it, so they do not have to spend time on it again at a later date. Regardless, it is the administrator's responsibility to restart the failed application, which takes time and resources.
- **Roles and responsibilities.** In this scenario, developers have little to no direct role in application health in production. The developer's role is typically limited to helping diagnose issues to determine if the application is the root cause.
- **Minimal automation.** The only automation in this process is the load balancer recognizing the HTTP 500 error and pausing HTTP traffic to the failing application. Load balancers will also recover by occasionally sending traffic to detect a recovered server and resume traffic. Because there is no formal "health" contract between the application, the load balancer, and the application server, there is no automated way to determine if an application is failing.

Given the scale of microservices deployments, with hundreds to thousands of application instances, administrators cannot manually manage individual application instances at this scale. Developers can play a key role in significantly improving the overall health and efficiency of a production environment by reducing the need for manual intervention. More specifically, developers can proactively expose application health to Kubernetes. Kubernetes uses *probes* to check the exposed health and take corrective action if necessary. In the next section, we'll cover MicroProfile Health as an API to expose application heath, and follow that by covering how Kubernetes liveness and readiness probes can take corrective action.

6.2 MicroProfile Health

The MicroProfile community recognizes that modern Java microservices run on single application stacks, which in turn often run in containers. There is also recognition that modern container platforms have defined a formal health contract between the containerized application and the platform. With a formal contract in place, the platform can restart failing application containers and pause traffic to an containerized application that is not ready to accept traffic. The *MicroProfile Health* specification supports such a contract by defining the following:

- **Health endpoints.** Application health is accessible at the `/q/health/live` and `/q/health/ready` endpoints using REST calls.
- **HTTP status code.** The HTTP status code reflects the health status
- **HTTP response payload.** The JSON payload also provides status, along with additional health metadata and context.
- **Application Liveness.** Specifies whether an application is up and running properly.
- **Application Readiness.** Specifies whether an application is ready to accept traffic.
- **Application health API.** Expose application readiness and liveness based on custom application logic.

6.2.1 Liveness vs readiness

While it may not be apparent, there is a clear separation of concerns between liveness and readiness. The underlying platform makes an HTTP request to the `/q/health/live` endpoint to determine if it should restart the application container. For example, an application running out of memory can cause unpredictable behavior that may require a restart.

The underlying platform makes an HTTP request to the `/q/health/ready` endpoint to determine if an application instance is ready to accept traffic. If it is not ready, then the underlying platform will not send traffic to the application instance. An application can be "live," but not be "ready." For example, it may take time to pre-populate an in-memory cache or connect to an external service. During this time, the application is live and running properly, but may not be ready to receive external traffic because a service it depends on may not be running properly.

NOTE **Sending an HTTP request to the `/q/health` endpoint will return the combined liveness and readiness status. MicroProfile Health has deprecated this endpoint in favor of separate `/q/health/live` and `/q/health/ready` endpoints.**

6.2.2 Determining liveness and readiness status

The underlying platform has two means to determine status. It can check the HTTP status code or parse the JSON payload to obtain the status (*UP*, *DOWN*). Table 6.1 shows the correlation between the HTTP status code and JSON payload.

Table 6.1 Health endpoints, status codes, and JSON payload status

Health Check Endpoints	HTTP Status	JSON Payload Status
/q/health/live /q/health/ready	200	UP
/q/health/live /q/health/ready	503	DOWN
/q/health/live /q/health/ready	500	Undetermined *

* Request processing failed.

Figure 6.2 shows the flow of probe traffic on the left-hand side and the flow of application traffic on the right-hand side, in time sequence from top to bottom. The figure shows that after a period of unsuccessful attempts, the container will be restarted, and normal application traffic will resume until the probe detects the next health issue.

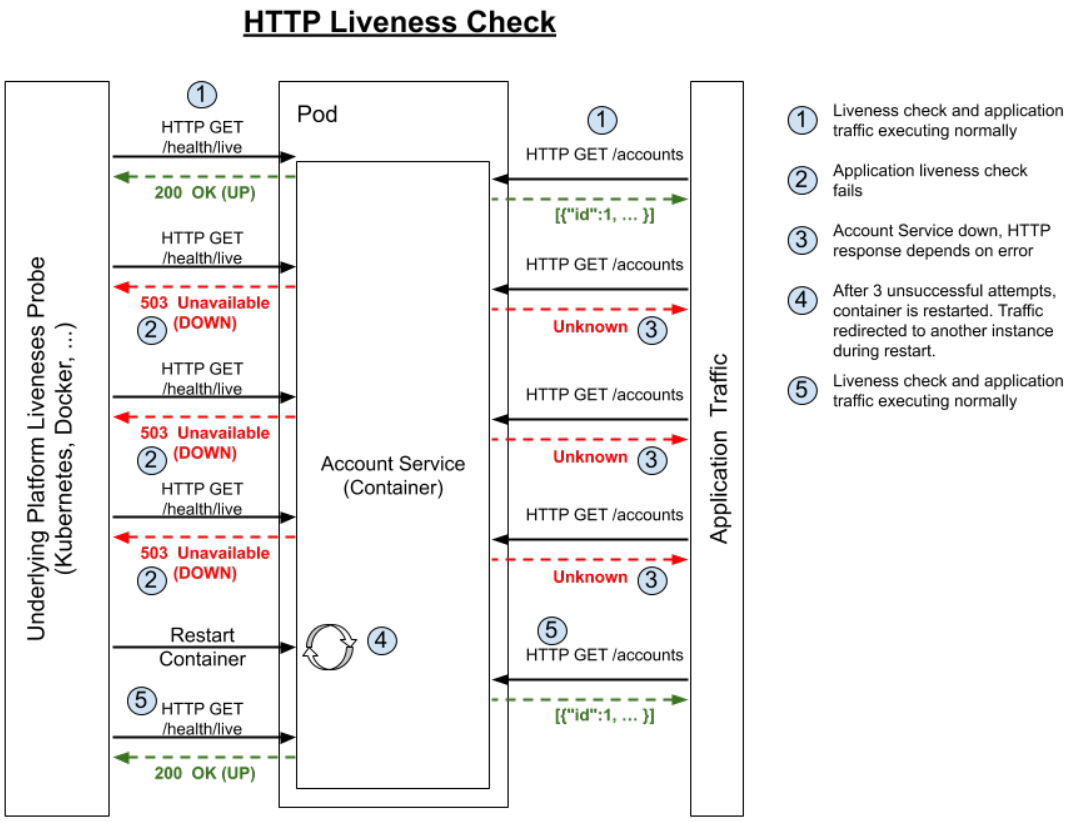


Figure 6.2 Liveness check and application traffic flow

Figure 6.3 shows a similar flow to figure 6.2, but instead represents a readiness health check. The figure shows that when a database connection is lost, the readiness check fails. The failure causes Kubernetes to redirect flow to another application instance until the database connection resumes.

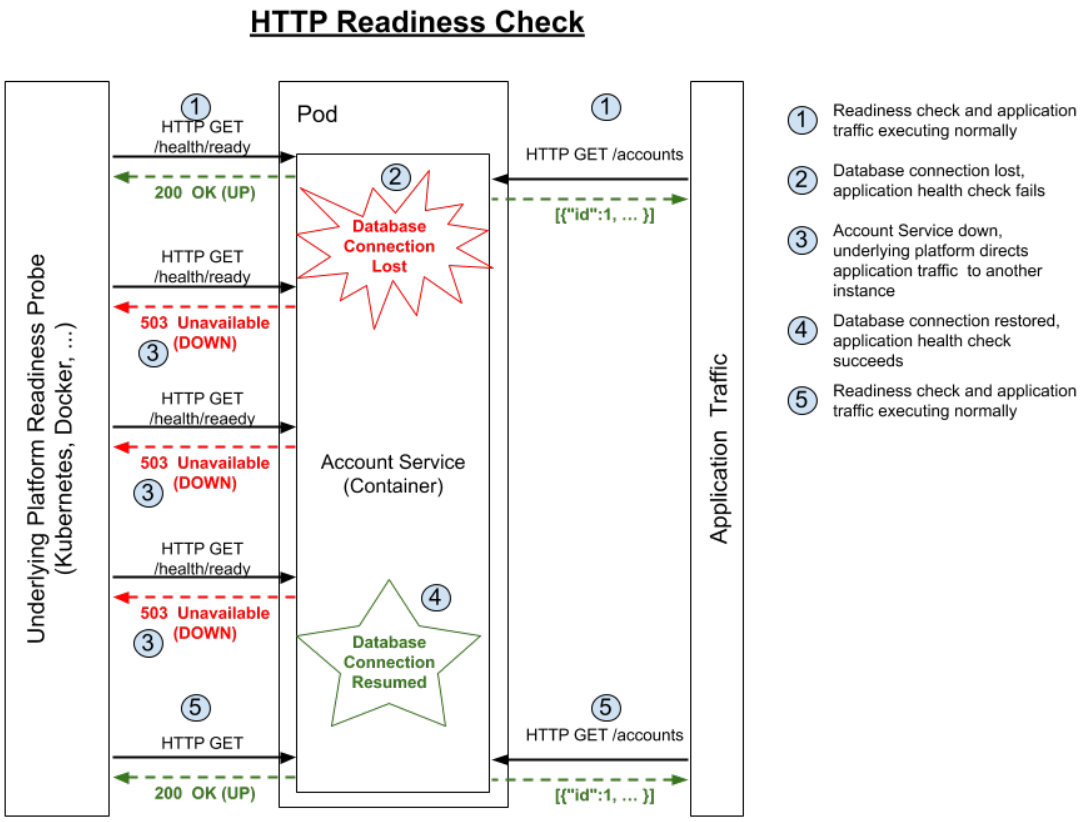


Figure 6.3 Readiness check and application traffic flow

A couple of notes about these figures. First, these use cases will be addressed shortly in code examples. Second, the probes are configurable and will also be covered shortly.

In fact, let's start coding now!

6.3 Getting started with MicroProfile Health

This chapter extends the *Account Service* and *Transaction Service* with application health logic. This logic provides Kubernetes enough metadata to take corrective action if necessary.

The *Account Service* uses a PostgreSQL database, which must be up and ready to accept requests. This can be done in a few steps:

1. **Check if PostgreSQL is running** in the Kubernetes cluster by running `kubectl get pods`. If the output does not contain text similar to `postgres-58db5b6954-27796`, then the database is not running.
2. **If the database is not running**, deploy the database by running `kubectl deploy -f postgresql_kubernetes.yml` from the top-level `chapter06` directory.
3. **Once the database is running**, forward local database traffic to the PostgreSQL pod in the Kubernetes cluster. To do this, run the command in Listing 6.1:

Listing 6.1 PostgreSQL Port Forwarding

```
# This will forward traffic until CTRL-C is pressed
kubectl port-forward service/postgres 5432:5432    ❶
```

❶ Forward traffic on localhost port 5432 to the kubernetes pod port 5432. During development, the *Account Service* will use localhost port 5432, which will forward traffic to the PostgreSQL pod.

With the database running, the next step is to start the *Account Service*. Install the parent `pom.xml`, and then start the *Account Service* as shown in Listing 6.2:

Listing 6.2 Install parent pom and build artifacts

```
mvn clean install -DskipTests          ❶
cd account-service
mvn quarkus:dev                        ❷
```

❶ Install parent `pom.xml`

❷ Start account-service in developeer mode

Check the health endpoint to determine the *Account Service* health status using the command in Listing 6.3:

Listing 6.3 Check health endpoint availability

```
curl -i localhost:8080/q/health/live
```

The resulting output in Listing 6.4 shows that Quarkus does not include a liveness health check endpoint by default:

Listing 6.4 Quarkus health endpoint unavailable

```
HTTP/1.1 404 Not Found
Content-Length: 0
Content-Type: application/json
```

Account Service requires an additional Quarkus extension for MicroProfile Health support. Using the code in Listing 6.5, add the `quarkus-smallrye-health` extension.

Listing 6.5 Add MicroProfile Health support using the smallrye-health extension:

```
mvn quarkus:add-extension -Dextensions="quarkus-smallrye-health"
```

Because Quarkus is running in developer mode, the extension will be loaded automatically.

6.3.1 Account Service MicroProfile Health Liveness

With MicroProfile Health support loaded, check the endpoint again with `curl -i localhost:8080/q/health/live`. This time, as seen in Listing 6.6, the result is very different:

Listing 6.6 Liveness health check output

```
HTTP/1.1 200 OK                                          ❶
content-type: application/json; charset=UTF-8
content-length: 46

{                                                        ❷
    "status": "UP",                                      ❸
    "checks": [                                          ❹
    ]
}
```

❶ A HTTP response code of 2XX means the service is up and running as expected.

❷ The HTTP response is a JSON-formatted payload

❸ The overall status of the service is *UP*. An HTTP response code of 2XX will always result in an *UP* status.

❹ The `checks` array contains named health checks and their status. The array is empty, meaning there are currently no custom liveness health checks.

While the output is relatively simple, there is some useful context to cover. First, without writing a custom health check, MicroProfile Health requires a default status of *UP*.

Second, the HTTP status maps to the JSON payload status. A 200 HTTP status maps to *UP*, and a 5XX HTTP status maps to *DOWN*. The underlying platform has the option of considering the remainder of the JSON payload for additional context before taking corrective action.

Last, the *MicroProfile Health* specification requires the JSON payload return two items. First, it must return a `status` of *UP* or *DOWN*. Second, it must return an array of `checks` that aggregate all liveness health checks. If one or more liveness health checks is *DOWN*, then the overall status for the health check is *DOWN*.

Having seen a liveness health check, it's time to create a custom liveness health check.

6.3.2 Creating an Account Service liveness health check

To learn the API, let's start by creating a liveness health check that always returns *UP* as seen in Listing 6.7:

Listing 6.7 AlwaysHealthyLivenessCheck.java

```
@ApplicationScoped                                                    ❶
@Liveness                                                             ❷
public class AlwaysHealthyLivenessCheck implements HealthCheck {      ❸
    @Override
    public HealthCheckResponse call() {                               ❹
        return HealthCheckResponse                                    ❺
                .named("Always live")                                 ❻
                .withData("time", String.valueOf(new Date()))         ❼
                .up()                                                 ❽
                .build();
    }
}
```

❶ A HealthCheck must be a CDI bean. It is annotated with @ApplicationScoped so a single CDI bean instance is created.

❷ This is a liveness health check.

❸ Health checks must implement the HealthCheck interface, which requires the call() method be implemented.

❹ The call() method is invoked whenever the `/q/health/live` endpoint is invoked, and must return a HealthCheckResponse object.

❺ A HealthCheckResponse object is created using a builder pattern.

❻ Each health check has a name which should reflect the intent of the health check.

❼ Contextual data can be added to the health check in the form of key/value pairs. In this health check, a time/date stamp is returned

❽ The state returned is always *UP*

To test the liveness health check, run `curl localhost:8080/q/health/live`. The results should match Listing 6.8.

Listing 6.8 Liveness health check output

```
HTTP/1.1 200 OK                                                       ❶
content-type: application/json; charset=UTF-8
content-length: 220


{
    "status": "UP",                                                   ❷
    "checks": [
        {                                                             ❸
            "name": "Always live",
            "status": "UP",                                           ❹
            "data": {
                "time": "Mon Sep 28 23:56:38 PDT 2020"
            }
        }
    ]
}
```

❶ The HTTP status returns OK because the application status is *UP*

❷ The JSON health status *UP* matches the HTTP response status

❸ The remainder of the JSON reflects the values defined in the HealthCheckResponse object.

❹ The `AlwaysHealthyLivenessCheck` status is *UP*. Therefore the overall status is also *UP*. If any individual health check status is *DOWN*, then the overall status is *DOWN*.

Next, let's get a firmer understanding of application readiness.

6.3.3 Account Service MicroProfile Health Readiness

With a sound understanding of application liveness, the next step is to check application readiness using `curl -i http://localhost:8080/q/health/ready`. Interestingly, the output in Listing 6.9 may be a bit unexpected.

Listing 6.9 Account Service is ready to accept traffic

```
HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 150

{
    "status": "UP",
    "checks": [                                                    ❶
        {
            "name": "Database connections health check",
            "status": "UP"
        }
    ]
}
```

❶ Database connection is operating properly

The output includes a pre-configured database readiness health check. If the database becomes unavailable, then the *Account Service* will return an HTTP 503 status code, and Kubernetes will not forward traffic to the service. What is providing the database readiness health check? The Hibernate ORM with Panache extension automatically adds the Agroal datasource extension as an application dependency. The Agroal datasource extension provides the readiness health check. All relational databases supported by Quarkus will benefit from a readiness check.

NOTE **As a general rule of thumb, Quarkus extensions that provide client connectivity to backend services have built-in readiness health checks, including relational and non-relational databases, messaging systems like Kafka and JMS, Amazon services like S3 and DynamoDB, and more.**

6.3.4 Disabling vendor readiness health checks

Sometimes it is preferable to disable vendor readiness health checks like the Agroal readiness health check. For example, instead of pausing traffic to an application, the application can continue with fallback logic if the backend service is unreachable. There are two means to disable vendor readiness health checks.

First, by setting the MicroProfile Health `mp.health.disable-default-procedures` to `true`, all vendor health checks are disabled. Disabling all vendor readiness health checks is a coarse-grained approach.

Second, Quarkus readiness health checks can be disabled on an extension-by-extension basis. To disable a Quarkus extension's readiness health check, use `quarkus.<client>.health.enabled=false`, where `<client>` is the extension to disable. For example, to disable the datasource health check provided by the Agroal extension, use `quarkus.datasource.health.enabled=false`. The Quarkus extension guides document the relevant property name.

6.3.5 Creating a readiness health check

Creating a readiness health check is nearly identical to creating the liveness health check. The only differences are using the @Readiness annotation instead of the @Liveness annotation and the business logic to determine readiness. Since the *Account Service* already has a built-in database readiness health check, let's create a readiness health check on the *Transaction Service* that checks the *Account Service* readiness. If it is not ready, then the *Transaction Service* will return *DOWN* as well. First, we'll need to add the health extension to the *Transaction Service*.

Listing 6.10 Add the health extension to the Transaction Service.

```
cd transaction-service
mvn quarkus:add-extension -Dextensions="quarkus-smallrye-health"
```

With the health extension added, create the `AccountHealthReadinessCheck` class in Listing 6.11:

Listing 6.11 AccountHealthReadinessCheck.java

```
@Readiness                                                          ❶ ❷
public class AccountHealthReadinessCheck implements HealthCheck {
    @Inject
    @RestClient
    AccountService accountService;                                  ❸

    BigDecimal balance;

    @Override
    public HealthCheckResponse call() {
        try {
            balance = accountService.getBalance(999999999L);            ❹
        } catch (WebApplicationException ex) {
            // This class is a singleton, so clear last request's balance
            balance = new BigDecimal(Integer.MIN_VALUE);

            if (ex.getResponse().getStatus() >= 500) {              ❺
                return HealthCheckResponse
                        .named("AccountServiceCheck")
                        .withData("exception", ex.toString())
                        .down()
                        .build();
            }
        }

        return HealthCheckResponse                                  ❻
                .named("AccountServiceCheck")
                .withData("balance", balance.toString())
                .up()
                .build();
    }
}
```

❶ This is a readiness health check

❷ Quarkus will automatically make this a @Singleton CDI bean when no scope is provided While not a portable feature, this does tidy up the code a bit and delivers some Quarkus developer joy

❸ Inject an instance of the AccountService Rest Client, which will be used to invoke the *Account Service*

❹ Test *Account Service* availability by invoking an endpoint and getting the balance of a special "Health Check" account.

❺ Only return a *DOWN* status if there is an 5XX HTTP status code, meaning the service was unable to respond to a valid HTTP request. All other status codes imply that the service is responding to requests.

❻ Return an *UP* status along with the balance.

Add a readiness health check account to the accounts table as shown in Listing 6.12:

Listing 6.12 Add test account to src/main/resources/import.sql

```
INSERT INTO account(id, accountNumber, accountStatus, balance, customerName,
customerNumber) VALUES (9, 999999999, 0, 999999999.01, 'Readiness HealthCheck',
99999999999);
```

Update `application.properties` with the properties in listing 6.13:

Listing 6.13 Additional Transaction Service properties

```
%dev.quarkus.http.port=8088                                                    ❶
%dev.io.quarkus.transactions.AccountService/mp-rest/url=http://localhost:8080  ❷
```

❶ The account service is already listening on port 8080, so the transaction service will listen on port 8088 to avoid a port conflict when running in developer mode.

❷ Because the *Account Service* is also running locally on port 8080, the Rest Client must access the *Account Service* on `localhost` when running in developer mode.

Within a new terminal window, start the *Transaction Service* as shown in Listing 6.14:

Listing 6.14 Start the Transaction Service

```
mvn compile quarkus:dev \
    -Ddebug=5006                  ❶
```

❶ Quarkus defaults the debug port to 5005, which is the debug port for the *Account Service* that is already running. Set the *Transaction Service* debug port to 5006.

To test `AccountHealthReadinessCheck`, run `curl -i localhost:8088/q/health/ready` to see the output in Listing 6.15:

Listing 6.15 ReadinessCheck is accepting traffic

```
HTTP/1.1 200 OK                                                    ❶
content-type: application/json; charset=UTF-8
content-length: 234

{
    "status": "UP",
    "checks": [
        {
            "name": "AccountServiceCheck",                         ❷
            "status": "UP",
            "data": {
                "balance": "999999999.01"
            }
        }
    ]
}
```

❶ The overall HTTP status is 200, meaning the service is ready to accept traffic

❷ The *Account Service* is ready to accept traffic.

Currently, there are three services up and running - PostgreSQL, the *Account Service*, and the *Transaction Service*. Figure 6.4 shows the service health readiness status of these services.

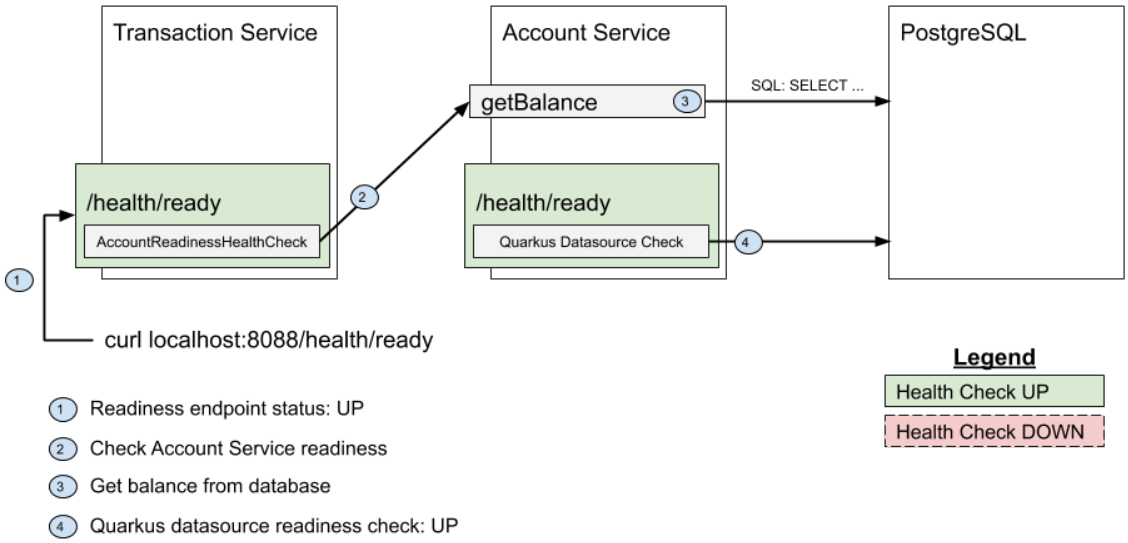


Figure 6.4 Service readiness health check status

Next, stop the port forwarding started in 6.1 by pressing CTRL-C in the terminal running the `kubectl port-forward …` command. Check the readiness endpoint again by running `curl -i localhost:8088/q/health/ready`. The result, shown in Listing 6.16, shows that the transaction service is *DOWN* and not ready.

Listing 6.16 AccountHealthReadinessCheck is DOWN and not ready

```
HTTP/1.1 503 Service Unavailable                                    ❶
content-type: application/json; charset=UTF-8
content-length: 276


{
    "status": "DOWN",
    "checks": [
        {
            "name": "AccountServiceCheck",
            "status": "DOWN",
            "data": {                                               ❷
                "exception": "javax.ws.rs.WebApplicationException: Unknown error,
                status code 500"
            }
        }
    ]
}
```

❶ The *Transaction Service* is not ready

❷ Invoking the *Account Service* results in an exception, causing the *Transaction Service* to be down

As a result of the database being down, there is a cascading failure of the *Account Service* not being ready, followed by the *Transaction Service* not being ready. This is outlined in Figure 6.5.

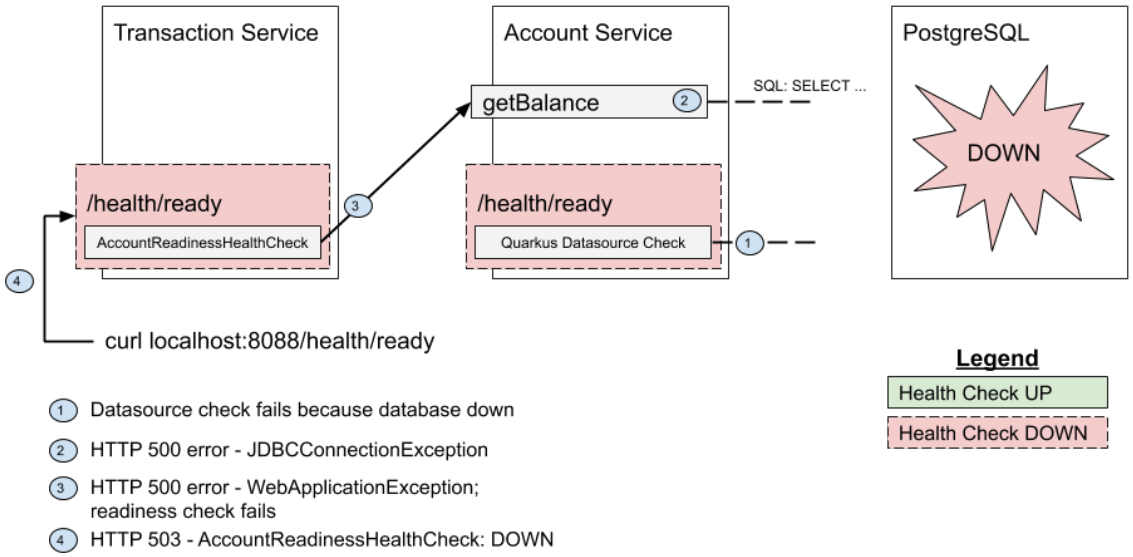


Figure 6.5 Service readiness cascading failure

NOTE **The MicroProfile Fault Tolerance chapter discusses how to avoid cascading failures.**

The next couple of sections will cover Quarkus-specific health features, and then the services will be deployed to Kubernetes

6.3.6 Quarkus Health Groups

Quarkus extends the MicroProfile Health feature set by adding *Health Groups*. A health group allows for custom grouping of health checks. Health groups are useful for monitoring health checks that do not impact container access (readiness) and container lifecycle (liveness) because they exist at a separate REST endpoint. These endpoints are likely not monitored directly by Kubernetes liveness or readiness probes but instead by 3rd party or custom tooling. For example, external tooling can probe a health group's endpoint to monitor informational, non-critical health checks.

To create a health group, use `@HealthGroup("group-name")`. Listing 6.17 shows an example of a health check group.

Listing 6.17 CustomGroupLivenessCheckHealth.java health check group example

```
@ApplicationScoped
@HealthGroup("custom")                                          ❶
public class CustomGroupLivenessCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.up("custom liveness");       ❷
    }
}
```

❶ Specify the custom health group.

❷ Similar to the `AlwaysHealthyReadinessCheck`, the `CustomGroupLivenessCheck` always returns *UP*. In a real-world scenario, this health group check would utilize business logic to determine the health status.

All health groups can be accessed at `/q/health/group`, and a specific health check group can be accessed at `/q/health/group/<group>`, where `group` is the health group name. See Listing 6.18 for example output when running `curl -i http://localhost:8088/q/health/group/custom` to access the *custom* health group.

Listing 6.18 Health check group output

```
HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 132


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

Having just covered a Quarkus-specific feature, let's cover one more Quarkus-specific application health feature, the Quarkus Health UI, before moving on to Kubernetes deployments.

6.3.7 Displaying the Quarkus Health UI

As an option to viewing the JSON output, Quarkus includes a helpful Health UI for viewing health status while developing an application and is not intended to be a production tool. To enable the UI, as seen in Figure 6.6, add `quarkus.smallrye-health.ui.enable=true` to the application.properties file. The Health UI can also be auto-refreshed at regular intervals by clicking on the gear icon in the Health UI title bar and setting the refresh interval. This example shows the health check UI (available at `http://localhost:8080/q/health-ui`) enabled on the *Account Service* without access to the PostgreSQL database.

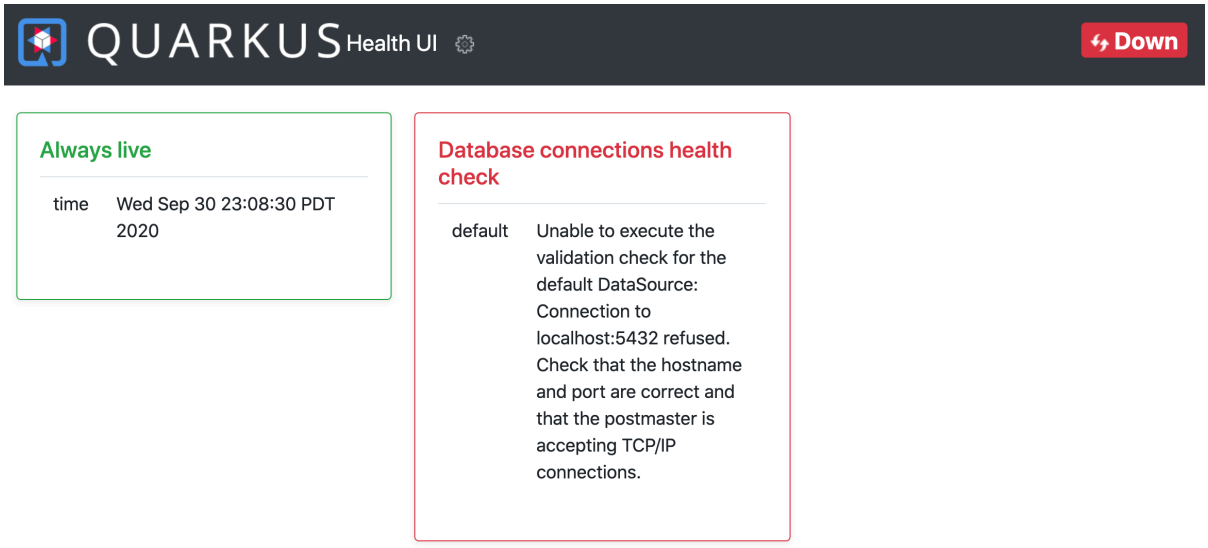


Figure 6.6 Health check UI enabled on Account Service

The UI can also be included in production builds, like native binaries and jar deployments, by adding the property `quarkus.smallrye-health.ui.always-include=true` to application.properties.

It is time to put all of this newfound health check knowledge to work by adding Kubernetes health check probes and deploying to Kubernetes.

6.4 Kubernetes Liveness and Readiness probes

Kubernetes is one of the underlying platforms that offer liveness and readiness health check probes as a built-in capability. However, they need to be enabled and configured. Table 6.2 describes Kubernetes health check probe configuration parameters. The parameters are configured with Quarkus properties in application.properties.

Table 6.2 Kubernetes health check probe configuration parameters

Kubernetes Probe Parameter	Quarkus Property	Description and Quarkus defaults
initialDelaySeconds	• quarkus.kubernetes.liveness-probe.initial-delay • quarkus.kubernetes.readiness-probe.initial-delay	The amount of time to wait before starting to probe. Defaults to 0 seconds.
periodSeconds	• quarkus.kubernetes.liveness-probe.period • quarkus.kubernetes.readiness-probe.period	Probe interval. Defaults to 30 seconds.
timeout	• quarkus.kubernetes.liveness-probe.timeout • quarkus.kubernetes.readiness-probe.timeout	Amount of time to wait for probe to complete. Defaults to 10 seconds.
successThreshold	• quarkus.kubernetes.liveness-probe.success-threshold • quarkus.kubernetes.readiness-probe.success-threshold	Minimum consecutive successful probes to be considered successful after having failed. Defaults to 1. Must be 1 for liveness.
failureThreshold	• quarkus.kubernetes.liveness-probe.failure-threshold • quarkus.kubernetes.readiness-probe.failure-threshold	Retry failureThreshold times before giving up. Giving up on a liveness probe will restart container. Giving up on a readiness probe will pause traffic to container. Defaults to 3.

NOTE See the Quarkus Kubernetes and OpenShift Extension documentation [19] for additional liveness and readiness probe properties.

The Quarkus health extension generates the Kubernetes probe YAML automatically. Listing 6.19 is a snippet of the liveness probe YAML that was generated automatically in `target/kubernetes/minikube.yaml`.

Listing 6.19 Generated liveness probe YAML

```
# ...
        livenessProbe:
          failureThreshold: 3
          httpGet:                    ❶
            path: /q/health/live      ❷
            port: 8080                ❸
            scheme: HTTP              ❹
          initialDelaySeconds: 0
          periodSeconds: 30
          successThreshold: 1
          timeoutSeconds: 10
# ...
```

❶ Probe health endpoint using HTTP GET

❷ Health path to probe

❸ Port to probe

❹ Probe using HTTP (vs HTTPS)

> **NOTE** **Pods can have more than one container. Liveness and readiness probes are defined per container. Therefore probes restart and pause traffic to individual containers within the pod and not the pod as a whole.**

6.4.1 Customizing health check properties

The probe parameter listed in Table 6.2 specify reasonable defaults. Health check probes can be customized to reflect the specific needs of the business and the application. For example, from a business perspective, probes can check business-critical applications at a more frequent interval to more rapidly detect and resolve potential issues. On the other hand, some applications take longer to start and should have a higher `initialDelaySeconds` setting. Determining proper probe settings may take a bit of "trial-and-error" testing, but accepting the default probe property values is a good place to start.

To make developing probes easier, add the properties shown in Listing 6.20 to application.properties of both the *Account Service* and the *Transaction Service*. The intent is to encounter liveness and readiness issues sooner to make the round-trip coding faster. Do not use these values in production if they do not meet the needs of the application.

Listing 6.20 Override probe defaults for quicker round-trip development and testing

```
# Health Probe configuration                                    ❶
quarkus.kubernetes.liveness-probe.initial-delay=10
quarkus.kubernetes.liveness-probe.period=2
quarkus.kubernetes.liveness-probe.timeout=5

quarkus.kubernetes.readiness-probe.initial-delay=10
quarkus.kubernetes.readiness-probe.period=2
quarkus.kubernetes.readiness-probe.timeout=5
```

❶ The properties will generate YAML similar to those in Listing 6.19, but with the specified values.

With updated health check properties in place, the next step is to deploy the updated services to Kubernetes and see liveness probes (container restarts) and readiness probes (traffic pauses) in action.

6.4.2 Deploying to Kubernetes

Before deploying to Kubernetes, set the docker registry to the instance running in Minikube. Generated container images will now be pushed directly into the Kubernetes docker registry. With the docker registry set, deploy to Kubernetes and then track the deployment. See Listing 6.21 for steps.

Listing 6.21 Deploy Account Service to Kubernetes

```
eval $(minikube -p minikube docker-env)                                    ❶

# Run this command in the chapter top-level directory
mvn clean package -DskipTests -Dquarkus.kubernetes.deploy=true             ❷

# Run next command in a separate terminal window, and leave running
kubectl get pods -w                                                        ❸
```

❶ Use the docker registry running in Minikube. Alternatively, run `minikube docker-env` and set the environment variables manually.

❷ Deploy the *Account Service* and *Transaction Service* to Kubernetes

❸ Follow the deployment by watching pod lifecycles

See Listing 6.22 to see the output of the `kubectl get pods -w` command.

Listing 6.22 Pod status terminal window output

```
NAME                                READY   STATUS              RESTARTS   AGE    ❶
...
account-service-68f7c4779c-jpggz    0/1     Pending             0          0s     ❷
account-service-68f7c4779c-jpggz    0/1     ContainerCreating   0          0s     ❸
account-service-68f7c4779c-jpggz    0/1     Running             0          3s     ❹
transaction-service-5fb7f69496-d86sg   0/1     Pending             0          0s
transaction-service-5fb7f69496-d86sg   0/1     ContainerCreating   0          0s
transaction-service-5fb7f69496-d86sg   0/1     Running             0          2s
account-service-68f7c4779c-jpggz    1/1     Running             0          13s    ❺
transaction-service-5fb7f69496-d86sg   1/1     Running             0          15s    ❻
```

❶ The *READY* column identifies the number of containers in the pod ready to accept traffic. *0/1* means zero of one container is ready. *1/1* means one of one container is ready. The *RESTARTS* column is incremented each time a container restarts

❷ The pod and its containers are scheduled to be created on a node in the cluster

❸ Kubernetes is creating the pod and its containers. This includes downloading the container image from an image registry like Docker Hub

❹ The container has been created, is starting, but is not yet ready to accept traffic

❺ The container is running and ready to accept traffic

❻ There are equivalent steps for the *Transaction Service*. Both services should start successfully

IMPORTANT **There may be container restarts during the deployment. Deploying multiple services with minimal CPU cores allocated Minikube may result in a service starting to surpass the `initial-delay` setting. One or two restarts is possible until the deployment assumes a steady state. If the number of restarts for a pod surpasses four or five, then it will be time to troubleshoot with commands like `kubectl logs <POD_NAME>`.**

To simplify accessing the *Transaction Service* URL, store the service URL in an environment variable using the command in Listing 6.23:

Listing 6.23 Get transaction service URL

```
export TRANSACTIONS_URL=$(minikube service --url transaction-service)    ❶
```

❶ Store the `transaction-service` URL in the `TRANSACTIONS_URL` environment variable

Next, verify the *Transaction Service* is healthy by running `curl -i $TRANSACTIONS_URL/q/health/live`, with the resulting HTTP status shown Listing 6.24.

Listing 6.24 Output of `curl -i $TRANSACTIONS_URL/q/health/live`

```
HTTP/1.1 200 OK                                    ❶
content-type: application/json; charset=UTF-8
content-length: 411
...
```

❶ Only the HTTP status code is shown in this listing.

Last, test that the *Transaction Service* is ready with `curl -i $TRANSACTIONS_URL/q/health/ready`. The output should be identical to Listing 6.15, returning an HTTP status of 200 and a JSON payload status of *UP*. Figure 6.7 shows the flow of readiness checks between services.

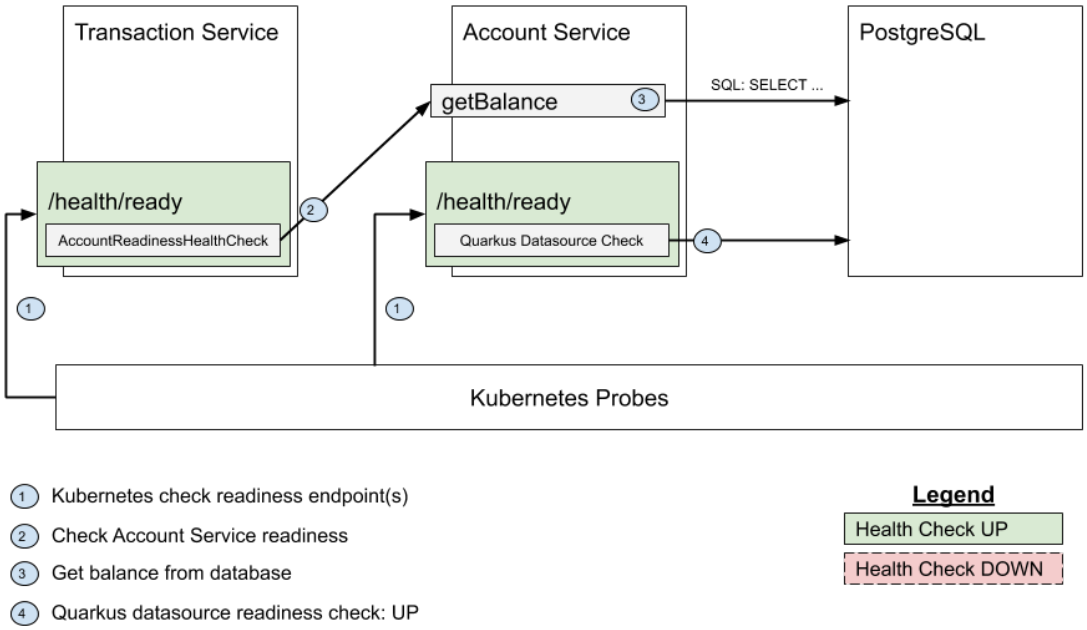


Figure 6.7 Service readiness health check status in Kubernetes

6.4.3 Testing the readiness health check in Kubernetes

With the healthy services up and running, let's introduce a readiness failure. An easy way of doing this is to scale down the number of PostgreSQL instances to zero, so the datasource health check fails. It is helpful to have the pod status terminal window created in Listing 6.21 easily viewable when running the commands in this chapter to track the pod lifecycle.

Run the command in Listing 6.25 to scale down the number of PostgreSQL instances to zero.

Listing 6.25 Scale database to zero instances (replicas)

```
kubectl scale --replicas=0 deployment/postgres
```

The pod status terminal window, shown in Listing 6.26 will update to show the pods terminating:

Listing 6.26 Output of `kubectl get pods -w` when scaling to zero pods

```
NAME                                READY   STATUS        RESTARTS   AGE
...
postgres-58db5b6954-2pg7x           1/1     Terminating   0          13m    ❶
postgres-58db5b6954-2pg7x           0/1     Terminating   0          13m    ❷
account-service-68f7c4779c-jpggz    0/1     Running       0          7m59s  ❸
transaction-service-5fb7f69496-d86sg 0/1    Running       0          7m50s  ❹
```

❶ postgres pod is ready, but is terminating

❷ postgres pod is no longer ready, and is *Terminating* or has been terminated

❸ The *Account Service* pod is no longer ready (*0/1*). Its readiness health check status is *DOWN* because the database is *DOWN*. The pod is still running but is not accepting traffic.

❹ The *Transaction Service* pod is no longer ready (*0/1*). Its readiness health check status is *DOWN* because the *Account Service* is not accepting traffic. The pod is still running but is not accepting traffic.

Access the *Transaction Service* readiness endpoint with `curl -i $TRANSACTIONS_URL/q/health/ready` and notice there is a *Connection refused* message. By scaling down the number of PostreSQL instances to zero, the probe causes a pause to the *Account Service* container traffic, which paused traffic to the *Transaction Service*, including its `/q/health/ready` endpoint. Figure 6.8 shows the cascading service failure in kubernetes.

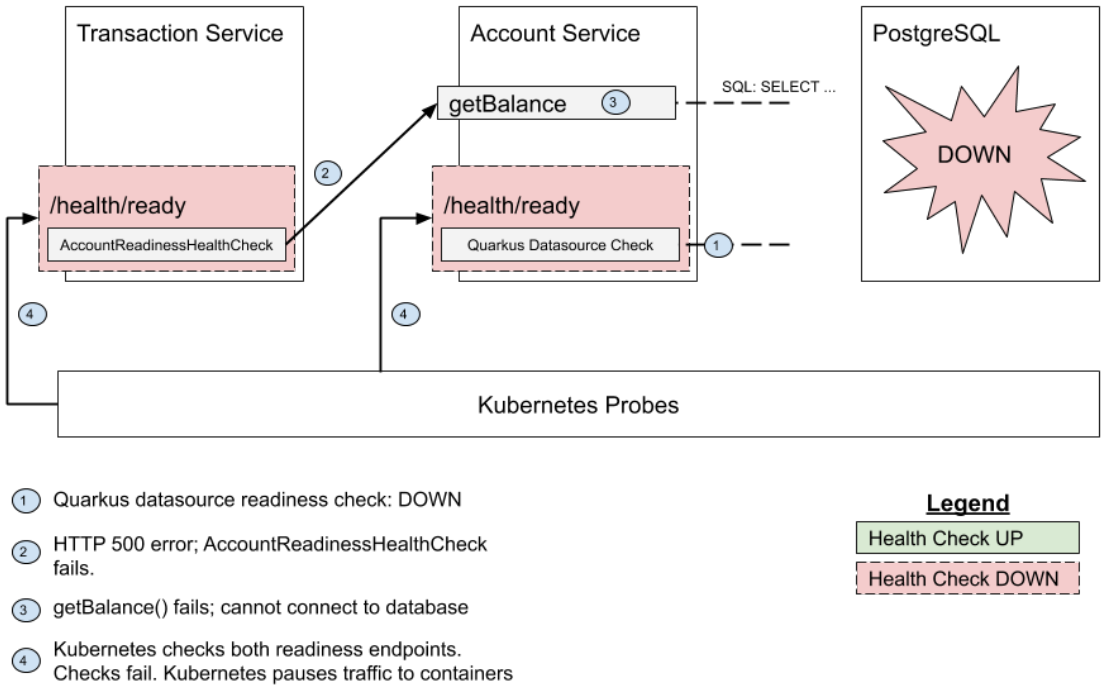


Figure 6.8 Service readiness health check cascading failure in Kubernetes

To resume back towards a healthy status, restart the database by running the command in Listing 6.27.

Listing 6.27 Scale database to one instance

```
kubectl scale --replicas=1 deployment/postgres
```

By scaling the database down to zero instances and then scaling it back up to one instance, the database contents are lost because the database schemas and data are not ephemeral in the current configuration. The easiest way to correct this is to create a new *Account_Service* instance to

regenerate the tables and re-populate the database. Real-world production deployments would not regenerate tables and re-populate databases every time a pod is created. However, it is helpful as a learning aid in this case. To add another *Account Service* instance, run the command in Listing 6.28. Note the output in the pod status terminal window in Listing 6.29.

Listing 6.28 Scale account-service to two instances

```
kubectl scale --replicas=2 deployment/account-service
```

Listing 6.29 Pod status output when scaling to 2 instances

```
NAME                                READY   STATUS              RESTARTS   AGE
...
account-service-68f7c4779c-bf458    0/1     Pending             0          1s    ❶
account-service-68f7c4779c-bf458    0/1     ContainerCreating   0          1s    ❷
account-service-68f7c4779c-bf458    0/1     Running             0          2s    ❸
account-service-68f7c4779c-bf458    1/1     Running             0          12s   ❹
transaction-service-5fb7f69496-d86sg 1/1    Running             0          23m   ❺
```

❶ Kubernetes is scheduling the pod creation

❷ The pod is being created

❸ The pod container is running but is not yet ready

❹ The pod is ready to service traffic

❺ The new *Account Service* instance connects to the database and inserts the special "Health Check" account. This results in the *Transaction Service* readiness health check state changing to *UP* and ready to accept traffic.

Of course, a real production scenario would include a database with persistent configuration and data, so this step to create two instances would typically be unnecessary.

Verify the *UP* status by running `curl -i $TRANSACTIONS_URL/q/health/ready`.

6.5 Summary

This chapter discusses how Kubernetes improves upon traditional application server handling of improperly running applications. It shows a step-by-step approach to creating and testing health checks locally and in Kubernetes.

The automation introduced with readiness and liveness health checks significantly reduce the need for root cause analysis to deal with one-off issues and edge cases. The up-front effort to treat application health as a first-class concern can save developers and administrators significant time and effort in the long run.

To summarize:

- Traditional application servers require manual intervention to react to failure. Manual

intervention does not scale well in an environment with hundreds to thousands of containers.

- Combining the automation of Kubernetes health check probes with developer health checks can provide a more responsive and efficient Kubernetes cluster and microservices architecture.
- Kubernetes can pause traffic to a container that is unable or not yet ready to accept traffic and resume traffic when ready, based on developer guidance through health checks.
- Kubernetes can restart a failing or failed container, based on developer guidance through health checks
- Developers can create readiness and liveness health checks to provide more accurate application-specific health status.

7 Resilience Strategies

This chapter covers:

- The importance of building resilient applications
- MicroProfile Fault Tolerance strategies
- When to apply each fault tolerance strategy
- How to apply each fault tolerance strategy
- How to configure and disable fault tolerance annotations using properties

Application robustness is critically important in a microservices architecture, where there are many service interdependencies. A microservices architecture can have many service interdependencies. A service susceptible to failure can negatively impact other services. This chapter covers using resilience patterns to improve application robustness to maintain overall health.

7.1 Resilience Strategies Overview

Services will eventually experience downtime, whether planned or unplanned. A service can reduce its downtime using resilience strategies when the services it depends on are unreliable or unavailable.

Quarkus offers its resilience strategies using the MicroProfile Fault Tolerance APIs. These annotation-based APIs are applied to classes or methods, standalone or in combination. Table 7.1 lists the available fault tolerance annotations.

Table 7.1 MicroProfile Fault Tolerance annotations

Annotation	Description
@Asynchronous	Execute a method using a separate thread
@Bulkhead	Limits the number of concurrent requests
@CircuitBreaker	Avoid repeated failures
@Fallback	Use alternative logic when a method completes exceptionally (throws an exception)
@Retry	Retries a method call when the method completes exceptionally
@Timeout	Prevents method from executing longer than a specified amount of time

7.2 Executing a method under a separate thread with @Asynchronous

A service may have to call a slow-responding remote service. Instead of blocking a worker thread by waiting for a response, the `@Asynchronous` annotation uses a separate thread to invoke the remote service to increase concurrency and throughput. See Listing 7.1 for an example:

Listing 7.1 @Asynchronous example

```
@Asynchronous                                    ❶
public String invokeLongRunningOperation() {
 callLongRunningRemoteService();
}
```

❶ Use a thread from a separate thread pool to execute a blocking operation.

This book does not advocate using the `@Asynchronous` annotation with Quarkus and will not cover the annotation in detail. The `@Asynchronous` annotation is for runtimes that make heavy use of threads and thread pools to achieve higher concurrency and throughput, like Jakarta EE runtimes. Quarkus uses a non-blocking network stack and event loop execution model based on Netty and Eclipse Vert.x. It can achieve higher concurrency and throughput using its inherent asynchronous and reactive APIs while using less RAM and CPU overhead.

For example, the Quarkus *RestEASY Reactive* extension enables the use of JAX-RS annotations and handles requests directly on the IO thread. Developers can use the APIs they already know while benefiting from the throughput typically reserved for asynchronous runtimes like Vert.x.

7.3 Constraining concurrency with bulkheads

The bulkhead concept comes from shipbuilding, which constrains a compromised section of a ship's hull by closing bulkhead doors to isolate the incoming water. The Bulkhead architectural pattern applies this concept to prevent a failure in one service from cascading to another service by limiting the number of concurrent method invocations.

For example, a service may make remote calls to a slow-executing backend service. In

thread-per-request runtimes like traditional Java EE and Spring, each remote invocation to a slow service consumes memory resources and threads from a thread pool in the calling service, eventually overcoming available resources. As with the `@Asynchronous` annotation, this is less of an issue with Quarkus' *RestEASY Reactive* due to its efficient threading model.

Bulkheads are also useful when a remote service is memory or CPU constrained, and too many concurrent requests will overload the service and cause it to fail. For example, a microservice may invoke a business-critical legacy system that is too costly or difficult to upgrade its software or hardware. The legacy system can benefit from a high-traffic microservice using a Bulkhead to limit concurrent access.

MicroProfile Fault Tolerance specifies bulkheads using the `@Bulkhead` annotation, which can be applied either to a method or to class. Listing 7.2 shows an example:

Listing 7.2 Bulkhead Example

```
@Bulkhead(10)                                    ❶
public String invokeLegacySystem() {
...
}
```

❶ `invokeLegacySystem()` is limited to 10 concurrent invocations. Attempting to exceed 10 will result in a BulkheadException

The Bulkhead annotation accepts the parameters defined in Table 7.2.

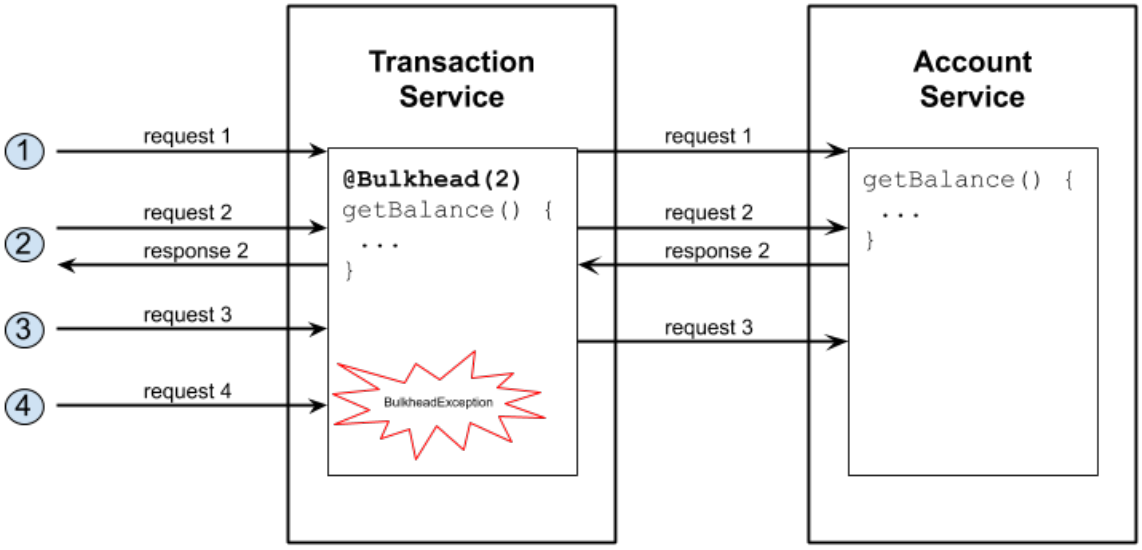
Table 7.2 @Bulkhead parameters

Parameter	Default	Description
value	10	The maximum number of concurrent invocations.
waitingTaskQueue	10	When @Bulkhead is used with @Asynchronous, this parameter specifies the size of the request thread queue.

`value` uses a semaphore, only allowing the specified number of concurrent invocations. When annotating the same method with `@Bulkhead` and `@Asynchronous`, `value` defines the number of concurrent threads allowed to invoke a method concurrently.

The `@Bulkhead` annotation can be used together with `@Asynchronous`, `@CircuitBreaker`, `@Fallback`, `@Retry`, and `@Timeout`.

Figure 7.1 demonstrates a bulkhead limiting the number of concurrent invocations to 2.



1. TransactionService receives request 1, invokes AccountService.getBalance(), and is waiting for a response. Semaphore count: 1
2. TransactionService receives request 2, and invokes AccountService.getBalance(). Semaphore count (during request): 2. Semaphore count (after response): 1
3. TransactionService receives request 3, invokes AccountService.getBalance(), and is waiting for a response. Semaphore count: 2
4. TransactionService receives request 3. Semaphore count already at limit of 2. BulkheadException is thrown.

Figure 7.1 Bulkhead sequence diagram

With a firm understanding of Bulkheads, the next step is to apply the @Bulkhead annotation using a service.

7.4 Update a TransactionService with a bulkhead

To use MicroProfile Fault Tolerance APIs with Quarkus, install the `quarkus-smallrye-fault-tolerance` extension as shown in Listing 7.3:

Listing 7.3 Install Quarkus MicroProfile Fault Tolerance extension

```
cd transaction-service
mvn quarkus:add-extension -Dextensions="quarkus-smallrye-fault-tolerance"
```

Update the `newTransactionWithApi()` method to use a bulkhead. To keep testing simple, the Bulkhead will allow one concurrent invocation.

Listing 7.4 Add @Bulkhead to TransactionResource.newTransactionWithAPI() method

```
@POST
@Path("/api/{acctNumber}")
@Bulkhead(1)                                          ❶
public Response newTransactionWithApi(
     @PathParam("acctNumber") Long accountNumber,
     BigDecimal amount)
throws MalformedURLException {
   ...
}
```

❶ If more than one concurrent operation is attempted, then a `BulkheadException` will be thrown.

As with prior chapters, start the PostgreSQL database and start port forwarding using the commands in listing 7.5:

Listing 7.5 Start PostgreSQL and start port forwarding

```
# From chapter7 top-level directory
kubectl apply -f ./postgresql_kubernetes.yml

# It may take some for PostgreSQL to start
kubectl port-forward service/postgres 5432:5432
```

Start the `AccountService` in *Terminal 1* using `mvn quarkus:dev`. In *Terminal 2*, start TransactionService using `mvn quarkus:dev -Ddebug=5006`. This instance of quarkus has to specify a debug port that does not conflict with the default debug port (5005) used by `AccountService`.

Open two more terminals, *Terminal 3* and *Terminal 4*. Each terminal will run simple `curl` commands to avoid installing any special tools. Run the code in Listing 7.6 in both terminals at the same time:

Listing 7.6 Terminal 3 and Terminal 4

```
count=0
while (( count++ <= 100 )); do
   curl -i \
        -H "Content-Type: application/json" \
        -X POST \
        -d "2.03" \
        http://localhost:8088/transactions/api/444666
   echo
done
```

In each terminal, the output should show a random mix of `HTTP/1.0 200 OK` responses and `BulkheadException` output as shown in Listing 7.7`:

Listing 7.7 Sample Terminal 3 and Terminal 4 output

```
HTTP/1.1 200 OK
Content-Length: 0

HTTP/1.1 500 Internal Server Error
content-type: text/html; charset=utf-8
content-length: 13993
...
...
org.eclipse.microprofile.faulttolerance.exceptions.BulkheadException
...
...

HTTP/1.1 200 OK
Content-Length: 0
```

The Bulkhead is successfully limiting the method to a single concurrent invocation. However, a *500 Internal Server Error* is not an ideal HTTP response to return to the caller!

The next section introduces the `@Fallback` annotation to execute alternative logic when to handle a BulheadException properly.

7.5 Exception handling with Fallbacks

The `@Fallback` annotation facilitates exception handling by specifying a fallback method containing alternative logic when the annotated method completes exceptionally. `@Fallback` can be triggered by any Java exception, including those thrown by other Fault Tolerance resilience strategies.

`@Fallback` accepts the parameters defined in Table 7.2:

Table 7.3 @Fallback parameters

Parameter	Description
applyOn	List of exceptions that trigger a fallback
fallbackMethod	Method to invoke when the annotated method throws an exception. The fallback method must have the same method signature (parameter types and return type as the annotated method). Use either this parameter or the `value` parameter.
skipOn	List of exceptions that should not trigger fallback method. This list takes precedence over the types listed in the `applyOn` parameter
value	FallbackHandler class. Use either this parameter or the `fallbackMethod` parameter.

This example uses a `fallbackMethod` to replace the *500 Internal Server Error* HTTP status code caused by the `BulkheadException` with a meaningful HTTP status code. Add the `@Fallback` annotation to `newTransactionWithApi()` and a fallback method as shown in Listing 7.8:

Listing 7.8 Add @Fallback to newTransactionWithAPI() method

```
@POST
@Path("/api/{acctNumber}")
@Bulkhead(1)
@Fallback(fallbackMethod = "bulkheadFallbackGetBalance",                ❶
                  applyOn = { BulkheadException.class })                ❷
public Response newTransactionWithApi(
     @PathParam("acctNumber") Long accountNumber,
     BigDecimal amount)
throws MalformedURLException {
   ...
}

public Response bulkheadFallbackGetBalance(Long accountNumber,
                                           BigDecimal amount) {         ❸
  return Response.status(Response.Status.TOO_MANY_REQUESTS).build();    ❹
}
```

❶ Invoke `bulkheadFallbackGetBalance()` method on an exception

❷ More specifically, invoke the `fallbackMethod` on a `BulkheadException`. Any other exceptions will be handled in a default manner.

❸ The fallback method has the same method signature (parameter types and return type) as `newTransactionWithApi()`

❹ Return a more context-appropriate `429 TOO_MANY_REQUESTS` HTTP status code

Re-run the shell script outlined in Listing 7.6 in *Terminal 3* and *Terminal 4* at the same time. The output should look similar to 7.9.

Listing 7.9 Output after adding a fallbackMethod

```
HTTP/1.1 200 OK
Content-Length: 0

HTTP/1.1 429 Too Many Requests          ❶
Content-Length: 0

HTTP/1.1 200 OK
Content-Length: 0
```

❶ The *500 Internal Server Exception* HTTP status code and Java exception output is now a *429 Too Many Requests* HTTP status code with an empty response body.

A fallback can be combined with other MicroProfile Fault Tolerance annotations. The next section will use `@Fallback` with the `@Timeout` annotation.

7.6 Defining execution Timeouts

Method invocations intermittently take a long time to execute. When a thread is blocked waiting for method completion, it is not handling other incoming requests. Additionally, the service may also have response time requirements to meet business objectives impacted by latency. Use `@Timeout` to limit the amount of time a thread can use to execute a method.

`@Timeout` accepts the parameters defined in Table 7.4:

Table 7.4 @Timeout parameters

Parameter	Default	Description
value	1000ms	A TimeoutException will be thrown if method execution time exceeds this `value`.
unit	ChronoUnit.MILLIS	Time unit of the value parameter.

The `@Timeout` annotation can be used together with `@Asynchronous`, `@Bulkhead`, `@CircuitBreaker`, `@Fallback`, and `@Retry`.

Add a method to the TransactionService to get the account balance from AccountService. It has a timeout of 100ms and will call a fallback method on a `TimeoutException`. Add the code as shown in Listing 7.10:

Listing 7.10 Add a getBalance() method to TransactionResource.java

```
@GET
@Path("/{acctnumber}/balance")
@Timeout(100)                                                    ❶
@Fallback(fallbackMethod = "timeoutFallbackGetBalance")          ❷
@Produces(MediaType.APPLICATION_JSON)
public Response getBalance(                                      ❸
     @PathParam("acctnumber") Long accountNumber) {
  String balance = accountService.getBalance(accountNumber).toString();

  return Response.ok(balance).build();
}

public Response timeoutFallbackGetBalance(Long accountNumber) {
  return Response.status(Response.Status.GATEWAY_TIMEOUT).build();  ❹
}
```

❶ Throw a `TimeoutException` if `getBalance()` takes longer than 100ms to execute

❷ Call `timeoutFallbackGetBalance()` if any exception is thrown

❸ Invokes `accountService.getBalance()` and returns the account balance. `AccountService.getBalance()` will need to complete in less than 100ms or a `TimeoutException` will be thrown.

❹ Return a reasonably context-appropriate HTTP *GATEWAY_TIMEOUT* status code.

The `@Timeout` annotation will be tested using WireMock and JUnit tests.

Update WiremockAccountService to include calls to the new `getBalance()` method. This class

will also include a small amount of refactoring. Update `WiremockAccountService` as shown in listing 7.11.

Listing 7.11 Updated WireMockAccountService to test `@Timeout`

```
public class WiremockAccountService implements QuarkusTestResourceLifecycleManager {
  private WireMockServer wireMockServer;

  @Override
  public Map<String, String> start() {
    wireMockServer = new WireMockServer();
    wireMockServer.start();

    mockAccountService();                                                    ❶
    mockTimeout();

    return Collections.singletonMap("io.quarkus.transactions.AccountService/mp-rest/url",
     wireMockServer.baseUrl());
  }

  protected void mockAccountService() {                                      ❶
    stubFor(get(urlEqualTo("/accounts/121212/balance"))
        .willReturn(aResponse().withHeader("Content-Type", "application/json")
        .withBody("435.76")));

    stubFor(post(urlEqualTo("/accounts/121212/transaction")).willReturn(aResponse()
        // noContent() needed to be changed once the external service returned a Map
        .withHeader("Content-Type", "application/json").withStatus(200).withBody("{}")));
  }

  protected void mockTimeout() {
    stubFor(get(urlEqualTo("/accounts/123456/balance"))                      ❷
        .willReturn(aResponse()
        .withHeader("Content-Type","application/json")
        .withStatus(200)                                                     ❸
        .withFixedDelay(200)                                                 ❹
        .withBody("435.76")));

    stubFor(get(urlEqualTo("/accounts/456789/balance"))                      ❺
        .willReturn(aResponse()
        .withHeader("Content-Type", "application/json")
        .withStatus(200) .withBody("435.76")));
  }

  @Override
  public void stop() {
    if (null != wireMockServer) {
      wireMockServer.stop();
    }
  }
}
```

❶ Refactor mockAccountService() into its own method

❷ Any invocation of the `/accounts/123456/balance` endpoint will invoke this stub

❸ Return a "200" HTTP *OK* status code

❹ Add a 200ms delay, which will force a `TimeoutException` on any remote call with a timeout less than 200ms

❺ Any invocation of the `/accounts/456789/balance` endpoint will invoke this stub, which will not force a `TimeoutException`.

With the AccountService endpoint mocked, create the JUnit test to test the `@Timeout` annotation as shown in Listing 7.12.

Listing 7.12 Create a FaultyAccountServiceTest class

```
@QuarkusTest
@QuarkusTestResource(WiremockAccountService.class)                    ❶
public class FaultyAccountServiceTest {
  @Test
  void testTimeout() {
    given()
      .contentType(ContentType.JSON)
    .get("/transactions/123456/balance").then().statusCode(504);      ❷

    given()
      .contentType(ContentType.JSON)
    .get("/transactions/456789/balance").then().statusCode(200);      ❸
  }
}
```

❶ Bind the WiremockAccountService to the lifecycle of `QuarkusTest`.

❷ The mocked `/transactions/1234546/balance` endpoint defines a 200ms delay. The getBalance() method defines a 100ms timeout, forcing a TimeoutException. The timeout results in a call to the fallback method with a return of a *504 GATEWAY TIMEOUT* HTTP status code.

❸ The mocked `/transactions/456789/balance` endpoint returns a 200 (OK) HTTP status code

Before running the test, **stop the AccountService** to avoid port conflicts between the `AccountService` and the WireMock server. Test the application using `mvn test`, with sample output in Listing 7.13:

Listing 7.13 Sample mvn test outupt

```
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

The next section introduces the *Retry* resilience strategy and how it can be combined with other resilience strategies like `@Timeout` to improve the overall resilience of TransactionService.

7.7 Recovering from temporary failure with Retry

Failure can be rare, perhaps due to a remote system that has an occasional unstable connection. It may be appropriate to retry a method call a few times before handling the failure in this context.

The `@Retry` annotation will retry method invocations a configurable number of times if the method completes exceptionally. The annotation accepts the parameters defined in Table 7.5:

Table 7.5 @Retry parameters

Parameter	Default	Description
abortOn	None	List of exceptions that do not trigger a retry
delay	0ms	Delay between each retry
delayUnit	ChronoUnit.MILLIS	Time unit of the delay parameter
jitter	0ms	Adds or subtracts a random amount of time between each retry. For example, a delay of 100ms with a jitter of 20ms results in a delay between 80m and 120ms
jitterDelayUnit	ChronoUnit.MILLIS	Time unit of the value parameter
maxDuration	1800000	Maximum duration for all retries
durationUnit	ChronoUnit.MILLIS	Time unit of the maxDuration parameter.
maxRetries	3	Maximum retry attempts
retryOn	Any exception	List of exceptions that trigger a retry

CAUTION **Use the Retry resilience strategy with caution. Retrying a remote call on an overloaded backend service with a small delay exacerbates the problem.**

The `@Retry` annotation can be used together with `@Asynchronous`, `@Bulkhead`, `@CircuitBreaker`, `@Fallback`, and `@Timeout`.

Add the following `@Retry` code to `transactionService.getBalance()` as shown in Listing 7.14:

Listing 7.14 Add @Retry annotation

```
@GET
@Path("/{acctnumber}/balance")
@Timeout(100)
@Retry(delay = 100,                                                    ❶
         jitter = 25,                                                  ❷
         maxRetries = 3,                                               ❸
         retryOn = TimeoutException.class)                             ❹
@Fallback(fallbackMethod = "timeoutFallbackGetBalance")
@Produces(MediaType.APPLICATION_JSON)
public Response getBalance(
     @PathParam("acctnumber") Long accountNumber) {
  String balance = accountService.getBalance(accountNumber).toString();

  return Response.ok(balance).build();
}
```

❶ Wait 100ms between retries

❷ Add or subtract 25ms from retry delay. The delay between retries will be a random value between 75 and 125ms

❸ Retry up to three times

❹ Retry on a TimeoutException only. Other exceptions will be handled normally.

Test the retry annotation by running `mvn test`. The output will not change from Listing 7.13. The mock always returns a *504 GATEWAY TIMEOUT*. As a result, the `@Retry` annotation consumes three timeout exceptions, and the final result remains a *504 GATEWAY TIMEOUT*.

The *Retry* resilience strategy attempts to recover from a failure. The next section discusses the *Circuit Breaker* resilience strategy as another popular approach to handling a failure.

7.8 Avoid repeated failure with Circuit Breakers

A circuit breaker avoids operations that are likely to fail. It is a resilience pattern popularized by the Netflix Hystrix framework and is also the most complex resilience pattern to understand. A circuit breaker consists of three steps:

1. Detect repeated failure, typically of expensive operations like a remote service invocation
2. "Fail fast" by immediately throwing an exception instead of conducting an expensive operation
3. Attempt to recover by occasionally allowing the expensive operation. If successful, resume normal operations

All three steps are configurable to meet contextual needs.

7.8.1 MicroProfile Fault Tolerance - @CircuitBreaker

The MicroProfile Fault Tolerance specification defines the `@CircuitBreaker` annotation and its behavior. The annotation accepts the parameters defined in Table 7.6.

Table 7.6 @CircuitBreaker parameters

Parameter	Default	Description
requestVolumeThreshold	20	The size of the rolling window (number of requests) used to calculate the opening of a circuit
failureRatio	.5	Open the circuit if the ratio of failed requests within the `requestVolumeThreshold` window exceeds this number. For example, if the `requestVolumeThreshold` is 4, then two failed requests of the last four will open the circuit
delay	5000ms	The amount of time the circuit remains open before allowing a request
delayUnit	ChronoUnit.MILLIS	Time unit of the delay parameter
successThreshold	1	The number of successful trial requests to close the circuit
failOn	Any exception	List of exceptions which should be considered failures
skipOn	None	List of exceptions that should not open the circuit. This list takes precedence over the types listed in the `failOn` parameter

The `@CircuitBreaker` annotation can be used together with `@Timeout`, `@Fallback`, `@Asynchronous`, `@Bulkhead`, and `@Retry`.

7.8.2 How a circuit breaker works

Figure 7.2 shows a visual time sequence of a circuit breaker, followed by a description of each labeled step.

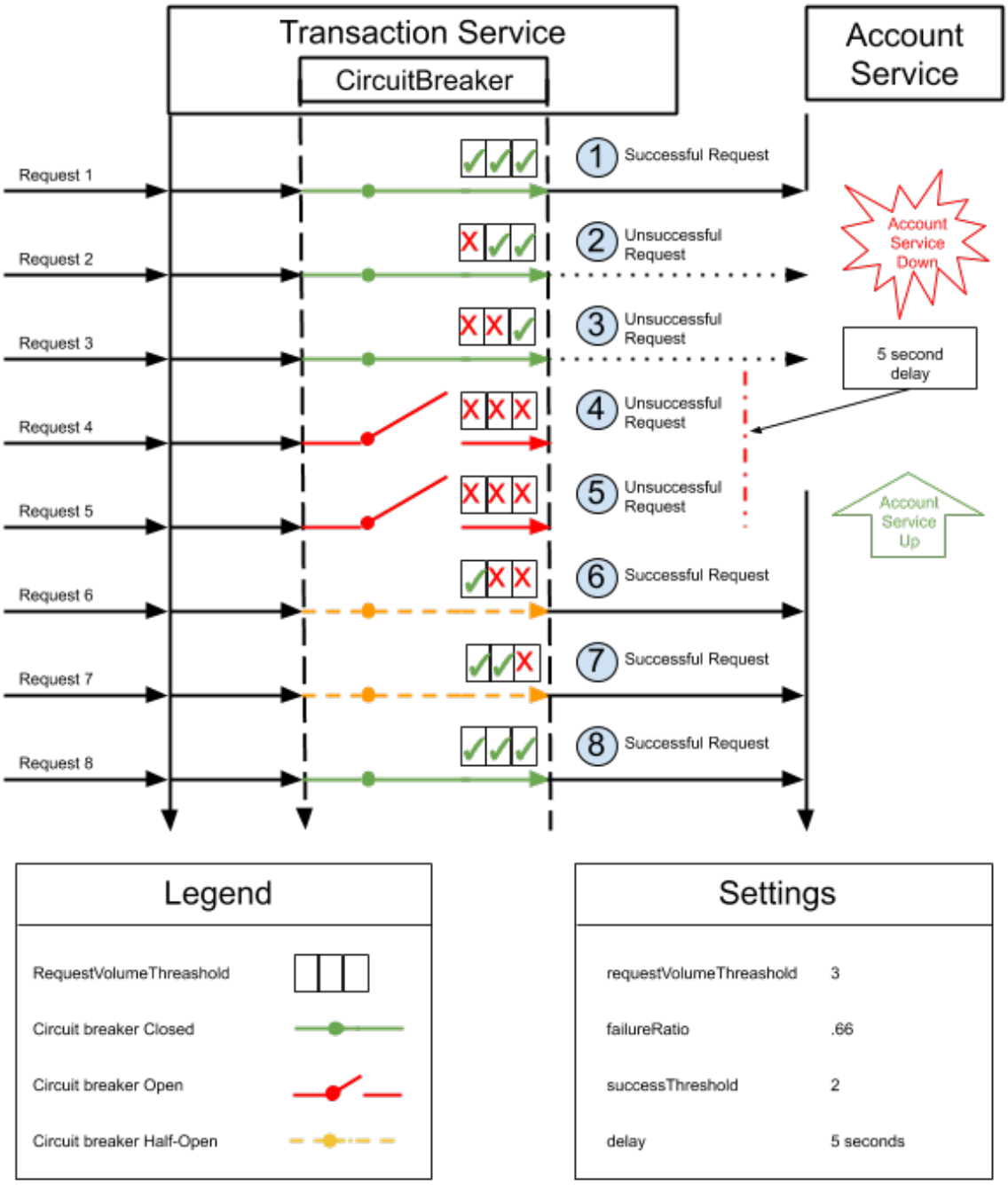


Figure 7.2 Circuit breaker in action

1. **Successful request**. The requestVolumeThreshold is 3. The last three requests have been successful, identified by the three green checkmarks.
2. **Unsuccessful request**. `AccountService` is down. MicroProfile Rest Client throws a `HttpHostConnectException`. One third (33%) of the requests have failed, identified by

the red X and two green checkmarks.

3. **Unsuccessful request**. `AccountService` is down. A `HttpHostConnectException` is thrown. The failure rate is two-thirds (66%), as shown by two red Xs. Two-thirds meets the `failureRatio`, and the next failure will result in a `CircuitBreakerException`. All requests for the next `delay` seconds (set to 5 seconds) will automatically result in a CircuitBreakerOpenException.
4. **Unsuccessful request**. The last three requests have failed. Note, the circuit opens at the end of step 3 circuit. This step represents all requests that occur during the 5-second delay.
5. **Unsuccessful request**. Although `AccountService` is back up and running, the circuit breaker will not allow any requests until 5 seconds have passed.
6. **Successful request**. After 5 second delay, the circuit is in a half-open state until `successThreshold` requests (set at 2) have successfully completed. This is the first successful request with the circuit in the half-open state.
7. **Successful request**. The second successful request. After the request, the `successThreshold` increments to 2, and the circuit will close.
8. **Successful request**. Normal request processing resumes.

7.8.3 Updating TransactionService to use @CircuitBreaker

Instead of creating another fallback method to handle a `CircuitBreakerException`, all fallback handling is moved into a separate `FallbackHandler` class with convenient console output. Add the code as shown in Listing <fallback_handler>>:

Listing 7.15 TransactionServiceFallbackHandler class

```
public class TransactionServiceFallbackHandler
       implements FallbackHandler<Response> {                              ❶

    Logger LOG = Logger.getLogger(TransactionServiceFallbackHandler.class);

    @Override
    public Response handle(ExecutionContext context) {                     ❷

        Response response;
        String name;

        if (context.getFailure().getCause() == null) {                     ❸
             name = context.getFailure() .getClass().getSimpleName();
         } else {
            name = context.getFailure().getCause().getClass().getSimpleName();
         }

        switch (name) {
            case "BulkheadException":
                response = Response
                           .status(Response.Status.TOO_MANY_REQUESTS)      ❹
                           .build();
                break;

            case "TimeoutException":
                response = Response
                           .status(Response.Status.GATEWAY_TIMEOUT)        ❺
                           .build();
                break;

            case "CircuitBreakerOpenException":
                response = Response
                           .status(Response.Status.SERVICE_UNAVAILABLE)    ❻
                           .build();
                break;

            case "WebApplicationException":
            case "HttpHostConnectException":
                response = Response
                           .status(Response.Status.BAD_GATEWAY)            ❼
                           .build();
                break;

        default:
            response = Response
                       .status(Response.Status.NOT_IMPLEMENTED)
                       .build();

        }

        LOG.info("******** "
             +  context.getMethod().getName()
             + ": " + name
             + " ********");

        return response;
    }
}
```

❶ A `FallbackHandler` class must implement the `FallbackHandler` interface

❷ The `FallbackHandler` must implement the `handle()` method. The `ExecutionContext` parameter gives contextual information such as the annotated method that generated the fallback and the exception that generated the fallback.

❸ The fallback handler logic keys on the exception name

❹ A `BulkheadException` will return a *TOO_MANY_REQUESTS* HTTP status code with an empty body

❺ A `TimeoutException` will return a GATEWAY_TIMOUT HTTP status code with an empty body

❻ A `CircuitBreakerException` will return a SERVICE_UNAVAILABLE HTTP status code with an empty body

❼ The *MicroProfile Rest Client* generates a `HttpHostConnectException` when it cannot connect to the backend REST service, and the circuit breaker circuit is in the *open* state.

The `@Fallback` annotations need the `fallbackMethod` replaced with the `FallbackHandler`. Add the `@CircuitBreaker` annotation to the `newTransactionWithApi()` method as shown in Listing 7.16:

Listing 7.16 Add @CircuitBreaker to TransactionResource.newTransactionWithApi()

```
@POST
@Path("/api/{acctNumber}")
@Bulkhead(1)
@CircuitBreaker(
    requestVolumeThreshold=3,                                    ❶
    failureRatio=.66,                                            ❷
    delay = 5,                                                   ❸
    delayUnit = ChronoUnit.SECONDS,                              ❹
    successThreshold=2                                           ❺
  )
@Fallback(value = TransactionServiceFallbackHandler.class)       ❻
  public Response newTransactionWithApi(
      @PathParam("acctNumber") Long accountNumber, BigDecimal amount) {
  ...
}

...

@GET
@Path("/bulkhead/{acctnumber}/balance")
@Timeout(100)
@Fallback(value = TransactionServiceFallbackHandler.class)       ❼
@Produces(MediaType.APPLICATION_JSON)
public Response getBalance(
    @PathParam("acctnumber") Long accountNumber) {
   ...
}
```

❶ To make testing the circuit breaker simpler, set the requestVolumeThreshold to the low value of 3.

❷ Set a failure ratio of .66 (two thirds). If 2 of the most recent three requests fail, the circuit breaker will open

❸ Set a delay of 5 seconds

❹ Set delay time unit to *seconds*

❺ A circuit breaker in the half-open state will close with two continuous successful requests

❻ Update newTransactionWithAPI to invoke the `TransactionServiceFallbackHandler` instead of the `fallbackMethod`

❼ Update `getbalance()` to invoke the `TransactionServiceFallbackHandler` instead of the `fallbackMethod`

7.8.4 Test the circuit breaker

To test the circuit breaker, extend the `WiremockAccountService` with the following:

```
public class WiremockAccountService implements QuarkusTestResourceLifecycleManager {
  private WireMockServer wireMockServer;

  private static final String SERVER_ERROR_1 = "CB Fail 1";                    ❶
  private static final String SERVER_ERROR_2 = "CB Fail 2";
  private static final String CB_OPEN_1 = "CB Open 1";
  private static final String CB_OPEN_2 = "CB Open 2";
  private static final String CB_OPEN_3 = "CB Open 3";
  private static final String CB_SUCCESS_1 = "CB Success 1";
  private static final String CB_SUCCESS_2 = "CB Success 2";

  ...

  @Override
  public Map<String, String> start() {
    wireMockServer = new WireMockServer();
    wireMockServer.start();

    mockAccountService();
    mockTimeout();
    mockCircuitBreaker();                                                      ❷

    ..
  }

  void mockCircuitBreaker() {
    // Define wiremock scenario to support the required by a circuitbreaker state machine

    createCircuitBreakerStub(Scenario.STARTED, SERVER_ERROR_1, "100.00", 200);  ❸
    createCircuitBreakerStub(SERVER_ERROR_1, SERVER_ERROR_2, "200.00", 502);    ❹
    createCircuitBreakerStub(SERVER_ERROR_2, CB_OPEN_1, "300.00", 502);         ❺
    createCircuitBreakerStub(CB_OPEN_1, CB_OPEN_2, "400.00", 200);              ❻
    createCircuitBreakerStub(CB_OPEN_2, CB_OPEN_3, "400.00", 200);             //
    createCircuitBreakerStub(CB_OPEN_3, CB_SUCCESS_1, "500.00", 200);           ❼
    createCircuitBreakerStub(CB_SUCCESS_1, CB_SUCCESS_2, "600.00", 200);        ❽
  }

  void createCircuitBreakerStub(String currentState, String nextState,          ❾
                                String response, int status) {
    stubFor(post(urlEqualTo("/accounts/444666/transaction")).inScenario("circuitbreaker")
        .whenScenarioStateIs(currentState).willSetStateTo(nextState).willReturn(
            aResponse().withStatus(status).withHeader("Content-Type", MediaType.TEXT_PLAIN)
            .withBody(response)));
  }

  ...
```

❶ The states defined for the `circuitbreaker` the Wiremock "scenario". Each field defines a circuit breaker state in order

❷ Create the circuit breaker mock

❸ Create a wiremock circuit breaker stub for each scenario state transition. This first stub defines the initial request in the requestVolumeThreshold

❹ Returns a 502, the first error the circuit breaker receives

❺ Returns a 502, the second error the circuit breaker receives. This second error will open the circuit breaker

❻ The circuit breaker is open. Even though the request returns 200, simulating service availability, the circuit breaker is in its delay period.

❼ The first successful call after the delay period

❽ The second successful call closes the circuit

❾ Any call to the `/accounts/444666/transaction` endpoint invokes a stub. Each call to the endpoint will advance the state in the `circuitbreaker` scenario. The body of the response is the account balance.

With the WireMock updated to support a circuit breaker, the next step is to update `FaultyAccountService` to test the circuit breaker as shown in listing 7.17.

1. Circuit breaker JUnit test

```
@Test
void testCircuitBreaker() {
  RequestSpecification request =
    given()
      .body("142.12")
      .contentType(ContentType.JSON);

  request.post("/transactions/api/444666").then().statusCode(200);     ❶
  request.post("/transactions/api/444666").then().statusCode(502);     ❷
  request.post("/transactions/api/444666").then().statusCode(502);     ❸
  request.post("/transactions/api/444666").then().statusCode(503);     ❹
  request.post("/transactions/api/444666").then().statusCode(503);     ❺

  try {
    TimeUnit.MILLISECONDS.sleep(1000);                                 ❻
  } catch (InterruptedException e) {
  }

  request.post("/transactions/api/444666").then().statusCode(200);     ❼
  request.post("/transactions/api/444666").then().statusCode(200);     ❽
}
```

❶ This successful request defines the initial request in the requestVolumeThreshold window

❷ Expect a 502, the first error the circuit breaker receives

❸ Expect a 502, the second error the circuit breaker receives. This request will open the circuit breaker

❹ The circuit breaker is open.

❺ The circuit breaker is still open.

❻ Sleep long enough to get past the circuitbreaker `delay`

❼ The first successful call after the delay period

❽ The second successful call closes the circuit. The circuit is now closed, and further invocations will continue normally.

> **NOTE** **Early Quarkus releases used the Hystrix framework as the underlying implementation. Hystrix has been deprecated, so later Quarkus releases use a custom implementation. Because developers develop to the MicroProfile Fault Tolerance specification, their application source code did not change. This demonstrates the real-world value of developing to specifications instead of implementations.**

7.9 Override annotation parameter values using properties

MicroProfile Fault Tolerance can globally enable or disable fault tolerance annotations, or modify annotation parameters at runtime using properties. This feature recognizes that operational needs change as the deployment environment changes. By overriding annotation parameters using properties, non-Java developers responsible for a reliable production environment can adjust fault tolerance parameters to address production needs.

Service meshes are becoming more common, which gives the operations team more control and visibility into a microservices deployment. A service mesh can shape network traffic and apply its own fault tolerance features to maintain a more reliable Kubernetes cluster. For example, by externalizing fault tolerance annotation parameters using properties, operations can ensure that application `@Timeout` or `@Retry` annotations do not conflict with the equivalent service mesh settings.

There are four ways to enable/disable fault tolerance annotations using properties:

1. **MP_Fault_Tolerance_NonFallback_Enabled=true**. Disable all fault tolerance annotations, except for `@Fallback` annotations.
2. **<annotation>/enabled=false**. Disable all fault tolerance annotations of a specific type used within the application. For example, `Bulkhead/enabled=false` disables all Bulkheads in the application.
3. **<class>/<annotation>/enabled=false**. Disable the specified annotation on the specified class. For example, `io.quarkus.transactions.TransactionResource/Timeout/enabled=false` will disable all `@Timeout` annotations defined on the `TransactionResource` class and any of its methods.
4. **<class>/<method>/<annotation>/enabled=false**. Disable the specified annotation on a specified method in the specified class. For example, `io.quarkus.transactions.TransactionResource/getBalance/Timeout/enabled=false`

will disable the `@Timeout` annotation on the `TransactionResource.getBalance()` method, and all other `@Timeout` annotations in `TransactionResource` are unaffected.

As shown in Listing 7.18, add the following to application.properties to disable all timeouts in the `TransactionResource` class:

Listing 7.17 application.properties

```
# Modify the MicroProfile Fault Tolerance settings
io.quarkus.transactions.TransactionResource/Timeout/enabled=false
```

Run `mvn test`. As shown in Listing 7.19, the test will fail because the expected timeout no longer occurs. While failing a test is not ideal, this does show that the `@Timeout` annotation has been disabled.

1. mvn test failure - expected timeout does nt occur

```
[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   FaultyAccountServiceTest.testTimeout:21 1 expectation failed.
Expected status code <504> but was <502>.

[INFO]
[ERROR] Tests run: 3, Failures: 1, Errors: 0, Skipped: 0
```

To change an annotation parameter, the property format is `<class>/<method>/<annotation>/<parameter>=value`. Define the following property as shown in Listing 7.20:

Listing 7.18 application.properties

```
# io.quarkus.transactions.TransactionResource/Timeout/enabled=false          ❶
io.quarkus.transactions.TransactionResource/getBalance/Timeout/value=150     ❷
```

❶ Comment out property disabling timeouts

❷ Change timeout value from 100 to 150. This remains under the WireMock stub delay of 200ms, which will force a `TimeoutException`.

Run `mvn test` and all tests should pass. With everything working locally, the next step is to deploy the services to Kubernetes.

7.10 Deploy to Kubernetes

Deploy the updated `TransactionService` to Kubernetes as shown in Listing 7.21. Run the same commands for the `AccountService` to ensure they are both up and running.

Listing 7.19 Terminal 2

```
# Use the minikube docker daemon to build the image
eval $(/usr/local/bin/minikube docker-env)

# Deploy to Kubernetes. Run this for both the AccountService
# and the TransactionService
mvn package -Dquarkus.kubernetes.deploy=true
```

Test the bulkhead logic while running in Kubernetes. The approach and code are nearly identical to that of Listing 7.6. In *Terminal 1* and *Terminal 2*, simultaneously run the code as shown in Listing 7.22:

Listing 7.20 Terminal 1

```
TRANSACTION_URL=`minikube service transaction-service --url`
count=0
while (( count++ <= 100 )); do
   curl -i \
        -H "Content-Type: application/json" \
        -X POST \
        -d "2.03" \
        $TRANSACTION_URL/transactions/api/444666
   echo
done
```

Next, run the command in each terminal simultaneously. The output will look similar to Listing 7.23:

Listing 7.21 Terminal 1

```
HTTP/1.1 200 OK                                  ❶
Content-Length: 0

HTTP/1.1 200 OK                                  ❷
Content-Length: 0

HTTP/1.1 503 Service Unavailable                 ❸
Content-Length: 0

HTTP/1.1 503 Service Unavailable                 ❹
Content-Length: 0
```

❶ Successful request

❷ Successful request

❸ The response returned when a `CircuitBreakerException` is thrown. Between *Terminal 1* and *Terminal 2* requests, at least two of the most recent three requests resulted in a `BulkheadException`

❹ The circuit breaker remains open. The circuit breaker will likely not close until the script is run from one terminal at a time.

NOTE **To test the Bulkhead and receive the 429 TOO_MANY_REQUESTS HTTP status code, the CircuitBreaker must skip** `BulkheadException`s. Either set the `@CircuitBreaker skipOn` **parameter to** `BulkheadException.class`**, or set it using application.properties using** `io.quarkus.transactions.TransactionResource/newTransactionWithApi/CircuitBreaker/skipOn=org.eclipse.microprofile.faulttolerance.exceptions.BulkheadException`**. This is left as an excercise to the reader.**

7.11 Summary

This chapter covers how to improve application availability using MicroProfile Fault Tolerance annotations. Each annotation offers a strategy to address potential failures to increase application availability.

To summarize:

- Resilience strategies improve application robustness.
- *MicroProfile Fault Tolerance* supports six resilience strategies: `@Asynchronous`, `@Bulkhead`, `@CircuitBreaker`, `@Fallback`, `@Retry`, and `@Timeout`.
- `@Asynchronous` executes threads on a separate thread
- The `@Bulkhead` limits the number of concurrent requests to avoid cascading failures
- The `@CircuitBreaker` avoid repeated failures by recognizing a failure and avoids executing logic for a period of time
- `@Fallback` executes alternative logic when an exception is thrown
- `@Retry` will retry a method call when an exception is thrown
- `@Timeout` prevents a method from waiting longer than a specified amount of time
- The Quarkus *RestEASY Reactive* extension can replace the `@Asynchronous` annotation.
- *MicroProfile Fault Tolerance* annotations can be enabled, disable, and customized using properties.

A Minikube

Minikube is a single-node Kubernetes cluster that targets desktop macOS, Linux, and Windows.

A.1 Installing and starting Kubernetes

Many examples require Minikube installed on your local machine. Head over to https://kubernetes.io/docs/tasks/tools/ and follow the instructions to install both kubectl and Minikube, along with any pre-requisites.

> **NOTE** The examples have been tested with Minikube v1.12.3 and Kubernetes v1.18.3.

After installing Minikube, see Listing 1.1:

Listing A.22 Starting Minikube

```
minikube start
```

A.2 Stopping and deleting the Minikube virtual machine

Minikube caan be stopped with the command shown in Listing 1.2:

Listing A.23 Stopping Minikube

```
minikube stop
```

After stopping Minikube, it can be restarted with the command in Listing 1.1.

After completing all exercises, the Minikube virtual machine can be deleted as shown in Listing 1.3.

Listing A.24 Deleting Minikube virtual machine

```
minikube delete
```

Notes

1. See https://jakarta.ee/
2. https://livebook.manning.com/book/enterprise-java-microservices
3. https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215
4. https://docs.google.com/document/d/1cCMN298rUAX45Nq-ZJv755WZe8yVjyAo1Gz_wN8r7wc/edit#heading=h.orn7
5. See https://graalvm.org/
6. https://quarkus.io/guides/maven-tooling See the maven tooling guide
7. See https://beanvalidation.org/
8. see https://quarkus.io/guides
9. see https://quarkus.io/guides/all-config
10. See https://hub.docker.com/_/postgres
11. See https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/
12. https://www.martinfowler.com/books/eaa.html
13. https://www.martinfowler.com/eaaCatalog/activeRecord.html
14. https://www.martinfowler.com/books/eaa.html
15. https://martinfowler.com/eaaCatalog/repository.html
16. https://microprofile.io/
17. https://resteasy.github.io/
18. https://thorntail.io/
19. See https://quarkus.io/guides