**10**

**Bringing It Together: MuShop**

Throughout this book, we’ve examined many cloud native development principles, techniques, and technologies. In this chapter, we describe a sample application, MuShop (pronounced Mew-Shop), that puts several of these notions and services in OCI into practice. The goal of this application is to provide a working example that implements the cloud native application design principles and technologies and showcases the operational model for such an application. It can act as a reference point for implementation concepts ranging from application development patterns to infrastructure and deployment automation, or it can act as inspiration for your own microservices. The complete source code for the example application is available on GitHub.[**1**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_1a)

It can be daunting at first for enterprise application developers to shift perspective to cloud native processes and practices, and it is common to view them with a healthy dose of skepticism. Yet cloud native applications are evolving and growing in scale all around us every day. As enterprises increasingly move to cloud providers for infrastructure, applications teams often face the tough choice of whether to refactor an enterprise application to a cloud native model or to simply “lift and shift” to cloud infrastructure. Many application teams choose to “lift and shift” because it offers the path of least resistance, short-term cost savings, and lower risk. The long-term costs, however, typically uncover the design shortcomings of the original application, which was never designed to be computationally elastic or to minimize operational cost. This is natural because these applications were designed for a CapEx[**2**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_2a) world, with an upfront infrastructure cost. The possibility of further cost optimization on the cloud invariably forces development teams to look for an operationally efficient model and to refactor the applications to achieve that elasticity. This path of lift-and-shift followed by planned optimization has proven to be a successful playbook for many organizations.

MuShop fits that traditional notion of a data-driven, transactional application, but reinvented as a set of distributed microservices and operated using cloud native principles and technologies. The reason to choose a transactional application for this example and not something more esoteric is to keep it within the realm of these everyday enterprise applications.

The motivation to build this application as a reference sample and demonstrate the various concepts and services covered in this book is twofold:

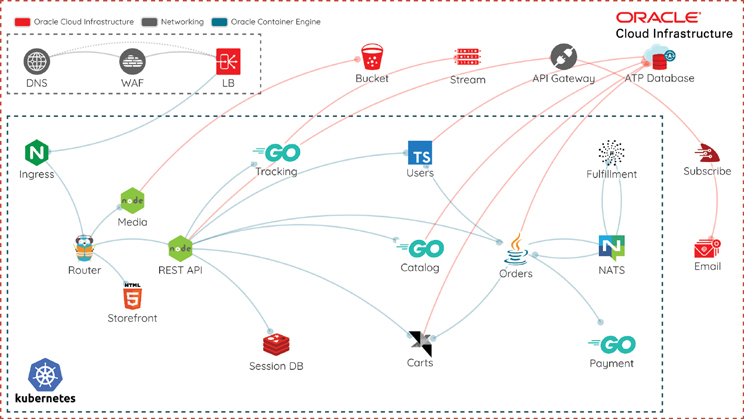
* To demonstrate the concepts to users who are new to cloud native development and to provide a deployable reference point that acts as a sandbox and a learning tool. Deploying code to a sandbox and examining how each component works is one of the best learning tools for developers.
* For experienced cloud native developers who are new to OCI, it provides a familiar application construct while introducing them to platforms and services within OCI.

MuShop showcases concepts, practices, and OCI services that are commonly used for cloud native development. In some respects, it has been overengineered, to prove a point or demonstrate features. For instance, most microservices in MuShop use a different programming language, framework, or technology stack. This is simply to demonstrate the polyglot possibilities with microservice architecture and the freedom it affords developers to choose the right tools for the job. This is not intended to suggest that all cloud native applications should consist of polyglot services. This chapter calls out these design choices and discusses them in detail. The name MuShop is a tongue-in-cheek reference to using micro (μ, mu) services to implement an e-commerce site that sells products for cats (mew!).

**Architecture**

Microservices architecture breaks down complex software systems into smaller, independent application processes. These smaller applications focus on some specific functionality that they can provide as a service, exposed through a well-defined interface. These applications are independently deployed and operated, but they communicate with each other over the interfaces they expose. The originally complex system now consists of much smaller entities that focus on very specific areas and functionality, so it becomes easier to isolate bottlenecks, scale the system, rearchitect just parts of it (maintaining the interface contracts), and more. This is not just an architectural approach to building modern applications that are scalable, portable, and resilient; it is also a software development methodology, in that it enables independent teams to build, test, and release software in relative isolation. This approach also allows teams to implement functionality, using any tools or technology stacks of their choice, as long as the interface contracts and API protocols are honored. Many of these applications also leverage the twelve-factor application methodology,[**3**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_3a) which is a set of popular guiding principles to implement software that is to be delivered as a service. Because most microservices are implementations in which some functionality is delivered as a service, the twelve-factor methodology is great at providing implementation guidance.

MuShop implements an e-commerce use case—a website that delivers a shopping experience for products, carts, order management, and more. MuShop consists of several microservices written in various languages, using frameworks and libraries that are popular in their respective communities. Each service exposes or provides a REST API that makes up the features of the service. The choice to use REST as the API protocol in these services (instead of other choices, such as gRPC) is a conscious one; although gRPC offers several advantages over REST in terms of efficiencies and performance, REST is more familiar to the wider audience and makes the examples more relatable for users. [Figure 10-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ch10fig01) shows the high-level architecture of MuShop.



**Figure 10-1** High-Level Architecture of the MuShop Example Application—Each Microservice Uses Its Own Stack to Demonstrate the Polyglot Nature of This Application

It might not look like a typical *n*-tier architecture of a traditional data-driven application; instead, it could look like a web of dependencies. The tiered model still exists within these applications; however, it has been up-leveled to show the individual services and the interactions between them. A quick glance can show you the critical services that many other services depend on and illustrate what each service interacts with. The diagram also separates the applications and services that are deployed to a Kubernetes cluster from platforms and services outside the cluster. All the components that are deployed to the cluster are component services and supporting services of MuShop, and these services depend on external services that are outside the cluster. Examples of these external dependencies include an autonomous Oracle database and object storage buckets.

Most architectures have external dependencies that are managed cloud services, legacy systems, or self-managed systems that are run outside the cluster itself. Although this is natural and common, care should be taken to decouple them from the applications using configuration. Strong dependence on these external services can introduce vendor lock-in and make the application less portable. Therefore, it is key to design applications so that these dependencies can be managed as externalized configuration or using standard interfaces and APIs. These could be a standard object storage API or an Object Relational Mapping (ORM) framework, such as JPA in Java-based applications, to abstract your applications from hard dependencies on the underlying database.

MuShop manages these external dependencies through configuration that can be provided to the individual containers. This configuration can be in the form of environment variables or configuration files that are provided to the containers at runtime (such as a config map in Kubernetes). This allows the same containers to be run with different configuration options, such as for development environments and production environments.

An architectural goal of adopting a cloud native development model is to improve system resilience. One way to do that is to eliminate single points of failure in the system. A single service that is used by several other services has the potential to become a bottleneck. Microservice architectures make it easy to identify these services and plan accordingly. External dependencies pose a similar challenge, as with dependence on a single database. Microservice architectures often recommend a database-per-service model, with each service using its own database on a dedicated infrastructure. MuShop, however, opts for a slightly modified take on this pattern by using separate database schemas on a shared database infrastructure in its default configuration. This is done to demonstrate application-level isolation while keeping the infrastructure needs of a sample application to a minimum. From a service perspective, and for implementation purposes, this is close to the database-per-service pattern. A mere configuration change to how the application connects to databases can switch from multiple schemas to completely separate databases. Because configuration is external to the application containers for the services, switching between a shared database infrastructure and dedicated databases for each service can be achieved without requiring code changes or rebuilding the container images. When appropriate, this pattern can also be used to share database resources in dev/test environments; then a switch can be made to use dedicated databases for more critical environments, simply using external configuration.

**Source Code Structure**

The source code for MuShop is managed in a single repository, to make it easier to work with and understand the entire system in the same context. This style of using a single source code repository to store multiple projects is sometimes called a monorepo. The typical manner of source-controlling an individual project in its own repository is called a multirepo in this context.

Multirepos are familiar to most developers and are the norm. The choice to use a monorepo or multirepo for your microservices applications depends on a set of factors that should be carefully considered. By their nature, microservices have a smaller codebase, fewer developers, and fewer dependencies than the system as a whole. A multirepo typically gives these teams more autonomy, and these projects can move independently. The commit histories for each project look clean, with commits related to just that project or service. Additionally, code reviews are streamlined and focused, and CI/CD configurations are simple because they focus on building just a single service or application. In sensitive environments, multirepos can meet compliance and security needs around who has access to sensitive code. On the other hand, multirepos can silo developers, and when a single small team works on several services, multirepos have to work across repositories, which can be counterproductive as the number of repositories increases. The drawbacks are easily understood with an example. Consider a team that uses a multirepo to implement a new feature. If a feature is to be implemented across three separate services, then, by extension, it is spread across three separate repositories. This can make it difficult to see the whole feature in its entirety because its code is spread across the repositories. This approach might also be overkill when the same team members are forced to switch among the three repositories to make quick changes that are then tracked as three separate code commits. Multirepos can also act as an organization barrier, siloing developers into their own corners and features.

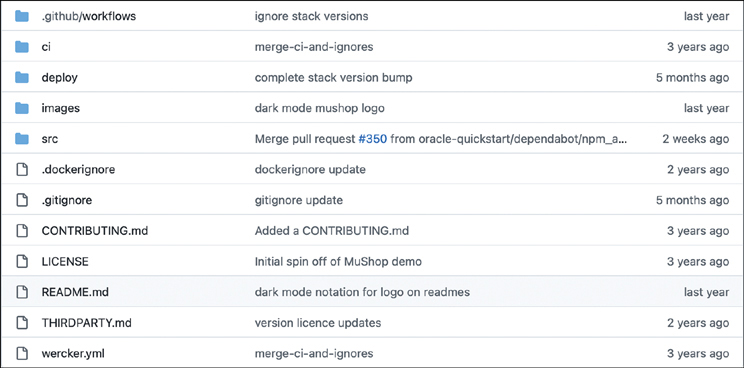
A monorepo, on the other hand, can encourage consistency of coding standards and styles when using the same programming language and comparable tech stacks. Additionally, you get a complete view of the larger system as a whole. The dependencies of the whole system can be analyzed at once, and useful reports can be generated easily in a monorepo. Above all, a monorepo breaks down organizational walls between developers and promotes collaboration. On the flip side, with large teams and complex systems comprised of numerous microservices, a monorepo can be quite chaotic, with the commit history muddled by commits, merges, and conflict resolutions across teams. Care also needs to be taken to craft your CI/CD pipelines so that the changes to the repository trigger the appropriate build and deployment workflows.

Beyond these pros and cons, organizational and compliance requirements might dictate the choice of source code structuring. MuShop uses a monorepo primarily because of its role as an example application. As such, we wanted to make it easier for users to get started with MuShop; not having to clone multiple repositories was a choice we made to support that.

Within MuShop, the source code is primarily organized into two groups:

* Code that automates the deployment, including infrastructure automation. This can be found under the directory deploy.
* The source code for each of the microservices. This can be found under src. MuShop uses a variety of programming languages and frameworks for its services. Each service under the src acts as an independent project, with its own dependency management, build tools, tests, and packaging. The common aspect is that all the services are packaged as container images and follow the general principles of the twelve-factor application methodology.

[Figure 10-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ch10fig02) shows the high-level view.



**Figure 10-2** The Layout of the MuShop Code Repository

To familiarize yourself with MuShop, clone it to your workstation. You need to have Git[**4**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_4a) installed. If you are new to using Git, a client such as the GitHub desktop ([https://desktop.github.com](https://desktop.github.com/))[**5**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_5a) is highly recommended.

To clone the repo, run the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0386-01a)

git clone https://github.com/oracle-quickstart/oci-cloudnative.git

This clones the repository to your workstation, and you should be able to browse the source code.

**Services**

The source code for every service that makes up MuShop can be found in the src folder. MuShop takes a polyglot approach to microservices, with each service choosing a different programming language and framework and having its own stack. Some of these services are built for both the x86 (amd64) and ARM (arm64) platforms, which enables more portability for workloads as ARM-based compute platforms are becoming more mainstream with cloud providers. This offers an example of how to implement a development workflow to target multiple architectures for your workloads and effectively package them as containers that support these multiple architectures. Each service is responsible for a single function; their nuances are described in the sections that follow.

**Storefront**

**Technology stack:** JavaScript

**Target architecture:** amd64/arm64

**Description:** The Storefront is a responsive single-page web application that implements the MuShop storefront. This is the web page that users visit to shop at MuShop. As the user browses the store, adds items to the cart, and creates orders on the browser, this web application makes calls to the other services that expose these respective services. Instead of having the storefront depend on and know about every back-end API and how to access it, these APIs are aggregated by the API service, which becomes the single entry point for the storefront to access all services. The API service acts as a facade for the APIs provided by the various microservices.

The application uses UIKit ([https://getuikit.com](https://getuikit.com/)) for the UI components. It also uses axios ([https://axios-http.com](https://axios-http.com/)) as its HTTP client to make API calls to the API service.

**API**

**Technology stack:** nodeJS

**Target architecture:** amd64/arm64

**Description:** This service acts as a storefront back end. It is written in Node.js and orchestrates services for consumption by the Storefront web application. The storefront UI makes its API calls to the API service, which then passes these along to the respective implementations. The API service acts as a facade for the APIs provided by the various microservices, such as the Catalog service for browsing the store, Carts service for adding items and keeping track of them across sessions, and Orders Service to create and manage orders. Its role is similar to that of a reverse proxy, or a very lightweight API gateway. The design choice of not using a managed full-featured API gateway service such as the OCI API Gateway was made simply because it does not require most of the features offered by such platforms. Implementing a service discovery mechanism and using that directly from the client (storefront) is another way to get a similar effect without the added facade, and this could be more efficient in some circumstances. However, this shifts the API orchestration overhead to the client and removes the capability to have centralized management for API endpoints.

The API service also supports a “mock mode,” which completely mocks the services underneath. This is generally useful for the development and testing of the API consumers (such as the storefront web application) without having real implementations for the actual microservices. Most API gateways can also provide similar functionality, to enable these use cases of parallel development.

**Catalog**

**Technology stack:** Go

**Target architecture:** amd64/arm64

**Description:** The Catalog service provides an API for querying the catalog/product information. The product data is stored on Oracle Autonomous Database. The service uses the GOdror (Go Driver for Oracle DB) with GoKit to interact with the Oracle DB from Go. The API exposed by the service is read only, and the sample application uses an SQL script that is run at application deployment time to seed the catalog data into the database.

**Carts**

**Technology stack:** Java, using the Helidon framework

**Target architecture:** amd64/arm64

**Description:** This service provides a cart that users can use while shopping. The cart is tied to the user profile and is persisted in an Oracle database. The Carts service uses the Autonomous Database’s JSON features to store cart data as JSON documents instead of relational tables. This service provides an example for building a Java-based microservice using the Helidon framework (<https://helidon.io/>), as well as demonstrating the features of the Oracle database for JSON document storage, queries, and joins across document and relational data models.

It is built using the maven build tool.

**User**

**Technology stack:** TypeScript, NextJS, and TypeORM

**Target architecture:** amd64/arm64

**Description:** The User service manages customer accounts, customer profile data, and authentication. The User service provides the capability to create new user accounts and update their profile information. It also handles authentication in MuShop because introducing an identity-management solution would have been overkill for a sample application. Identity information maintained by other services (such as the order created by a user) can reference user identifiers (such as a user ID) provided by the User service. It is important to note that MuShop uses a slightly modified version of the database-per-service pattern, which is essentially that every microservice has its own database. As an example application, to keep resource usage minimal while promoting the architectural patterns that promote data isolation and better resiliency than, say, using multiple databases, MuShop uses multiple schemas. This means that services that use user information, such as the orders service, maintain its data in a separate schema or database from the user service. Validation of identifiers is done across services, not by a traditional foreign key relationship that is common in monolithic applications.

The users service is built in TypeScript using the NestJS (<https://nextjs.org/>) framework, which uses progressive JavaScript to build efficient Node.js server applications. It also uses TypeORM, which is an Object Relational Mapper for TypeScript and JavaScript. Oracle DB connectivity is enabled by the official node-oracledb package from Oracle, for Node.js.

**Orders**

**Technology stack:** Java, using the SpringBoot framework

**Target architecture:** amd64/arm64

**Description:** The orders microservice is a lightweight application built using SpringBoot (<https://spring.io/projects/springboot>) that leverages Spring JPA for database connectivity. It exposes a REST API for order management operations. The application is typical of a spring JPA application and exposes CRUD operations on orders. It interacts with the user service over the REST API to track and validate users for whom the orders are created.

It also interacts with the Fulfillment service over an asynchronous message bus (NATS.io). As orders are created, it sends messages to the fulfillment service to fulfill them. This interaction is meant to showcase a messaging-driven pattern, which is common in microservices and many modern reactive architectures. Messaging systems promote elasticity, scaling, and loose coupling for services that operate at different velocities in a cloud native environment. If the order volume spikes, the service can scale up, but a related service can be cushioned from these spikes using the message bus, which buffers the increased order flow to a rate that the fulfillment service is capable of. More importantly, messaging systems can set failure boundaries. If the fulfillment system goes down, that failure should not cascade onto other services, such as orders. The messaging service that sits between microservices and the asynchronous communication model prevents these cascading failures.

It uses gradle (<https://gradle.org/>) as its build tool.

**Fulfillment**

**Technology stack:** Java, using the Micronaut framework

**Target architecture:** amd64/arm64

**Description:** Fulfillment application models an asynchronous service that models a fulfillment workflow based on incoming messages that represent orders that have been placed. The messaging platform used is nats.io, and the fulfillment application listens for messages appearing on a topic. The application reads these messages and sends reply messages to indicate the processing of orders on a separate topic that the orders application listens to. The orders application updates the status of an order based on the messages it receives from the fulfillment service. This message-driven flow between the orders and fulfillment applications showcases the asynchronous patterns that can be used in cloud native applications.

The Fulfillment application is written in Java and uses the Micronaut (<https://micronaut.io/>) framework. It is built using the gradle build tool. This application is built as a native binary using the GraalVM (<https://www.graalvm.org/>) native image compiler. The GraalVM native image compiler can create platform-specific native binaries for several languages, including Java. The native image is machine code, and no traditional Java Virtual Machine (JVM) is required to run it. This improves the performance of the application, in most cases. The gradle build included can be configured to switch between creating a standard Java bytecode and using the native image. For the Fulfillment application, there is an order of magnitude difference in the performanc due to its use of the GraalVM Native Image compiler.

**Payment**

**Technology stack:** Go

**Target architecture:** amd64/arm64

**Description:** This is a bare-bones service written in Go that performs a simple validation out of the box and can be expanded to cover new use cases and integrations. Because MuShop was built to showcase microservices architectures and was often used in hands-on workshops and meetups, the payment service scaffold primarily acted as the bare-bones service to expand on and make code changes to these settings.

**Assets**

**Technology stack:** Node.js

**Target architecture:** amd64/arm64

**Description:** This is a container used during deployment of the application that pushes the image assets used by the application to OCI object storage. It contains a Node.js OCI client and the images used in the application. These images are uploaded to the object storage bucket that is provided as a parameter. It runs only during deployment and exists after the images are uploaded to the object storage. The object storage bucket to use for the images is typically created by infrastructure automation such as Terraform, and the URLs are provided to the Helm chart. Users can optionally override this as well.

**DBTools**

**Technology stack:** None. This is a collection of utilities.

**Target architecture:** amd64/arm64

**Description:** This container packages common runtime tools for database interactions that are used across services to interact with the database. For instance, MuShop services that interact with the database might need to set up or update the database schema when a new version is deployed or seed data into the database when first run. This container packages the tools used to execute database scripts so that they can be run as Kubernetes jobs when required.

**Edge Router**

**Technology stack:** None

**Target architecture:** amd64/arm64

**Description:** This is an optional container running the Traefik proxy, for use in development environments.

**Events**

**Technology stack:** Go

**Target architecture:** amd64/arm64

**Description:** The Events service is an optional service that captures events from the storefront and sends them to an event stream. Stream-processing applications can listen on this stream and react to it. This service is intended to showcase integration with the OCI streaming service.

**Newsletter Subscription**

**Technology stack:** Node.js

**Target architecture:** amd64/arm64

**Description:** The newsletter subscription is a serverless function that is hosted on the OCI Functions platform. When users sign up for the MuShop newsletter, the storefront captures the email address and invokes the function through the OCI API gateway. The function sends an email to the recipient, informing them that they are subscribed to a newsletter. Note that the subscription is not tracked or stored anywhere; the sample application simply showcases how to invoke a function through an API gateway and how to send emails using the Oracle Email Delivery service.

**Load**

**Technology stack:** Python

**Target architecture:** amd64/arm64

**Description:** The load directory contains a set of test scripts that use the locust.io stress-testing tool. The tool lets developers define user behavior using Python scripts. The scripts themselves send HTTP requests and receive responses much like the Storefront. Locust.io lets developers define user flows by creating conditional logic based on the responses received and creating more requests based on values from previous requests. Flows can be randomized as well. When the flow is defined, the locust.io tool can spawn multiple instances that simulate users and hit the services concurrently. MuShop uses these load scripts to test the functionality, as well as simulate users for creating realistic load conditions to test scaling operations and the resiliency of the microservices.

**Building the Services**

The source code for every service that makes up MuShop can be found in the src folder. Microservices consist of multiple smaller applications that are each built independently. Compared to a traditional monolithic application, building microservices can often be complex, with each application having subtle nuances in its build processes. On top of this, for a polyglot application such as MuShop, in which each service is built using separate tool chains, maintaining these build tool chains can become overwhelming. Imagine maintaining a local build environment with the compilers and tools to build Go applications, Java applications (some using maven, others using gradle), TypeScript applications, and more. Now consider every team member who needs to build these applications maintaining a similar setup on their workstations and the whole team being in lock-step with upgrades to all versions of compilers and tools. Even though these microservices vary widely in the build tools they use, they all share a common packaging format of a container image. Ultimately, the end result of each of these build processes is a container image.

To address the complexity of the build tooling and standardize it for teams, it is a common and preferred practice for containerized application development to also containerize the build tools and processes. Instead of requiring developers to maintain these tool chains, the build tools and processes are contained within a container, the application code to be built is mounted onto a container, and the build and packaging are executed within the container. This makes the build process smooth and extremely portable because the build tools and build environment no longer need to be maintained locally. Because the containers that provide the build environment are built from Dockerfiles, it helps to codify the build environment and the tools it is expected to have and to maintain a history of how the build environment changes over time. Docker also provides support for multistage builds that help with adding software and build tools for build stages, but basing the final output image on a slimmer base image that includes only the runtime dependencies. For instance, the final image for an application need not include a package manager or build tools when the build stages use containers that have these components. Avoiding image bloat by retaining only runtime dependencies for the applications optimizes the image for its size and minimizes its attack surface.

MuShop takes this approach, and each microservice describes how it is built by providing a Dockerfile. These Dockerfiles also take advantage of Docker’s multistage builds to optimize the final images. [Listing 10-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_1) shows an abbreviated version of the Dockerfile for the Orders service.

**Listing 10-1** An Example Showing the Multistage Builds Using Docker

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0392-01a)

# Stage 1 : Setup the build environment

FROM gradle:6.5 as buildenv

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY settings.gradle /usr/src/app

COPY build.gradle /usr/src/app

# Stage 2 : Build the application

FROM buildenv as appbuild

COPY src /usr/src/app/src

RUN gradle clean test bootJar

# Stage 3 : Application container

FROM openjdk:13-slim

COPY --from=appbuild /usr/src/app/build/libs/orders-1.0.0.jar /app/orders-

1.0.0.jar

EXPOSE 80

ENTRYPOINT java $JAVA\_OPTS -jar /app/orders-1.0.0.jar --port=80

The build process is divided into three stages. The first stage sets up the build environment. It starts off by choosing the gradle container image as the base image because the application uses gradle as its build tool. It then adds the gradle settings file and build file for the application. This is set up as a single stage, to take advantage of how Docker runs multistage builds. When this build is run by Docker, it creates an intermediate container for this stage, and this can be cached. If there are no changes to the files in this container—that is, the gradle settings and the build file (which includes library dependencies)—then this container does not have to be re-created in subsequent builds, making those subsequent builds faster.

The second stage adds the application source code and runs the actual gradle build. This works because this stage is based on the intermediate container created in the first stage, which includes the gradle build tools and settings required for this build. This container gets updated every time the source code changes, which is expected because you need to rebuild the application when you make changes to the code.

The last stage builds the final application container. Notice that it starts with a new base image. It does not carry on from the build environment you set up and used in the first two stages because when you run the application, you do not need the build tools (such as gradle). You can keep the image small and lightweight by including only the runtime components you need. Because Orders is a Java application, you need a Java runtime, and this is why here you base the final application container on the openjdk:13-slim base image. It provides the basic Java runtime needed for this Java application. This stage simply copies the executable JAR file you built during the previous stage and then sets up the container to run the application.

Once again, the multistage build optimizes the build process here. If changes are made to the command-line flags or other parameters that are passed to the application, then a change to the final stage of the build without any source code changes will be very fast. This is because the first two stages are unaffected and the previously cached intermediate containers are still up to date; the build can then skip to the third stage directly without actually rebuilding the application.

The build flow for each application might be different and is based on the build process for the respective programming language and packaging models for the frameworks used. Because the builds are run in containers as well, automation systems can easily build these images and integrate with image repositories or sophisticated deployment models. MuShop is published on GitHub, so it uses GitHub workflows to automate the application and image build processes. MuShop also uses GitHub workflow features to support multiple CPU architectures for images so that these can be even more portable. The complete GitHub workflow for the applications can be seen by examining the .github/workflows directory. Here the containers.yaml workflow lays out the automated process that builds and pushes container images when their source code is updated. The docs.yaml file lays out the process for publishing docs to the GitHub site. There is a workflow to manage stale open issues on GitHub as well.

**Infrastructure Automation**

Infrastructure automation is a key operational characteristic in cloud native application architectures, and MuShop demonstrates these concepts using the OCI Terraform provider. The infrastructure management code is in the deploy/complete/terraform directory. The infrastructure automation flow covers the creation and management of resources such as the virtual network, the OKE cluster, and the worker node pool. The code is split into multiple files that work with specific resource areas, for better readability and maintainability. Module usage was consciously avoided for this Terraform configuration, for a few reasons. First, this approach would provide a more complete example of how these resources are used. Second, modules would have introduced external dependencies that are more difficult to control.

As with IAM policies, some OCI resources are always created in the home region for a tenancy. For this reason, you will notice that the Terraform configuration uses multiple provider definitions, one for the home regions and the other for the region where resources are deployed. [Listing 10-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_2) demonstrates multiple providers to interact with multiple regions at the same time.

**Listing 10-2** An Example Showing Multiple Terraform Provider Definitions to Work with Multiple Regions Simultaneously

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0394-01a)

provider "oci" {

alias = "home\_region"

tenancy\_ocid = var.tenancy\_ocid

region = lookup(data.oci\_identity\_regions.home\_region.regions[0], "name")

user\_ocid = var.user\_ocid

fingerprint = var.fingerprint

private\_key\_path = var.private\_key\_path

}

provider "oci" {

alias = "current\_region"

tenancy\_ocid = var.tenancy\_ocid

region = var.region

user\_ocid = var.user\_ocid

fingerprint = var.fingerprint

private\_key\_path = var.private\_key\_path

}

The Terraform configuration also uses the kubernetes and helm providers to deploy the application after the infrastructure resources are built. The default configuration values and parameters are provided in the included terraform.tfvars.example file. This file should be renamed to terraform.tfvars before use. The Terraform configuration can be customized by changing the values in this file. [Listing 10-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_3) shows the boilerplate code that contains placeholders that should be replaced with actual values.

**Listing 10-3** An Example terraform.tfvars File with Placeholder Values—the Content of This File Determines How Terraform Authenticates with OCI

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0395-01a)

# OCI authentication

tenancy\_ocid = "ocid1.tenancy....."

fingerprint = "" # e.g.: "5f:53:..." or leave blank if using CloudShell

user\_ocid = "" # e.g.: "ocid1.user..." or leave blank if using CloudShell

private\_key\_path = "" # e.g.: "/users/user/.oci/oci\_api\_key.pem"

# Deployment compartment

compartment\_ocid = "ocid1.compartment...."

# region

region = "us-ashburn-1"

The Terraform configuration can also be directly imported into the OCI Resource Manager service. The resource manager can load the Terraform configuration from a remote URL and generate a user interface for users to provide custom values for the terraform.tfvars configuration file. This provides a one-click experience for managing the infrastructure deployment and the application deployment. The direct link to import a stack into resource manager can be used for any Resource Manager Stack and can be used as an example to provide templated environments for team-based development. For instance, if the platform engineering team wants to standardize infrastructure topologies, it can provide a standardized stack that application teams can use that is guaranteed to have the topology and configuration that the platform team approved.

**Helm Charts**

As with most microservice applications on Kubernetes, MuShop consists of multiple pods, replica sets, services, config maps, and other Kubernetes resources. Each service in MuShop offers multiple configuration options that determine the behavior of the system. This configuration can be maintained in the Kubernetes manifests, but doing so for applications that have numerous moving parts and configuration options can be a challenge. Apart from becoming verbose and growing in complexity over time, this approach comingles the configuration data with the Kubernetes resource definitions.

In a typical development workflow, as you release new versions of your software, you want to build your containers once but then configure these containers and your Kubernetes deployments differently in your test and production environments. For instance, with MuShop, several of the services use databases; which database to connect to is a configuration option for these services. In a development environment, the service might use a shared database, whereas in production it might get a dedicated database. As developers, we want to make sure that the code that we build and test is exactly the code that we run in production. By decoupling the configuration from the code, we can ensure that the same application and the same container can be configured to run in various environments. However, maintaining a set of Kubernetes manifests for each deployment environment becomes hard to scale and keep track of.

A Helm chart provides a templating mechanism for Kubernetes manifests so that you do not have to maintain separate manifests for various configurations. Helm charts separate the Kubernetes resource definitions from the configuration values. The resource definitions are maintained in a template form with placeholders that can be replaced for actual values. The configuration values are separated from resource definitions, and this configuration is maintained in a human-readable YAML document, commonly referred to as a values file. Having configuration values consolidated like this makes it easy to understand the configuration values and maintain them over time. Because the configuration values are separate, it is also easy to keep track of changes by simply storing the configuration in a source control system. When a Helm chart is installed to a cluster, Helm replaces the placeholder values in the manifest templates using the configuration values from the values file provided to it. This effectively creates a complete manifest using these values. This generated manifest is applied to the Kubernetes cluster. [Listing 10-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_4) shows a deployment manifest template (truncated, for brevity), and [Listing 10-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_5) shows a values.yaml file helm parameter. [Listing 10-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_6) shows the full manifest file.

**Listing 10-4** An Example of a Deployment Manifest Template

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0396-01a)

apiVersion: apps/v1

kind: Deployment

metadata:

name: api-server

spec:

replicas: {{ .Values.replicaCount }}

template:

spec:

containers:

- name: api-server

image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"

imagePullPolicy: {{ .Values.image.pullPolicy }}

**Listing 10-5** **values.yaml** File

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0396-02a)

replicaCount: 1

image:

repository: iad.ocir.io/oracle/ateam/mushop-api

tag: 2.3.2

imagePullPolicy: IfNotPresent

**Listing 10-6** Resulting Deployment Manifest

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0397-01a)

apiVersion: apps/v1

kind: Deployment

metadata:

name: api-server

spec:

replicas: 1

template:

spec:

containers:

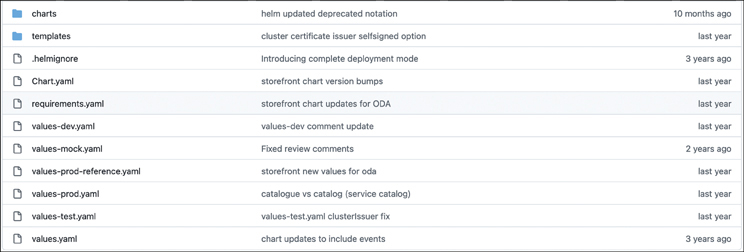
- name: api-server

image: iad.ocir.io/oracle/ateam/mushop-api:2.3.2

imagePullPolicy: IfNotPresent

As shown in [Listing 10-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_4), the manifest template has several placeholder values. The YAML-based values.yaml file shown in [Listing 10-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_5) defines these values in a way that is easy to navigate, read, and understand. The YAML format also makes this a well-organized format. At runtime, the placeholders in the template are replaced by the values from the values.yaml file, to generate a full manifest as shown in [Listing 10-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_6).

The Helm charts for MuShop are responsible for deploying and configuring each of the services in the system. Helm charts can reference other Helm charts from within it. This makes it easy to compose an application out of independent Helm charts. Every microservice in MuShop provides its own Helm chart, which exposes its configuration parameters in a well-defined form. This Helm chart for an individual service can be used to configure and manage that service alone. Several of these Helm charts can be pulled together to create a higher-level Helm chart that configures and manages multiple services. MuShop is organized in this manner, and the MuShop Helm charts are found under the path <repository\_location>/deploy/complete/Helm-chart/mushop/. Here you will find the top-level chart, which is made up of multiple subcharts, one for each service. The subcharts can be found under the charts directory, as [Figure 10-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ch10fig03) illustrates.



**Figure 10-3** The MuShop Helm Chart Consists of Multiple Subcharts—the Application Can Be Configured Differently for Various Environments by Using Customized values.yaml Files

Each application or microservice has its own configuration and resources that are modeled by a Helm chart that can provision and manage that application. Each of these subcharts contains a values.yaml file that provides sensible and secure defaults for the application. The top-level MuShop Helm chart is composed of the subcharts for each of the microservices, along with some global resources and templates. Even though each microservice (subchart) provides its own values, the top-level chart can override values that are set by them. This enables developers to manage the configuration centrally. Finally, the user can provide a fully customized YAML file that fully expresses the configuration and overrides any of the configurations that have been set underneath. [Figure 10-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ch10fig03) shows several of these values-files for various environments such as values-dev.yaml, values-test.yaml, and values-prod.yaml. When changing values for a subchart, only the elements that need to be changed are specified in the top-level values-file; the values not specified are not overridden, and the defaults from the chart’s values-file are still applied.

If you are new to using Helm charts with subcharts, this might seem confusing at first. Consider this example. The Orders Service has its default values defined within the orders subchart by the developers of the Orders Service, who have provided some sensible defaults. In this example, the application expects environment variables to be set for the container. [Listing 10-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_7) shows the default values.yaml that the orders subchart provides, with its defaults. Note the newOrdersSubject: mushop-orders, which determines the name of the subject onto which messages are posted when new orders are created. The developers of the orders service have set the default to be mushop-orders, which is a reasonable default.

**Listing 10-7** The Values File in Each Subchart Sets the Defaults for All Configuration Parameters—This Example Shows a Snippet from the values.yaml for the Orders Chart

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0398-01a)

env:

zipkin: zipkin.jaeger.svc.cluster.local

javaOpts: -Xms32m -Xmx150m -XX:MaxRAM=150m -Djava.security.egd=file:/dev/urandom

-Doracle.jdbc.fanEnabled=false -XX:+UnlockExperimentalVMOptions -XX:+UseZGC

-Dlogging.level.mushop.orders=INFO -Dspring.zipkin.enabled=false

natsHost: “nats”

natsPort: 4222

newOrdersSubject: mushop-orders

shippedOrdersSubject: mushop-shipments

However, when deploying MuShop to a development environment, you might need to override this. At the time of performing a Helm install, the user can provide a customized values.yaml file, such as values-dev.yaml. This file would contain only the configuration parameters that need to be overridden, nothing else. [Listing 10-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_8) shows such a value file in which the newOrdersSubject is overridden to have a value of test-env-orders. The other values in the orders Helm chart are still in effect, but the value for the newOrderSubject has been overridden.

**Listing 10-8** An Example Showing Selective Values from the Underlying Chart Being Overridden

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0399-01a)

... (truncated for brevity)

# The environment specific override.

orders:

env:

newOrdersSubject: test-env-orders

...

**Utilities and Supporting Components**

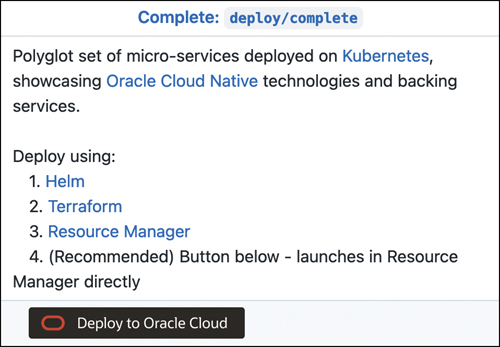
As with most applications, MuShop also uses third-party open-source software, where required. Within the application, MuShop uses the NATS messaging system to asynchronously connect microservices and set failure boundaries. MuShop also uses software that is outside the core application, to support its functionality. The monitoring stack is a good example. Observability is essential to smoothly running a microservices-based application because of the sheer number of moving parts in the system. MuShop uses the Grafana and Prometheus stack for monitoring and installs these to the Kubernetes cluster as well. To keep the code well organized, the core application Helm chart is separate from the supporting software. This helps to easily keep track of application changes, separate from changes to the supporting software. MuShop maintains these in a separate chart called MuShop Utilities. The MuShop Utilities chart includes the following components:

* **Prometheus:** Monitoring system and time series database for monitoring events from the workloads running in the cluster.
* **Grafana:** Utility for querying, visualizing, and alerting based on metrics data. It integrates with Prometheus as the data source.
* **metrics-server:** Metrics Server, which collects resource metrics from kubelets and exposes them.
* **ingress-nginx:** Kubernetes ingress controller implementation, which uses Nginx as its reverse proxy and load balancer.
* **cert manager:** A certificate controller that obtains certificates from issuers and periodically renews them. MuShop uses it to issue and manage SSL certificates when you configure your own domain name with MuShop.
* **Jenkins:** A popular CI/CD tool. It is not installed by default, but it can be enabled. If it is installed, it will be preconfigured to run builds by spinning up a pod to run the build and tearing it down after the build is complete.

**Deploying MuShop**

MuShop is designed to showcase multiple deployment options for your cloud native applications in OCI. The process is fully automated and covers both infrastructure and application deployments because the sample is designed to be deployed with no prior knowledge about tools and services in OCI. When building your own applications, you should consider whether this is the right approach for you. Most enterprise organizations tend to separate infrastructure management and application development. In these cases, infrastructure teams might use automation tools such as Terraform or the OCI Resource Manager service, and application teams might be using continuous delivery tools such as Jenkins or the OCI DevOps service.

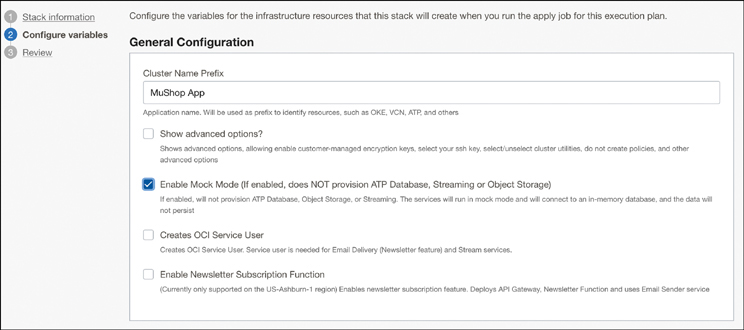
The default MuShop deployment itself is a simple but highly coordinated affair, and it is worthwhile to dive into the details of how the process works. As described in earlier sections, the infrastructure is managed by a Terraform configuration packaged as an OCI Resource Manager stack. The application is packaged as a Helm chart that can be deployed to a Kubernetes cluster using Helm. The process can be kicked off simply by using the Deploy button on the GitHub page. [Figure 10-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ch10fig04) shows the Deploy button on the GitHub page.



**Figure 10-4** Deploy Buttons Can Be Embedded Anywhere and Can Launch the OCI Resource Manager to Deploy Infrastructure and Applications

Clicking the button takes you to the OCI Resource Manager service where the stack will be imported. The button is simply an HTML link to the Resource Manager service, with a query parameter that points at the stack that needs to be deployed. In the case of MuShop, new versions are released as stacks (zipped Terraform configuration with a schema.yaml) through GitHub releases. You can create links or buttons like these that point to your stacks as well.

Some application configuration is exposed to the user, and these cases are captured by Terraform. Terraform passes these to the Helm chart using the values.yaml file used for the Helm chart. [Figure 10-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ch10fig05) and [Listing 10-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#list10_9) showcase how some of these Helm chart values are set from Terraform based on user input in the OCI Resource Manager service.



**Figure 10-5** The OCI Resource Manager Can Provide an Intuitive User Interface for the Terraform Code

**Listing 10-9** The OCI Resource Manager Supplies the User-Provided Values to the Terraform Code

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10_images.xhtml#f0401-01a)

# Create namespace mushop for the mushop microservices

resource "kubernetes\_namespace" "mushop\_namespace" {

metadata {

name = "mushop"

}

depends\_on = [oci\_containerengine\_node\_pool.oke\_node\_pool]

}

# Deploy mushop chart

resource "helm\_release" "mushop" {

name = "mushop"

chart = "../helm-chart/mushop"

namespace = kubernetes\_namespace.mushop\_namespace.id

wait = false

set {

name = "global.mock.service"

value = var.mushop\_mock\_mode\_all ? "all" : "none"

}

# ...[truncated for brevity]

}

When the Terraform configuration is applied, it uses the OCI provider for Terraform to create the basic infrastructure components, such as networks, databases, and the Kubernetes cluster itself. After the infrastructure components are created, the Terraform configuration uses the Kubernetes and Helm providers for Terraform to interact with the cluster that was created and installs the Helm chart to the cluster to complete the deployment.

You can access additional instructions regarding the deployment options and variations of MuShop on the documentation website.[**6**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_6a) This includes deploying with Istio[**7**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_7a) Service Mesh and integrating with OCI Logging and Analytics. Additionally, there is a variation of MuShop[**8**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_8a) where all services use Micronaut.[**9**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_9a)

**Summary**

This chapter took a deep dive into MuShop, a reference application that applies the concepts described throughout this book. From structuring source code for your microservices to exploring the various build practices and deployment models that improve development velocity, the example application (and its more than a dozen services, all built using a variety of technologies) showcases how you can effectively compose a complex application from a set of microservices and manage it effectively. This chapter also provided a walk-through of the deployment automation using Terraform and Helm charts, which can hopefully inspire your own Helm charts for your applications.

**References**

[1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_1) MuShop Source Code on GitHub: <https://github.com/oracle-quickstart/oci-cloudnative>

[2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_2) CapEx, which stands for capital expenditures, pertains to the financial resources a company puts into physical assets like properties, equipment, or infrastructure. These assets are anticipated to bring advantages to the company over a few years and are not regarded as expenses in the current period.

[3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_3) The Twelve-Factor Methodology: <https://12factor.net/>

[4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_4) Git: [https://git-scm.com](https://git-scm.com/)

[5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_5) GitHub Desktop: [https://desktop.github.com](https://desktop.github.com/)

[6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_6) MuShop Documentation: <https://oracle-quickstart.github.io/oci-cloudnative/>

[7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_7) Istio: <https://istio.io/>

[8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_8) MuShop Variation using Micronaut: <https://github.com/oracle-quickstart/oci-micronaut>

[9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ref10_9) Micronaut: <https://micronaut.io/>