NEBRASS LAMOUCHI

PLAYING WITH JAVA MICROSERVICES ON KUBERNETES AND OPENSHIFT

HANDS-ON BOOK FOR THE ABSOLUTE BEGINNER

First Edition

Playing with Java Microservices on Kubernetes and OpenShift

Nebrass Lamouchi

This book is for sale at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp

This version was published on 2020-01-14

ISBN 978-2-9564285-0-3

To Mom, it's impossible to thank you adequately for everything you've done.

To the soul of Dad, I miss you every day...

To Firass, you are the best, may God bless you...

To my sweetheart, since you've come into my life, there are so many new emotions and feelings begging to come out..

Contents

Acknowledgements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Preface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What this book covers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Part One: The Monolithics Era

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 1: Introduction to the Monolithic architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Introduction to an actual situation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Presenting the context

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

How to solve these issues ?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 2: Coding the Monolithic application

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Presenting our domain

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Use Case Diagram

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Class Diagram

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Sequence Diagram

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Coding the application

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Presenting the technology stack

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Java 8

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Maven

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Spring Boot

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

NetBeans IDE

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Implementing the Boutique

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Generating the project skull

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Creating the Persistence Layer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Cart

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

CartStatus

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Address

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Category

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Customer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Order

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

OrderItem

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Payment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Product

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

ProductStatus

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Review

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Creating the Service Layer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Typical Service: CartService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

AddressService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

CategoryService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

CustomerService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

OrderItemService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

OrderService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

PaymentService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Product Service

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

ReviewService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Creating the Web Layer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Typical RestController: CartResource

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

CategoryResource

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

CustomerResource

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

OrderItemResource

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

OrderResource

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

PaymentResource

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

ProductResource

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

ReviewResource

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Automated API documentation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Maven Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Java Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Hello World Swagger !

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 3 : Upgrading the Monolithic application

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Refactoring the database

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 4: Building & Deploying the Monolithic application

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Building the monolith

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Which package type: WAR vs JAR

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Build a JAR

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Build WAR if you

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Adding the Servlet Initializer Class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Exclude the embedded container from the WAR

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Change the package type to war in `pom.xml`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Package your application

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Deploying the monolith

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Deploying the JAR

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Deploying the WAR

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Part Two: The Microservices Era

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 5: Microservices Architecture Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

The Monolithic Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What is a Monolithic Architecture ?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Microservices Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What is a Microservices Architecture ?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What is really a Microservice?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Making the Switch

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 6: Splitting the Monolith: Bombarding the domain

What is Domain-Driven Design ?

Context

Domain

Model

Ubiquitous Language

Strategic Design

Bounded context

Bombarding La Boutique

Codebase

Dependencies and Commons

Entities

Example: Breaking Foreign Key Relationships

Refactoring Databases

Staging the Break

Transactional Boundaries

Try Again Later

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Abort the Entire Operation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Distributed Transactions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

So What to Do?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 7: Applying DDD to the code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Applying Bounded Contexts to Java Packages

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

The birth of the commons package

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

The birth of the configuration package

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Locating & breaking the BC Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Locating the BC Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Breaking the BC Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 8: Meeting the microservices concerns and patterns

Introduction

Cloud Patterns

Service discovery and registration

Externalized configuration

Circuit Breaker

Database per service

API gateway

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

CQRS

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Event sourcing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Log aggregation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Distributed tracing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Audit logging

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Application metrics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Health check API

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Security between services: Access Token

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What's next?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 9: Implementing the patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Externalized configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Generating the Config Server project skull

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Defining the properties of the Config Server

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Creating a centralized configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Enabling the Config Server engine

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 5: Run it !

Step 6: Spring Cloud Config Client

Service Discovery and Registration

Step 1: Generating the Eureka Server project skull

Step 2: Defining the properties of the Eureka Server

Step 3: Creating the main configuration of the Eureka Server

Step 4: Enabling the Eureka Server engine

Step 5: Update the global application.yml of the Config Server

Step 6: Run it !

Step 7: Client Side Load Balancer with Ribbon

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Distributed tracing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Getting the Zipkin Server

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Listing the dependencies to add to our microservices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Sleuth

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Zipkin Client

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Health check API

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Circuit Breaker

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Add the Maven dependency to your project

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Enable the circuit beaker

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Apply timeout and fallback method

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Enable the Hystrix Stream in the Actuator Endpoint

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 5: Monitoring Circuit Breakers using Hystrix Dashboard

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

API Gateway

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Generating the API Gateway project skull

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Enable the Zuul Capabilities

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Defining the route rules

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Enabling API specification on gateway using Swagger2

Step 5: Run it!

Log aggregation and analysis

Step 1: Installing Elasticsearch

Step 2: Installing Kibana

Step 3: Installing & Configuring Logstash

Installing Logstash

Configuring Logstash

Run it !

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Enabling the Logback features

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Adding Logback libraries to our microservices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Adding Logback configuration file to our microservices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 5: Adding the Logstash properties to the Config Server

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 6: Attending the Kibana party

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 10: Building the standalone microservices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Global Architecture Big Picture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Implementing the µServices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Before starting

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Generating the Commons project skull

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Moving Code from our monolith

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Moving Code from our monolith

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Building the project

The Product Service

Step 1: Generating the Product Service project skull

Step 2: Swagger 2

Step 3: Application Configuration

Step 4: Logback

Step 5: Activating the Circuit Beaker capability

Step 6: Moving Code from our monolith

The Order Service

Step 1: Generating the Order Service project skull

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Swagger 2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Application Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Logback

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 5: Activating the Circuit Beaker capability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 6: Moving Code from our monolith

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

The Customer Service

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Generating the Customer Service project skull

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Swagger 2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Application Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Logback

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 5: Activating the Circuit Beaker capability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 6: Moving Code from our monolith

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Part Three: Containers & Cloud Era

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 11. Getting started with Docker

Introduction to containerization

Our applications are deployed as a WAR or a JAR file to runtime environment, which can be a dedicated physical machine (server) or a virtual machine.

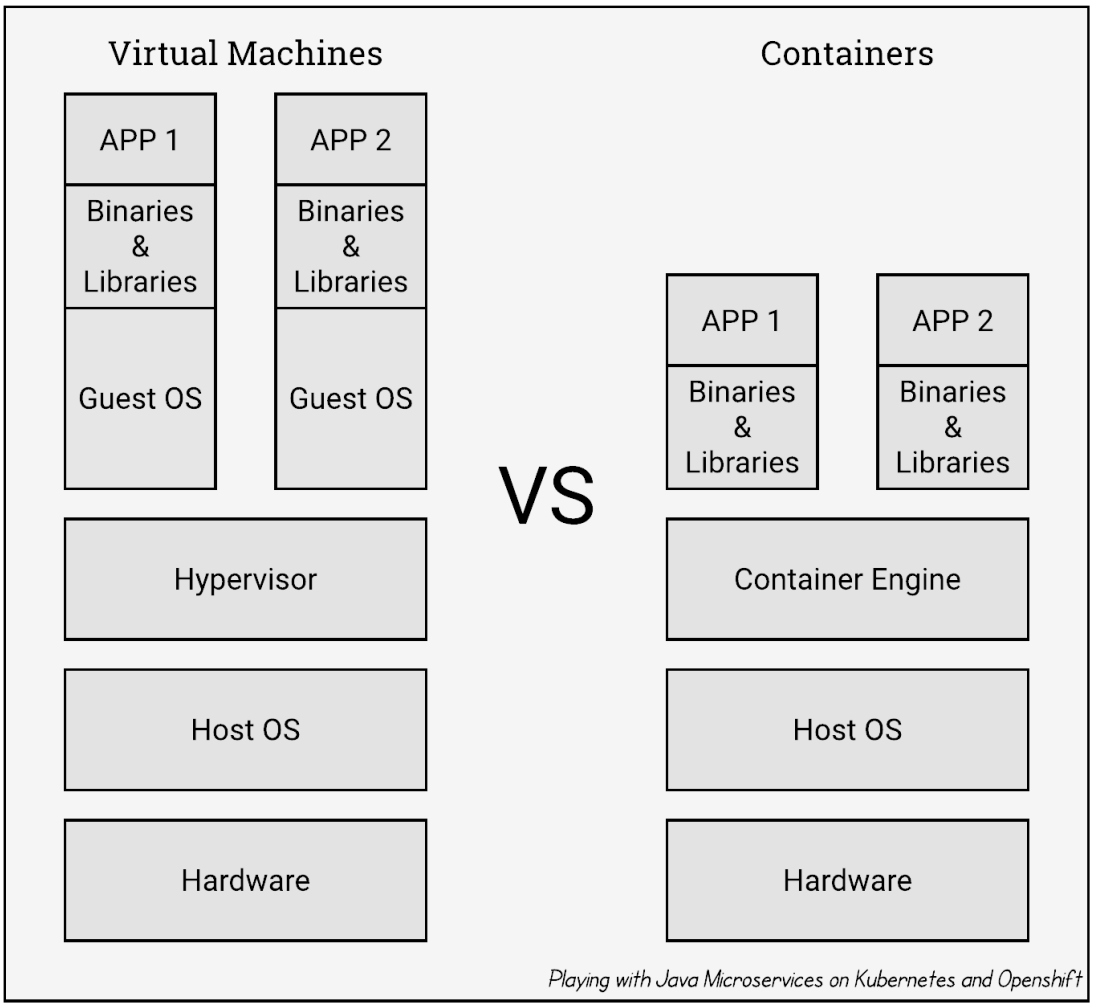
The major risk that encounters software application, is updates of the runtime that can break the application. For example, an OS update might include many updates, including libraries that can affect the running application with incompatible updates.

Generally, there are other software (database, proxy, etc..) that coexist with our application on the same host. They are all sharing the same OS and libraries. So, even if an update didn't cause any crash directly for the application, maybe there will be one for any of the other services.

Althought this updates are risky, we cannot deny them because of their interest in matter of security and stability, thru the provided bug fixes and enhances in updates. But, we can do some tests to our application and its context to see if updates will cause regressions or not. This task can be heavy especially if our application is huge.

Don't be sad!! There is a solution, we can use **containers**, which are a kind of isolated partition inside a single operating system. They provide many of the same benefits as virtual machines, such as security, storage, and network isolation, while they require far fewer hardware resources. **Containers** are great because they are quicker to launch and to terminate.

Containerization allows isolating and tracking resources utilization. This isolation protects our application from many risks linked with host OS update.



Container vs Virtual Machine

Containers have many benefits for both developers and system administrators.

- **Consistent Environment**:

Containers give developers the ability to create predictable environments that are isolated from other applications. Containers can also include software dependencies needed by the application, such as specific versions of programming language runtimes and other software libraries. From the developer's perspective, all this is guaranteed to be consistent no matter where the application is ultimately deployed. All this translates to productivity: developers and IT Ops teams spend less time debugging and diagnosing differences in environments, and more time shipping new functionality for users. And it means fewer bugs since developers can now make assumptions in dev and test environments they can be sure will hold true in production.

- **Run Anywhere**:

Containers are able to run virtually anywhere, greatly easing development and deployment: on Linux, Windows, and Mac operating systems; on virtual machines or bare metal; on a developer's machine or in data centers on-premises; and of course, in the public cloud. Wherever you want to run your software, you can use containers.

- **Isolation**:

Containers virtualize CPU, memory, storage, and network resources at the OS-level, providing developers with a sandboxed view of the OS logically isolated from other applications.

Finally, containers boost the microservices development approach because they provide a lightweight and reliable environment to create and run services that can be deployed to a production or development environment without the complexity of a multiple machine environment.

There are many container formats available. Docker is a popular, open-source container format.

Introducing Docker

What is Docker ?

Docker is a platform for developers and sysadmins to **develop**, **deploy**, **and run** applications with containers. The use of Linux containers to deploy applications is called *containerization*.

Docker logo

Containerization is increasingly popular because containers are:

- Flexible: Even the most complex applications can be containerized.
- Lightweight: Containers leverage and share the host kernel.
- Interchangeable: You can deploy updates and upgrades on-the-fly.
- Portable: You can build locally, deploy to the cloud, and run anywhere.
- Scalable: You can increase and automatically distribute container replicas.
- Stackable: You can stack services vertically and on-the-fly.

Images and containers

A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application–the code, a runtime, libraries, environment variables, and configuration files.

A **container** is a runtime instance of an image–what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, `docker ps`, just as you would in Linux.

Installation and first hands-on

Installation

To start playing with Docker, we need to install Docker. You can grab the version that is compatible with your platform from this URL: https://docs.docker.com/install/[1];

But when you will come on that page, you will find two versions: **Docker CE** and **Docker EE**; So which one you need ?

- **Docker CE**: Docker Community Edition is ideal for developers and small teams looking to get started with Docker and experimenting with container-based apps. Docker CE has three types of update channels, **stable**, **test**, and **nightly**:
  - **Stable** gives you latest releases for general availability.
  - **Test** gives pre-releases that are ready for testing before general availability.
  - **Nightly** gives you latest builds of work in progress for the next major release.
- **Docker EE**: Docker Enterprise Edition is designed for enterprise development and IT teams who build, ship, and run business-critical applications in production and at scale. Docker EE is integrated, certified, and supported to provide enterprises with the most secure container platform in the industry.

The installation of Docker is very easy, and does not need a tutorial to be done :)

After installation, to check if **Docker** is installed, we can for example start by checking the installed version by running the command `docker version`:

[1] https://docs.docker.com/install/

```
Client: <2>
 Version:           18.06.0-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        0ffa825
 Built:             Wed Jul 18 19:05:26 2018
 OS/Arch:           darwin/amd64
 Experimental:      false

Server: <1>
 Engine:
  Version:          18.06.0-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.3
  Git commit:       0ffa825
  Built:            Wed Jul 18 19:13:46 2018
  OS/Arch:          linux/amd64
  Experimental:     true
```

1. The **Docker daemon** listens for **Docker API** requests and manages Docker objects. A daemon can also communicate with other daemons to manage Docker services.
2. The **Docker client** is the primary way that many Docker users interact with Docker. When you run commands, the client sends these commands to the **Docker daemon**, which carries them out. The docker command uses the **Docker API**. The Docker client can communicate with more than one Docker daemon.

Docker uses a **client-server** architecture. The Docker client talks to the **Docker daemon** (`dockerd`), which does the heavy lifting of building, running, and distributing your Docker containers. The **Docker client and daemon** can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

Run your first container

We will start by running the `hello-world` image:

```
docker run hello-world
```

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9db2ca6ccae0: Pull complete
Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the “hello-world” image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

What is Docker Hub?

Docker Hub is a cloud-based registry service which allows you to link to code repositories, build your images and test them, stores manually pushed images, so you can deploy images to your hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

Docker Hub provides the following major features:

- **Image Repositories**: Find and pull images from community and official libraries, and manage, push to, and pull from private image libraries to which you have access.
- **Automated Builds**: Automatically create new images when you make changes to a source code repository.
- **Webhooks**: A feature of Automated Builds, Webhooks let you trigger actions after a successful push to a repository.
- **Organizations**: Create work groups to manage access to image repositories.
- **GitHub and Bitbucket Integration**: Add the Hub and your Docker Images to your current workflows.

To list the existing local Docker images:

```
docker image ls
```

```
REPOSITORY          TAG         IMAGE ID            CREATED             SIZE
hello-world         latest      2cb0d9787c4d        4 weeks ago         1.85kB
```

The **Image ID** is a random generated HEX value to identify an **Image**.

To list all the Docker containers:

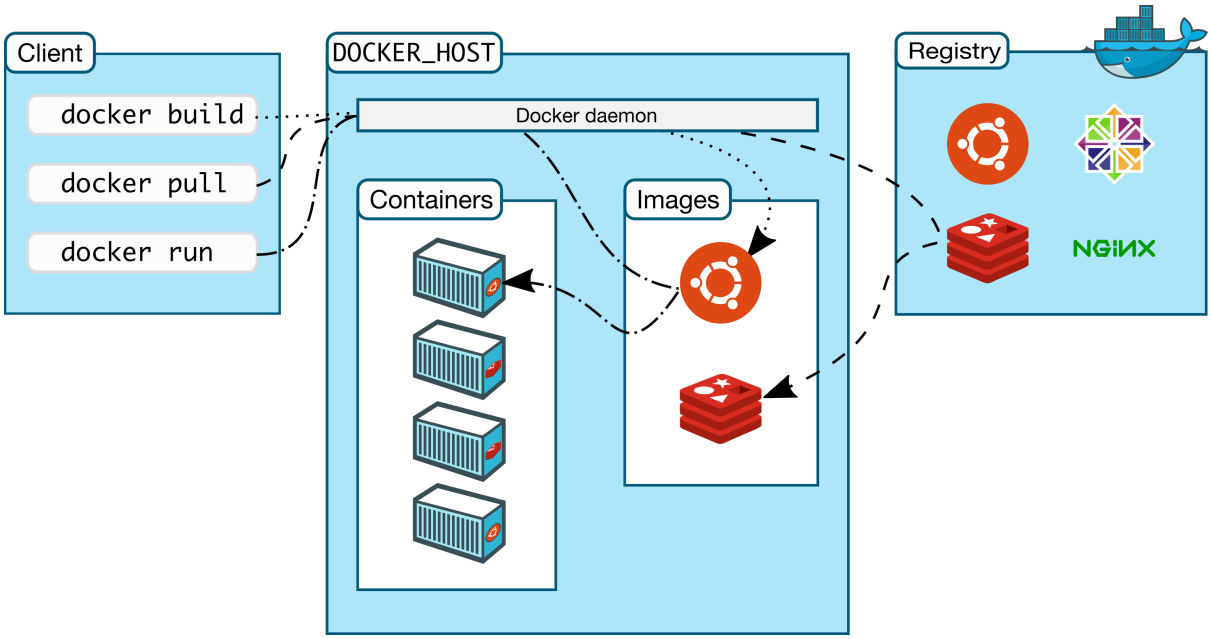
```
docker container ls --all
```

```
CONTAINER ID<1>  IMAGE          COMMAND     CREATED           STATUS                      NAMES<2>
54f4984ed6a8     hello-world    "/hello"    20 seconds ago    Exited (0) 9 seconds ago    suzyland
```

1. The **Container ID** is a random generated HEX value to identify an **Container**.
2. The **Container Name** that we got here is a randomly generated string name that the Docker Daemon created for us, as we didn't specify one.

Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.



Docker Ecosystem Architecture diagram - The offcial Docker documentation

The Docker daemon

The Docker daemon (`dockerd`) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (`docker`) is the primary way that many Docker users interact with Docker. When you use commands such as `docker run`, the client sends these commands to `dockerd`, which carries them out. The `docker` command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker registries

A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR).

When you use the `docker pull` or `docker run` commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the `ubuntu` image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Containers

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Docker Machine

Docker Machine is a tool that lets you install Docker Engine on virtual hosts, and manage the hosts with docker-machine commands. You can use Machine to create Docker hosts on your local

Mac or Windows box, on your company network, in your data center, or on cloud providers like Azure, AWS, or Digital Ocean.

Using `docker-machine` commands, you can start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon, and configure a Docker client to talk to your host.

Point the Machine CLI at a running, managed host, and you can run `docker` commands directly on that host. For example, run `docker-machine env default` to point to a host called `default`, follow on-screen instructions to complete `env` setup, and run `docker ps`, `docker run hello-world`, and so forth.

Machine was the only way to run Docker on Mac or Windows previous to Docker v1.12. Starting with the beta program and Docker v1.12, Docker for Mac and Docker for Windows are available as native apps and the better choice for this use case on newer desktops and laptops. The Docker Team recommends to try out these new apps. The installers for Docker for Mac and Docker for Windows include Docker Machine, along with Docker Compose (don't be afraid, we will see it..).

Why should I use it?

Docker Machine enables you to provision multiple remote Docker hosts on various flavors of Linux.

Additionally, Machine allows you to run Docker on older Mac or Windows systems, as described in the previous topic.

Docker Machine has these two broad use cases:

Using Docker on older machines

I have an older desktop system and want to run Docker on Mac or Windows:

If you work primarily on an older Mac or Windows laptop or desktop that doesn't meet the requirements for the new Docker for Mac and Docker for Windows apps, then you need Docker Machine run Docker Engine locally. Installing Docker Machine on a Mac or Windows box with the Docker Toolbox installer provisions a local virtual machine with Docker Engine, gives you the ability to connect it, and run docker commands.

Provision remote Docker instances

Docker Engine runs natively on Linux systems. If you have a Linux box as your primary system, and want to run docker commands, all you need to do is download and install Docker Engine. However, if you want an efficient way to provision multiple Docker hosts on a network, in the cloud or even locally, you need Docker Machine.

Whether your primary system is Mac, Windows, or Linux, you can install Docker Machine on it and use docker-machine commands to provision and manage large numbers of Docker hosts. It automatically creates hosts, installs Docker Engine on them, then configures the docker clients. Each managed host ("machine") is the combination of a Docker host and a configured client.

What's the difference between Docker Engine and Docker Machine?

When people say "Docker" they typically mean **Docker Engine**, the client-server application made up of the Docker daemon, a REST API that specifies interfaces for interacting with the daemon, and a command line interface (CLI) client that talks to the daemon (through the REST API wrapper). Docker Engine accepts `docker` commands from the CLI, such as `docker run <image>`, `docker ps` to list running containers, `docker image ls` to list images, and so on.

Docker Machine is a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them). Typically, you install Docker Machine on your local system. Docker Machine has its own command line client `docker-machine` and the Docker Engine client, `docker`. You can use Machine to install Docker Engine on one or more virtual systems. These virtual systems can be local (as when you use Machine to install and run Docker Engine in VirtualBox on Mac or Windows) or remote (as when you use Machine to provision Dockerized hosts on cloud providers). The Dockerized hosts themselves can be thought of, and are sometimes referred to as, managed "***machines***".

Diving into Docker Containers

Introduction

The best way to learn more about Docker, is to write an app in the Docker way :)

Your new development environment

In the past, if you were to start writing a Java app, your first order of business was to install a Java runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Java runtime as an image, no installation necessary. Then, your build can include the base Java image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These portable images are defined by something called a `Dockerfile`.

Define a container with Dockerfile

`Dockerfile` defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to "copy in" to that environment. However, after doing that, you can expect that the build of your app defined in this `Dockerfile` behaves exactly the same wherever it runs.

`Dockerfile`

Create an empty directory. Change directories (`cd`) into the new directory, create a file called `Dockerfile`, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

```
# Use an OpenJDK Runtime as a parent image
FROM openjdk:8-jre-alpine

# Add Maintainer Info
LABEL maintainer="lnibrass@gmail.com"

# Define environment variables
ENV SPRING_OUTPUT_ANSI_ENABLED=ALWAYS \
    JAVA_OPTS=""

# Set the working directory to /app
WORKDIR /app

# Copy the executable into the container at /app
ADD *.jar app.jar

# Make port 8080 available to the world outside this container
EXPOSE 8080

# Run app.jar when the container launches
CMD ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app/app.jar"]
```

This `Dockerfile` refers to a `app.jar` that we haven't created yet. This file is the executable JAR of our sample application.

Create sample application

We will create `sample-app`: a new Spring Boot application using Spring Initializr[2] we need to add the following dependencies:

- Web
- Actuator

Then, we will create a **Hello World RestController** in the main class:

```
@RestController
@SpringBootApplication
public class SampleAppApplication {

    public static void main(String[] args) {
        SpringApplication.run(SampleAppApplication.class, args);
    }

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello World!";
    }
}
```

[2] https://start.spring.io/

When we compile this application using `mvn clean install`, our JAR, will be available in the `target` folder.

Next, we will create the `Dockerfile` in the same folder as the JAR file. We need to do this so the **ADD** command can access the JAR file. In the `target` folder, run this command will show only the files in the current folder:

```
ls -p | grep -v /

Dockerfile
sample-app-0.0.1-SNAPSHOT.jar
sample-app-0.0.1-SNAPSHOT.jar.original
```

Now we verified that our `Dockerfile` and our JAR are together, we will need to build the Docker Image:

Now run the build command. This creates a Docker image, which we're going to tag using `-t` so it has a significant name:

```
docker build -t greatest-hello .
```

You will see the image building logs:

```
Sending build context to Docker daemon  17.57MB
Step 1/7 : FROM openjdk:8-jre-alpine <1>
8-jre-alpine: Pulling from library/openjdk <1>
8e3ba11ec2a2: Pull complete  <1>
311ad0da4533: Pull complete  <1>
391a6a6b3651: Pull complete  <1>
Digest: sha256:1bed44170948277881d88481ecbd07317eb7bae385560a9dd597bbfe02782766 <1>
Status: Downloaded newer image for openjdk:8-jre-alpine <1>
 ---> ccfb0c83b2fe <1>
Step 2/7 : LABEL maintainer="lnibrass@gmail.com"
 ---> Running in 0fe56f45fda0
Removing intermediate container 0fe56f45fda0
 ---> 6768f6729824 <2>
Step 3/7 : ENV SPRING_OUTPUT_ANSI_ENABLED=ALWAYS     JAVA_OPTS=""
 ---> Running in 302927709e3a
Removing intermediate container 302927709e3a
 ---> c270becb4a05 <2>
Step 4/7 : WORKDIR /app
 ---> Running in 8229163b4d00
Removing intermediate container 8229163b4d00
 ---> 695621f3f97a <2>
Step 5/7 : ADD *.jar app.jar
 ---> 5f9306b5b25a
Step 6/7 : EXPOSE 8080
 ---> Running in 2163da2c6d60
Removing intermediate container 2163da2c6d60
 ---> f18ed4e056b4 <2>
Step 7/7 : CMD ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app/app.jar"]
 ---> Running in b481d6bb1665
Removing intermediate container b481d6bb1665
 ---> 668aa1b9b527 <2>
Successfully built 668aa1b9b527 <3>
Successfully tagged greatest-hello:latest <4>
```

1. As Docker didn't find the `openjdk:8-jre-alpine` image locally, it proceeds to downling it from the **Docker Hub**.
2. Every instruction in the `Dockerfile` is built in a dedicated step; and it generates a separated **LAYER** in the **IMAGE**; the shown **HEX code** at the end of each **STEP** is the **ID** of the **LAYER**.
3. The **IMAGE ID**.
4. Our built image is tagged with **greatest-hello:latest**; we specified only the name (`greatest-hello`) and Docker added for us automatically the latest version tag.

Where is your built image? It's in your machine's local Docker image registry:

```
docker image ls
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
greatest-hello      latest              668aa1b9b527        5 minutes ago       101MB
openjdk             8-jre-alpine        ccfb0c83b2fe        4 weeks ago         83MB
```

We see that we have locally two images; the `openjdk` is the base image; and `greatest-hello` which is our built image.

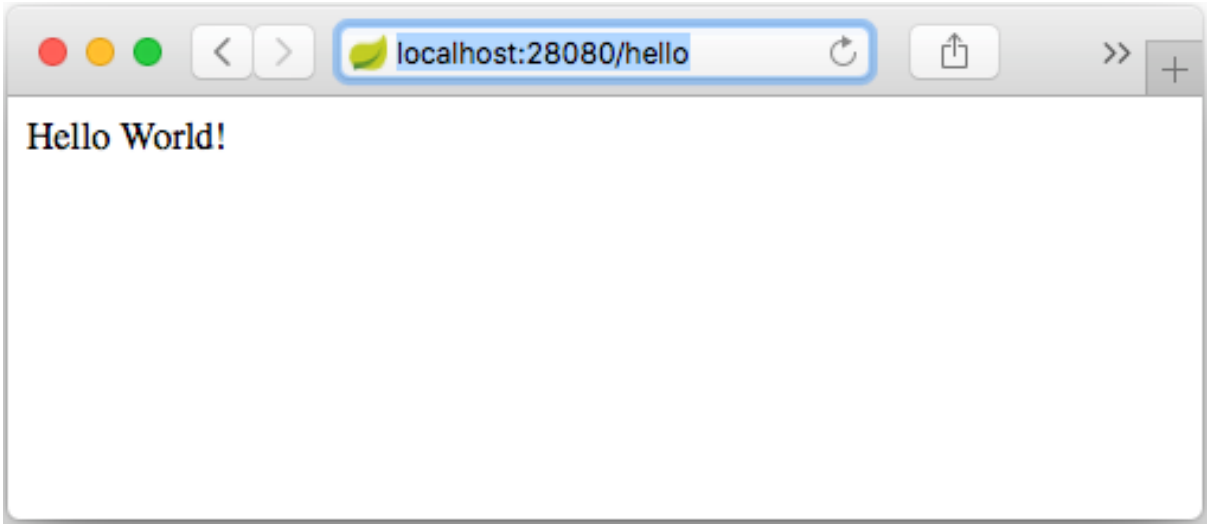
Run the app

Run the app, mapping your machine's port `28080` to the container's published port `8080` using `-p`:

```
docker run -p 28080:8080 greatest-hello
```

You should see the Spring Boot application starting log. There you will see a message mentionning `Tomcat started on port(s): 8080 (http) with context path ''`. But that message is coming from inside the container, which doesn't know you mapped port `8080` of that container to `28080`, making the correct URL `http://localhost:28080`.

Go to `http://localhost:28080/hello` in a web browser, you will see the response of our **RestController**.



Response of the REST service from the container

If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of `localhost`. For example, `http://192.168.99.100:28080/hello`. To find the IP address, use the command `docker-machine ip`.

You can also use the `curl` command in a shell to view the same content.

```
curl http://localhost:28080/hello

Hello World!%
```

What is **cURL**

cURL is a computer software project providing a library and command-line tool for transferring data using various protocols. The cURL project produces two products, **libcurl** and **cURL**. It was first released in 1997. The name stands for "Client URL".
libcurl: is a free client-side URL transfer library, supporting cookies, DICT, FTP, FTPS, Gopher, HTTP (with HTTP/2 support), HTTP POST, HTTP PUT, HTTP proxy tunneling, HTTPS, IMAP, Kerberos, LDAP, POP3, RTSP, SCP, and SMTP. The library supports the file URI scheme, SFTP, Telnet, TFTP, file transfer resume, FTP uploading, HTTP form-based upload, HTTPS certificates, LDAPS, proxies, and user-plus-password authentication.
cURL: cURL is a command-line tool for getting or sending files using URL syntax. Since cURL uses libcurl, it supports a range of common Internet protocols, currently including HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, LDAP, DAP, DICT, TELNET, FILE, IMAP, POP3, SMTP and RTSP

This port remapping of `28080:8080` demonstrates the difference between `EXPOSE` within the `Dockerfile` and what the `publish` value is set to when running `docker run -p`. In later steps, map port 28080 on the host to port 8080 in the container and use `http://localhost`.

In the console where you run the Docker image, hit `CTRL+C` in your terminal to quit.

On Windows, explicitly stop the container

On Windows systems, `CTRL+C` does not stop the container. So, first type `CTRL+C` to get the prompt back (or open another shell), then type `docker container ls` to list the running containers, followed by `docker container stop <Container NAME or ID>` to stop the container. Otherwise, you get an error response from the daemon when you try to re-run the container in the next step.

Now let's run the app in the background, in **detached mode**:

```
docker run -d -p 28080:8080 greatest-hello


397d8be9c7db7f91c94e139a3673210df9547d12fd0f93886b52ced0c51e4a04
```

Here we got the long container ID for our app and then are kicked back to your terminal. Our container is running in the background. We can also see the abbreviated container ID with `docker container ls` (and both work interchangeably when running commands):

```
docker container ls
```

```
CONTAINER ID  IMAGE           COMMAND     CREATED  STATUS  PORTS            NAMES
397d8be9c7db  greatest-hello  "java -D…"  47s      Up      28080->8080/tcp  tom_lee
```

Notice that `CONTAINER ID` matches what's on `http://localhost:28080`.

Now use `docker container stop` to end the process, using the `CONTAINER ID`, like so:

```
docker stop 397d8be9c7db
```

Share your image

To demonstrate the portability of what we just created, let's upload our built image and run it somewhere else. After all, you need to know how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The `docker` CLI uses Docker's public registry by default.

We use Docker's public registry here just because it's free and pre-configured, but there are many public ones to choose from, and you can even set up your own private registry using Docker Trusted Registry.

Log in with your Docker ID

If you don't have a Docker account, sign up for one at https://hub.docker.com/[3]. Make note of your username.

Log in to the Docker public registry on your local machine.

```
docker login
```

[3] https://hub.docker.com/

Tag the image

The notation for associating a local image with a repository on a registry is username/repository:tag. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as get-started:part2. This puts the image in the get-started repository and tag it as part2.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image uploads to your desired destination. The syntax of the command is:

```
docker tag image username/repository:tag
```

For example:

```
docker tag greatest-hello nebrass/docker-get-started:v2
```

Run the command: `docker image ls` to see your newly tagged image:

```
docker image ls
```

```
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
greatest-hello              latest              668aa1b9b527        43 minutes ago      101MB
nebrass/docker-get-started  v2                  668aa1b9b527        43 minutes ago      101MB
openjdk                     8-jre-alpine        ccfb0c83b2fe        4 weeks ago         83MB
```

Publish the image

Upload your tagged image to the repository:

```
docker push username/repository:tag
```

Once complete, the results of this upload are publicly available. If you log in to Docker Hub, you see the new image there, with its pull command.

You need to be authenticated using `docker login` command before pushing images.

We will push our `nebrass/docker-get-started:v2` tagged image:

```
docker push nebrass/docker-get-started:v2

The push refers to repository [docker.io/nebrass/docker-get-started]
87233a64097d: Pushed
b8becc4abdd9: Pushed
8bc7bbcd76b6: Mounted from library/openjdk
73046094a9b8: Mounted from library/openjdk
v2: digest: sha256:9b4c21e8445987396935342185868337df9cffbf11ac0500d40972f221f88d9e size: 1365
```

Pull and run the image from the remote repository

From now on, you can use docker run and run your app on any machine with this command:

```
docker run -p 28080:8080 username/repository:tag
```

If the image isn't available locally on the machine, Docker pulls it from the repository.

```
docker run -p 28080:8080 nebrass/docker-get-started:v2

Unable to find image 'nebrass/docker-get-started:v2' locally
v2: Pulling from nebrass/docker-get-started
8e3ba11ec2a2: Pull complete
311ad0da4533: Pull complete
e290204e3b0c: Pull complete
4e78951b0f63: Pull complete
Digest: sha256:9b4c21e8445987396935342185868337df9cffbf11ac0500d40972f221f88d9e
Status: Downloaded newer image for nebrass/docker-get-started:v2
...
```

No matter where `docker run` executes, it pulls your image, along with `openjdk` and all the dependencies, and runs your code. It all travels together in a neat little package, and you don't need to install anything on the host machine for Docker to run it.

Automating the Docker image build

Hey ! As you see, we put our `Dockerfile` file in the `target/` folder, to be in the neighborhood of the our **executable** JAR file. But this folder will be deleted each time that we do an `mvn clean install` :D

A solution can be putting the `Dockerfile` in the root of the directory of our project, and we need to edit the `ADD` command in the `Dockerfile` to include the `target` folder:

```
ADD target/*.jar app.jar
```

In the development process, we will be editing and builing the application many times. The **Maven** integration in the **IDE** (Eclipse, NetBeans, IntelliJ, etc..) can do the build just with one click, or even automatically :)

Our **Docker Image** also needs to be updated, but if it will be rebuilt manually, this can be heavy, and so heavy if we need to test our JAR in the **Docker Container** every time the application is updated.

Hopefully, we can automate this process using **Maven** :D and we have many available plugins doing this.

Spotify Maven plugin

Spotify has a Maven plugin called **Dockerfile-maven-plugin** which help to seamlessly integrate Docker with Maven.

The **Dockerfile-maven-plugin** needs that the `Dockerfile` be placed in the project root folder.

To use it, you'll need to add the plugin to the `plugins` of the `build` section in the `pom.xml` file:

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>dockerfile-maven-plugin</artifactId>
    <version>1.4.3</version>
    <configuration>
        <repository>nebrass/${project.artifactId}</repository> <1>
        <tag>${project.version}</tag> <2>
    </configuration>
    <executions>
        <execution>
            <id>default</id> <3>
            <phase>install</phase> <3>
            <goals> <3>
                <goal>build</goal> <3>
                <goal>push</goal> <3>
            </goals>
        </execution>
    </executions>
</plugin>
```

1. The built Docker Image repository and Image Name
2. The built Docker Image Tag
3. We have registered the `dockerfile:build` goal to the `install` phase of **Maven** build life cycle using the `<executions>` tag. So whenever you run `mvn install`, the `build` goal of `dockerfile-maven-plugin` is executed, and your docker image is built. We can add other goals here, for example the `push` goal, to push the built image to **Docker Hub**.

Dockerfile-maven-plugin
The `dockerfile-maven-plugin` uses the authentication information stored in any configuration files `~/.dockercfg` or `~/.docker/config.json` to push the **Docker Image** to the specified Docker repository. These configuration files are created when you login to docker via docker login.

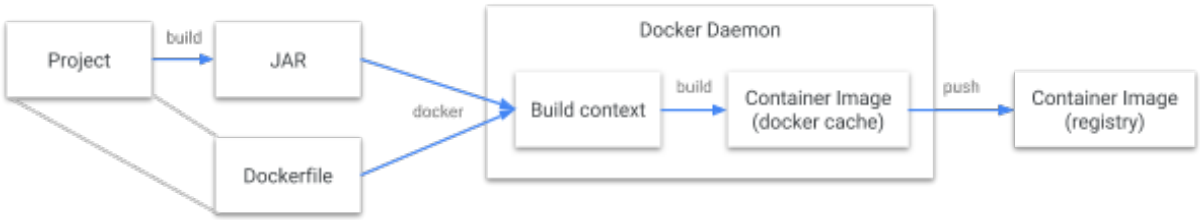
You can also specify authentication details in Maven `settings.xml` or `pom.xml` files.

GoogleContainerTools Jib Maven plugin

Jib is a Maven plugin for building Docker and OCI images for Java applications from Google.

Jib is a fast and simple container image builder that handles all the steps of packaging your application into a container image. **It does not require you to write a Dockerfile or have docker installed**, and it is directly integrated into **Maven** by just adding the plugin to your build and you'll have your Java application containerized in no time.

Docker build flow:



Docker build flow

Jib build flow:

Jib build flow

Jib is available as plugins for **Maven** and **Gradle** and requires minimal configuration. Simply add the plugin to your build definition and configure the target image. Jib also provides additional rules for building an image to a Docker daemon if you need it.

If you are building to a private registry, make sure to configure Jib with credentials for your registry. Learn more about this in the official Jib Documentation[4]

To use it, you'll need to add the plugin to the `plugins` of the `build` section in the `pom.xml` file:

```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>0.9.10</version>
    <configuration>
        <to>
            <image>nebrass/${project.artifactId}:${project.version}</image>
        </to>
    </configuration>
</plugin>
```

[4] https://goo.gl/gDs66G

The `<image>` tag specifies the repository name, the image name and the image tag. You can specify your image registry as a prefix in the image name:

- `gcr.io`
- `aws_account_id.dkr.ecr.region.amazonaws.com`
- `registry.hub.docker.com`

Pushing/pulling from private registries require authorization credentials. These can be retrieved using Docker credential helpers or defined in your Maven settings. If you do not define credentials explicitly, Jib will try to use credentials defined in your Docker config or infer common credential helpers.

Build your image

Build your container image to a container image registry with:

```
mvn compile jib:build
```

Subsequent builds are much faster than the initial build.

Build to Docker daemon

Jib can build your image directly to a Docker daemon. This uses the docker command line tool and requires that you have **Docker** available on your `PATH`.

```
mvn compile jib:dockerBuild
```

That's all tale !!

Meeting the Docker Services

In a distributed application, different pieces of the app are called "services." For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just "containers in production." A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Luckily it's very easy to define, run, and scale services with the Docker platform – just write a `docker-compose.yml` file.

Your first docker-compose.yml file

A `docker-compose.yml` file is a YAML file that defines how Docker containers should behave in production.

Save this file as `docker-compose.yml` wherever you want. Be sure you have pushed the image you created in Part 2 to a registry, and update this `.yml` by replacing `username/repo:tag` with your image details.

Example of a docker-compose.yml file

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "28080:8080"
    networks:
      - webnet
networks:
  webnet:
```

This `docker-compose.yml` file tells Docker to do the following:

- Pull the image we uploaded in step 2 from the registry.
- Run 5 instances of that image as a service called `web`, limiting each one to use, at most, 10% of the CPU (across all cores), and 50MB of RAM.
- Immediately restart containers if one fails.
- Map port 28080 on the host to web's port 8080.
- Instruct `web`'s containers to share port 8080 via a load-balanced network called `webnet`.
- Define the `webnet` network with the default settings (which is a load-balanced overlay network).

Run your new load-balanced app

Before we can use the docker stack deploy command we first run:

```
docker swarm init
```

If you don't run docker swarm init you get an error that "this node is not a swarm manager."

Ok but what is Swarm?

A swarm is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you're used to, but now they are executed on a cluster by a swarm manager. The machines in a swarm can be physical or virtual. After joining a swarm, they are referred to as nodes.

A swarm is made up of multiple nodes, which can be either physical or virtual machines. The basic concept is simple enough: run docker swarm init to enable swarm mode and make your current machine a swarm manager, then run docker swarm join on other machines to have them join the swarm as workers. Choose a tab below to see how this plays out in various contexts. We use VMs to quickly create a two-machine cluster and turn it into a swarm.

Now let's run it. You need to give your app a name. Here, it is set to getstartedlab:

```
docker stack deploy -c docker-compose.yml getstartedlab
```

Our single service stack is running 5 container instances of our deployed image on one host. Let's investigate.

Get the service ID for the one service in our application:

```
docker service ls
```

```
ID            NAME                MODE        REPLICAS  IMAGE                             PORTS
voz48hmxjg3g  getstartedlab_web  replicated  5/5       nebrass/docker-get-started:v2  *:28080->8080/tcp
```

Look for output for the `web` service, prepended with your app name. If you named it the same as shown in this example, the name is `getstartedlab_web`. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

A single container running in a service is called a **task**. Tasks are given unique IDs that numerically increment, up to the number of `replicas` you defined in `docker-compose.yml`. List the tasks for your service:

```
docker service ps getstartedlab_web
```

Tasks also show up if you just list all the containers on your system, though that is not filtered by service:

```
docker container ls -q
```

Scale the app

You can scale the app by changing the `replicas` value in `docker-compose.yml`, saving the change, and re-running the `docker stack deploy` command:

```
docker stack deploy -c docker-compose.yml getstartedlab
```

Docker performs an in-place update, no need to tear the stack down first or kill any containers.

Now, re-run `docker container ls -q` to see the deployed instances reconfigured. If you scaled up the replicas, more tasks, and hence, more containers, are started.

Remove the app and the swarm

Remove the app from the docker stack:

```
docker stack rm getstartedlab
```

Remove the swarm:

```
docker swarm leave --force
```

It's as easy as that to stand up and scale your app with Docker. You've taken a huge step towards learning how to run containers in production. Up next, you learn how to run this app as a bonafide swarm on a cluster of Docker machines.

Containerizing our microservices

Now we got started with Docker, we need to dockerize our microservices. We need to add the **Jib Maven Plugin**:

```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>0.9.10</version>
    <configuration>
        <to>
            <image>nebrass/${project.artifactId}:${project.version}</image>
        </to>
    </configuration>
</plugin>
```

And we need to build the Docker images:

```
mvn compile jib:dockerBuild
```

After building the Docker Images, just do `docker images` and you will get:

```
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nebrass/customer-service	0.0.1-SNAPSHOT	a11824661356	5 minutes ago	148MB
nebrass/order-service	0.0.1-SNAPSHOT	96d770656f1a	5 minutes ago	148MB
nebrass/hystrix-dashboard	0.0.1-SNAPSHOT	1aab18bc8798	5 minutes ago	125MB
nebrass/product-service	0.0.1-SNAPSHOT	fbc4fb89c8cd	5 minutes ago	147MB
nebrass/api-gateway	0.0.1-SNAPSHOT	73496f8bbf21	5 minutes ago	133MB
nebrass/configserver	0.0.1-SNAPSHOT	960d46621d0b	5 minutes ago	110MB
nebrass/eureka	0.0.1-SNAPSHOT	193bba74c1bb	5 minutes ago	128MB
openjdk	8-jre-alpine	0fe3f0d1ee48	4 days ago	83MB
openzipkin/zipkin	latest	639cba1daeb3	8 days ago	147MB

Now we need to run the images with the correct port :)

- Config Server

```
docker run --name config-server -d -p 8888:8888 nebrass/configserver:0.0.1-SNAPSHOT
```

- Eureka

```
docker run -d -p 8761:8761 nebrass/eureka:0.0.1-SNAPSHOT
```

This is will not work ! This configuration will be throwing a `java.net.ConnectException`:

```
...
INFO 1 --- [Thread-13] o.s.c.n.e.server.EurekaServerBootstrap : Eureka data center value eureka.datace\
nter is not set, defaulting to default
ERROR 1 --- [nfoReplicator-0] c.n.d.s.t.d.RedirectingEurekaHttpClient  : Request execution error

c.s.j.api.client.ClientHandlerException: java.net.ConnectException: Connection refused...
```

The exception thrown by Eureka is due that the program couldn't connect to the `Config-Server`, and the cause is very simple:
On startup, Eureka is requesting the `Config-Server` on the address declared in its `bootstrap.yml`, in the `spring.cloud.config.uri` property, which is `http://localhost:8888`.

So let's imagine how it's going on inside the container: The containerized Eureka application is located in a dedicated container; so when we are requesting the address `http://localhost:8888` inside the container, the localhost points on the the Eureka Container; while the `Config-Server` application is hosted on its own Container, and not on the current `localhost`.

So the solution is to make Eureka and the other containers connect to the `Config-Server` Container address, and not the classic `http://localhost:8888`. Don't be scared, we will not rewrite code and we will not change our code, we can resolve the problem using the environment variables and the great Spring Boot features to deal with them.

Injecting properties to Spring Boot Applications

We can pass/override properties easily using **System Environment Variables**. For example, if we want to pass/override the `spring.cloud.config.uri` property, we just have to assign the new value to the `SPRING_CLOUD_CONFIG_URI` environment variable. So, when the Spring Boot application starts, it parses the environment variables, and it gives priority to elements defined in the environment variables. So even a value is defined in `bootstrap.yml` or `application.properties`, if it an other value is passed in environment variable, this one is taken into consideration.

Docker offers us the ability to declare environment variables for the container on its creation:

```
docker run -e "env_var_name=env_var_value" image_name
```

If we name an environment variable without specifying a value, then the current value of the named variable is propagated into the container's environment.

So we can define the `spring.cloud.config.uri` property using the command:

```
docker run -d \
  -e SPRING_CLOUD_CONFIG_URI=http://CONFIG-SERVER-CONTAINER-IP:8888  \
  -p 8761:8761 nebrass/eureka:0.0.1-SNAPSHOT
```

We can get the value of `CONFIG-SERVER-CONTAINER-IP`, using the command:

```
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' config-server

172.17.0.2
```

So the command will look like:

```
docker run -d \
  -e SPRING_CLOUD_CONFIG_URI=http://172.17.0.2:8888 \
  -p 8761:8761 nebrass/eureka:0.0.1-SNAPSHOT
```

This is ok, but not so good. Containers keeps changing IPs and we will not inspect IPs every time. This is a heavy task and it's not productive.

Our lovely Docker gives us many great features that boost the performance and the productivity of our ecosystem, like the **Container Links**.

Container Links create environment variables which allow containers to communicate with each other. You can specify container links explicitly when you run a new container or edit an existing one.

So we can use the `links` in our command:

```
docker run --name eureka \
    --link config-server \ <1>
    -d -e SPRING_CLOUD_CONFIG_URI=http://config-server:8888 \ <2>
    -p 8761:8761 nebrass/eureka:0.0.1-SNAPSHOT
```

1. The same as writing `--link config-server=config-server`: makes the `config-server` container information, IP Address for example, available at our `eureka` container.
2. The `SPRING_CLOUD_CONFIG_URI` property points on a host called `config-server` that will be resolved to the Config Server Container IP Address. This is became possible using the linking mecanism.

> The `--link` flag is a legacy feature of Docker. It may eventually be removed. Unless you absolutely need to continue using it, we recommend that you use user-defined networks to facilitate communication between two containers instead of using `--link`. One feature that user-defined networks do not support that you can do with `--link` is sharing environmental variables between containers. However, you can use other mechanisms such as volumes to share environment variables between containers in a more controlled way.

- API Gateway

```
docker run --name api-gateway \
    --link config-server \
    -d -e SPRING_CLOUD_CONFIG_URI=http://config-server:8888 \
    -p 8222:8222 nebrass/api-gateway:0.0.1-SNAPSHOT
```

- Zipkin

```
docker run --name zipkin -d -p 9411:9411 openzipkin/zipkin
```

- Hystrix-dashboard

```
docker run --name hystrix-dashboard \
    --link config-server \
    -d -e SPRING_CLOUD_CONFIG_URI=http://config-server:8888 \
    -p 8988:8988 nebrass/hystrix-dashboard:0.0.1-SNAPSHOT
```

For `Product-Service`, `Order-Service` and `Customer-Service` we need to pass the **Zipkin Server Container IP Address** as we did for the **Config Server**.

- Product Service

```
docker run --name product-service \
    --link config-server \
    --link zipkin \
    -d -e SPRING_CLOUD_CONFIG_URI=http://config-server:8888 \
    -e SPRING_ZIPKIN_BASE-URL=http://zipkin:9411/ \
    -p 9990:9990 nebrass/product-service:0.0.1-SNAPSHOT
```

- Order Service

```
docker run --name order-service \
    --link config-server \
    --link zipkin \
    -d -e SPRING_CLOUD_CONFIG_URI=http://config-server:8888 \
    -e SPRING_ZIPKIN_BASE-URL=http://zipkin:9411/ \
    -p 9991:9991 nebrass/order-service:0.0.1-SNAPSHOT
```

- Customer Service

```
docker run --name customer-service \
    --link config-server \
    --link zipkin \
    -d -e SPRING_CLOUD_CONFIG_URI=http://config-server:8888 \
    -e SPRING_ZIPKIN_BASE-URL=http://zipkin:9411/ \
    -p 9992:9992 nebrass/customer-service:0.0.1-SNAPSHOT
```

But..

- What if one of these containers fail ?
- What if a monitor crashed or a container got unavailable ?

Things become complicated (managing the servers, managing the container state, ..).

Here comes, in the next chapter, the orchestration system, which provides many features like orchestrating computing, networking, storage infrastructure…

Chapter 12. Getting started with Kubernetes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What is Kubernetes ?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Kubernetes Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Kubernetes Core Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Kubectl

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Cluster

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Namespace

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Label

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Annotation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Selector

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Pod

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

ReplicationController

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

ReplicaSet

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

StatefulSet

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

DaemonSet

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Service

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Ingress

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Volume

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

PersistentVolume

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

PersistentVolumeClaim

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

StorageClass

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Job

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

CronJob

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

ConfigMap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Secret

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Run Kubernetes locally

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 13: The Kubernetes style

Discovering the Kubernetes style

Create the ConfigMap

Create the Secret

Deploy PostgreSQL to Kubernetes

What is Spring Cloud Kubernetes

Deploy it to Kubernetes

It works ! Hakuna Matata !

Revisiting our Cloud Patterns after meeting Kubernetes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Service Discovery and Registration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Engaging the Kubernetes-enabled Discovery library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Remove the Eureka Client dependency

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Add the Kubernetes-enabled Discovery library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Defining the Kubernetes Service name:

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Load Balancing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Server Side Load Balancing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Client Side Load Balancing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Externalized Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Replacing the Config Server by ConfigMaps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Removing the Spring Cloud Config footprints

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Adding the Maven Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 3: Creating ConfigMaps based on the Application properties files

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 4: Authorizing the ServiceAccount access to ConfigMaps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 5: Boosting Spring Cloud Kubernetes Config

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Log aggregation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Prepare the Minikube

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Install EFK using Helm

Step 2.1: Prepare Helm

Step 2.2: Add the Chart repository

Step 3: Installing the Elasticsearch Operator

Step 4: Installing the EFK Stack using Helm

Step 5: Remove the broadcasting appenders

Health check API

API Gateway

Step 1: Delete the old ApiGateway microservice

Step 2: Create the ApiGateway Ingress

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Distributed Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 1: Deploy Zipkin to Kubernetes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Step 2: Forward Sleth traces to Zipkin

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 14: Getting started with OpenShift

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What is really OpenShift ?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Run OpenShift locally

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Free OpenShift cluster

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What is the difference between OpenShift & Kubernetes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

The OpenShift Web Console

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Integrated Container Registry & ImageStreams

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Native CI/CD factory

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Logging and monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Version Control Integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Security

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What is changing in OpenShift?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Common Kubernetes and OpenShift resources

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

OpenShift specific resources

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 15: The Openshift style

Introduction

In the previous chapter, we crossed over the characteristics of OpenShift and we already said that OpenShift is mainly based on Kubernetes. In this chapter, we will discover what is changing when developping microservices for OpenShift instead of Kubernetes ?

What is the OpenShift style?

OpenShift is a Kubernetes-like platform. As we said in the previous chapter, it shares many common ressources with Kubernetes.

Route instead of Ingress

The `Ingress` object, in Kubernetes, is designed to signal the Kubernetes platform that a `Service` needs to be accessible to the outside world and it contains the configuration needed such as an externally-reachable URL, SSL, and more.

For OpenShift, Red Hat saw the need for enabling external access to a `Service` ***before the introduction of ingress objects in Kubernetes***, and created a concept called `Route` for the same purpose, with additional capabilities such as splitting traffic between multiple backends, sticky sessions, etc..

When a Route object is created on OpenShift, it gets picked up by the built-in HAProxy load balancer in order to expose the requested service and make it externally available with the given configuration. It's worth mentioning that although OpenShift provides this HAProxy-based built-in load-balancer, it has a pluggable architecture that allows admins to replace it with NGINX (and NGINX Plus) or external load-balancers like F5 BIG-IP.

You might be thinking at this point that this is confusing! Should one use the standard Kubernetes ingress objects or rather the OpenShift routes? What if you already have applications that are using the route objects? What if you are deploying an application written with Kubernetes ingress in mind?

Worry not! Since the release of Red Hat OpenShift 3.10, ingress objects are supported alongside route objects. The Red Hat OpenShift ingress controller implementation is designed to watch ingress objects and create one or more routes to fulfill the conditions specified. If you change the ingress object, the Red Hat OpenShift Ingress Controller syncs the changes and applies to the generated route objects. These are then picked up by the built-in HAProxy load balancer.

Ingress and Route while similar, have differences in maturity and capabilities. If you intend to deploy your application on multiple Kubernetes distributions at the same time then Ingress

might be a good option, although you will still have to deal with the specific ways that each Kubernetes distribution is providing the more complex routing capabilities such HTTPs and TLS re-encryption. For all other applications, you may run into scenarios where Ingress does not provide the functionality you need. The following table summarizes the differences between Kubernetes Ingress and Red Hat OpenShift Routes.

Building applications

In the Kubernetes world, we deploy some already-built container, that is hosted in a cluster reachable Containers Registry. We already did this. To deploy our Spring Boot/Cloud application to Kubernetes we:

- Built the WAR packaging the Spring Boot/Cloud application
- Built the Docker image holding the WAR package
- Pushed the Docker image to the Docker Hub Registry
- Created the Kubernetes Deployment object that pulled the Docker image and created the Docker Container on the cluster

In OpenShift, Red Hat thought on a new build mecanism called **Source-to-Image (S2I)**.

Source-to-Image (S2I) is a tool for building reproducible, Docker-formatted container images. It produces ready-to-run images by injecting application source into a container image and assembling a new image. The new image incorporates the base image (the builder) and built source and is ready to use with the `docker run` command. S2I supports incremental builds, which re-use previously downloaded dependencies, previously built artifacts, etc.

The advantages of **S2I** include the following:

- Image flexibility: S2I scripts can be written to inject application code into almost any existing Docker-formatted container image, taking advantage of the existing ecosystem. Note that, currently, S2I relies on tar to inject application source, so the image needs to be able to process tarred content.
- Speed: With S2I, the assemble process can perform a large number of complex operations without creating a new layer at each step, resulting in a fast process. In addition, S2I scripts can be written to re-use artifacts stored in a previous version of the application image, rather than having to download or build them each time the build is run.
- Patchability: S2I allows you to rebuild the application consistently if an underlying image needs a patch due to a security issue.
- Operational efficiency: By restricting build operations instead of allowing arbitrary actions, as a Dockerfile would allow, the PaaS operator can avoid accidental or intentional abuses of the build system.
- Operational security: Building an arbitrary Dockerfile exposes the host system to root privilege escalation. This can be exploited by a malicious user because the entire Docker build process is run as a user with Docker privileges. S2I restricts the operations performed as a root user and can run the scripts as a non-root user.
- User efficiency: S2I prevents developers from performing arbitrary yum

install type operations, which could slow down development iteration, during their application build.

- Ecosystem: S2I encourages a shared ecosystem of images where you can leverage best practices for your applications.
- Reproducibility: Produced images can include all inputs including specific versions of build tools and dependencies. This ensures that the image can be reproduced precisely.

Continuous Integration & Deployment

OpenShift Pipelines

OpenShift Pipelines give you control over building, deploying, and promoting your applications on OpenShift. Using a combination of the Jenkins Pipeline Build Strategy, Jenkinsfiles, and the OpenShift Domain Specific Language (DSL) (provided by the OpenShift Jenkins Client Plug-in), you can create advanced build, test, deploy, and promote pipelines for any scenario.

OpenShift Jenkins Client Plug-in

The OpenShift Jenkins Client Plug-in must be installed on your Jenkins master so the OpenShift DSL will be available to use within the JenkinsFile for your application. This plug-in is installed and enabled by default when using the OpenShift Jenkins image.

For more information about installing and configuring this plug-in, see Configuring Pipeline Execution.

OpenShift DSL

The OpenShift Jenkins Client Plug-in provides a fluent-styled DSL for communicating with the OpenShift API from within the Jenkins slaves. The OpenShift DSL is based on Groovy syntax and provides methods for controlling the lifecycle of your application such as create, build, deploy, and delete.

The full details of the API are embedded within the plug-in's online documentation within a running Jenkins instance. To find it:

- Create a new Pipeline Item.
- Click Pipeline Syntax below the DSL text area.
- From the left navigation menu, click Global Variables Reference.

Jenkins Pipeline Strategy

In order to take advantage of the OpenShift Pipelines within your project, you will must use the Jenkins Pipeline Build Strategy. This strategy defaults to using a jenkinsfile at the root of your source repository, but also provides the following configuration options:

- An inline jenkinsfile field within your BuildConfig.

- A jenkinsfilePath field within your BuildConfig that references the location of the jenkinsfile to use relative to the source contextDir.

The optional jenkinsfilePath field specifies the name of the file to use, relative to the source contextDir. If contextDir is omitted, it defaults to the root of the repository. If jenkinsfilePath is omitted, it defaults to jenkinsfile.

Jenkinsfile

The jenkinsfile utilizes the standard groovy language syntax to allow fine grained control over the configuration, build, and deployment of your application.

The `jenkinsfile` can be supplied in one of the following ways:

- A file located within your source code repository.
- Embedded as part of your build configuration using the `jenkinsfile` field.

When using the first option, the `jenkinsfile` must be included in your applications source code repository at one of the following locations:

- A file named `jenkinsfile` at the root of your repository.
- A file named `jenkinsfile` at the root of the source `contextDir` of your repository.
- A file name specified via the `jenkinsfilePath` field of the `JenkinsPiplineStrategy` section of your `BuildConfig`, which is relative to the source `contextDir` if supplied, otherwise it defaults to the root of the repository.

The `jenkinsfile` is executed on the Jenkins slave pod, which must have the OpenShift Client binaries available if you intend to use the OpenShift DSL.

Jenkins

OpenShift Container Platform provides a container image for running Jenkins. This image provides a Jenkins server instance, which can be used to set up a basic flow for continuous testing, integration, and delivery.

This image also includes a sample Jenkins job, which triggers a new build of a `BuildConfig` defined in OpenShift Container Platform, tests the output of that build, and then on successful build, retags the output to indicate the build is ready for production.

Using Jenkins as a Source-To-Image builder
To customize the official OpenShift Container Platform Jenkins image, you have two options:

- Use Docker layering.
- Use the image as a Source-To-Image builder, described here.

You can use S2I to copy your custom Jenkins Jobs definitions, additional plug-ins or replace the provided config.xml file with your own, custom, configuration.

Using the Jenkins Kubernetes Plug-in to Run Jobs

The official OpenShift Container Platform Jenkins image includes the pre-installed Kubernetes plug-in that allows Jenkins slaves to be dynamically provisioned on multiple container hosts using Kubernetes and OpenShift Container Platform.

To use the Kubernetes plug-in, OpenShift Container Platform provides three images suitable for use as Jenkins slaves: the **Base**, **Maven**, and **Node.js** images.

The first is a base image for Jenkins slaves:

- It pulls in both the required tools (headless Java, the Jenkins JNLP client) and the useful ones (including git, tar, zip, and nss among others).
- It establishes the JNLP slave agent as the entrypoint.
- It includes the oc client tooling for invoking command line operations from within Jenkins jobs, and
- It provides Dockerfiles for both CentOS and RHEL images.

Two additional images that extend the base image are also provided:

- Maven
- Node.js

Both the Maven and Node.js slave images are configured as Kubernetes Pod Template images within the OpenShift Container Platform Jenkins image's configuration for the Kubernetes plugin. That configuration includes labels for each of the images that can be applied to any of your Jenkins jobs under their "Restrict where this project can be run" setting. If the label is applied, execution of the given job will be done under an OpenShift Container Platform pod running the respective slave image.

The Maven and Node.js Jenkins slave images provide Dockerfiles for both CentOS and RHEL that you can reference when building new slave images. Also note the `contrib` and `contrib/bin` subdirectories. They allow for the insertion of configuration files and executable scripts for your image.

The Jenkins image also provides auto-discovery and auto-configuration of slave images for the Kubernetes plug-in. The Jenkins image searches for these in the existing image streams within the project that it is running in. The search specifically looks for image streams that have the label `role` set to `jenkins-slave`.

When it finds an image stream with this label, it generates the corresponding Kubernetes plug-in configuration so you can assign your Jenkins jobs to run in a pod running the container image provided by the image stream.

Note: this scanning is only performed once, when the Jenkins master is starting. If you label additional imagestreams, the Jenkins master will need to be restarted to pick up the additional images.

OpenShift Container Platform Pipeline Plug-in

The Jenkins image's list of pre-installed plug-ins includes a plug-in which assists in the creating of CI/CD workflows that run against an OpenShift Container Platform server. A series of build steps, post-build actions, as well as SCM-style polling are provided which equate to administrative and operational actions on the OpenShift Container Platform server and the API artifacts hosted there.

In addition to being accessible from the classic "freestyle" form of Jenkins job, the build steps as of version 1.0.14 of the OpenShift Container Platform Pipeline Plug-in are also available to Jenkins Pipeline jobs via the DSL extension points provided by the Jenkins Pipeline Plug-in. The OpenShift Jenkins Pipeline build strategy sample illustrates how to use the OpenShift Pipeline plugin DSL versions of its steps.

The sample Jenkins job that is pre-configured in the Jenkins image utilizes the OpenShift Container Platform pipeline plug-in and serves as an example of how to leverage the plug-in for creating CI/CD flows for OpenShift Container Platform in Jenkins.

OpenShift Container Platform Client Plug-in

The experiences gained working with users of the OpenShift Pipeline plug-in, coupled with the rapid evolution of both Jenkins and OpenShift, have provided valuable insight into how to integrate OpenShift Container Platform from Jenkins jobs.

As such, the new experimental OpenShift Client Plug-in for Jenkins is now offered as a technical preview and is included in the OpenShift Jenkins images on CentOS (**docker.io/openshift/jenkins-1-centos7:latest** and **docker.io/openshift/jenkins-2-centos7:latest**). The plug-in is also available from the Jenkins Update Center. The OpenShift Client plug-in will eventually replace the OpenShift Pipeline plug-in as the tool for OpenShift integration from Jenkins jobs. The OpenShift Client Plug-in provides:

- A Fluent-style syntax for use in Jenkins Pipelines
- Use of and exposure to any option available with oc
- Integration with Jenkins credentials and clusters
- Continued support for classic Jenkins Freestyle jobs

OpenShift Container Platform Sync Plug-in

To facilitate OpenShift Container Platform Pipeline build strategy for integration between Jenkins and OpenShift Container Platform, the OpenShift Sync plug-in monitors the API server of OpenShift Container Platform for updates `BuildConfigs` and `Builds` that employ the Pipeline strategy and either creates Jenkins Pipeline projects (when a `BuildConfig` is created) or starts jobs in the resulting projects (when a `Build` is started).

Conclusion

Does it worth to move from OpenShift ?

I got this question from many colleagues, customers and even from myself.. actually I think that Kubernetes can do all the needed job of 95% of projects.. For sure there are some cases that the need for OpenShift is justified.

When do I need OpenShift ?

In few words, it's better to choose OpenShift if:

- You need Technical/Commercial support from Red Hat
- You need more efficient Security layer that doesn't exist in Kubernetes
- You need an easier version of Kubernetes
- You are already in a Red Hat compliant environment (RHEL or CentOs servers, OpenStack, ...)

Is OpenShift really matters ?

I think OpenShift is a great product, but the 3.x editions have many problems and difficulties in matter of installation and upgrade. I believe that the budget of these operations is extremely huge, which is a serious point to take into consideration when choosing the solution.

After the acquisition of CoreOS by Red Hat, OpenShift will be adopting some great features that where available in CoreOS Tectonic, such as a one-click update mecanism. Which will be a good feature in the 4.x editions.

Can I do the job using Kubernetes ?

Yeah ! For sure ! Kuberentes is a Swiss-Army-knife :) You can take profit of all the greate features that we need from a container orchestrator. It's true that some needs will be so hard to enable and configure, such as security concepts of the cluster. But, it still a reachable target for the solution.

Personally, I recommend using raw Kubernetes and take profit of all its features. OpenShift stills a good alternative, but you have to be carefull of all the heavy tasks and requirements of OpenShift.

Part four: Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Chapter 16: Conclusion

Development

Fabric8 Maven Plugin

Deployment

End of the End

Part five: Bonus Chapters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Service Mesh

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

What is a Service Mesh ?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Why do we need Service Mesh ?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.

Bonus 1: Service Mesh: Istio

What is Istio?

Google presents **Istio** as an open platform to connect, monitor, and secure microservices.

Istio is a service mesh implementation that provides many cloud-native capabilities like:

- **Traffic management**: Service Discovery, Load balancing, Failure recovery, A/B testing, Canary releases, etc..
- **Observability**: Request Tracing, Metrics, Monitoring, Auditing, Logging, etc..
- **Security**: ACLs, Access control, Rate limiting, End-to-end authentication, etc..

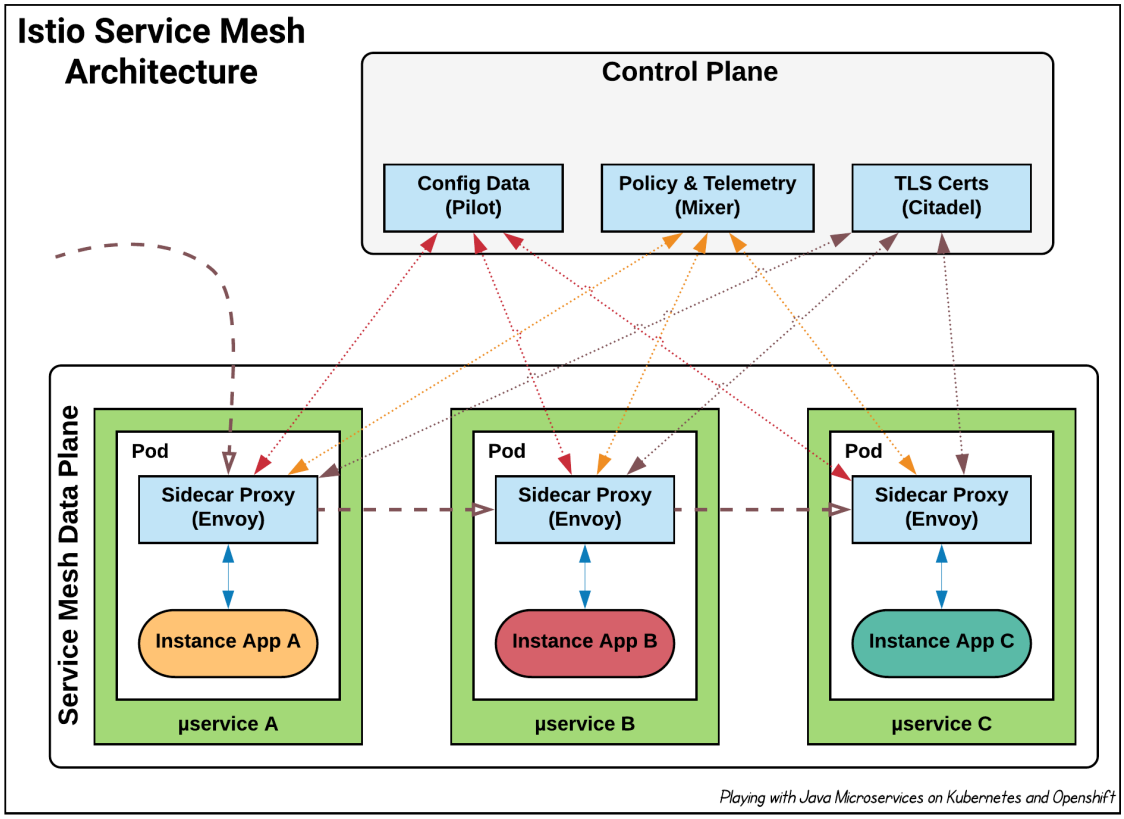
Istio delivers all these great features without any changes to the code of the microservices running with it on the same *Kubernetes* cluster.

In our case, we already implemented many of these features and capabilities when we were writing our microservices. If we had **Istio** at the beginning, we could save so much effort and time, by delegating all of these capabilities to **Istio**.

Istio Architecture

Istio service mesh is composed of two parts:

- The **data plane** is responsible for establishing, securing, and controlling the traffic through the Service Mesh. The management components that instruct the data plane how to behave is known as the "**control plane**".
- The **control plane** is the brains of the mesh and exposes an API for operators to manipulate the network behaviors.



Istio Service Mesh Architecture

Istio Components

From the *Istio Architecture* diagram, we can see different components, located in different areas of the ecosystem:

Envoy

Sidecar proxies per microservice to handle ingress/egress traffic between services in the cluster and from a service to external services. The proxies form a *secure microservice mesh* providing a rich set of functions like discovery, rich layer-7 routing, circuit breakers, policy enforcement and telemetry recording/reporting functions.

The service mesh is not an overlay network. It simplifies and enhances how microservices in an application talk to each other over the network provided by the underlying platform.

Envoy is deployed as a sidecar to the relevant microservice in the same Kubernetes pod. This deployment allows Istio to extract a wealth of signals about traffic behavior as attributes. Istio can, in turn, use these attributes in Mixer to enforce policy decisions, and send them to monitoring systems to provide information about the behavior of the entire mesh.

Mixer

Mixer is a central component that is leveraged by the proxies and microservices to enforce policies such as authorization, rate limits, quotas, authentication, request tracing and telemetry collection.

Mixer includes a flexible plugin model. This model enables Istio to interface with a variety of host environments and infrastructure backends. Thus, Istio abstracts the Envoy proxy and Istio-managed services from these details.

Pilot

Pilot provides service discovery for the Envoy sidecars, traffic management capabilities for intelligent routing (e.g., A/B tests, canary deployments, etc.), and resiliency (timeouts, retries, circuit breakers, etc.).

Pilot converts high level routing rules that control traffic behavior into Envoy-specific configurations, and propagates them to the sidecars at runtime. Pilot abstracts platform-specific service discovery mechanisms and synthesizes them into a standard format that any sidecar conforming with the Envoy data plane APIs can consume. This loose coupling allows Istio to run on multiple environments such as Kubernetes, Consul, or Nomad, while maintaining the same operator interface for traffic management.

Citadel

Citadel provides strong service-to-service and end-user authentication with built-in identity and credential management. You can use Citadel to upgrade unencrypted traffic in the service mesh. Using Citadel, operators can enforce policies based on service identity rather than on network controls. Starting from release 0.5, you can use Istio's authorization feature to control who can access your services.

Galley

Galley validates user authored Istio API configuration on behalf of the other Istio control plane components. Over time, Galley will take over responsibility as the top-level configuration ingestion, processing and distribution component of Istio. It will be responsible for insulating the rest of the Istio components from the details of obtaining user configuration from the underlying platform (e.g. Kubernetes).

Getting started with Istio

Requirements

We will be playing with **Istio** on *Kubernetes*. For testing the solution, you need to have, as usual, a running **Minikube** on your machine.

The example of this chapter will be running on **Minikube** *v0.35.0* with a custom config: *4 CPUs with 8Go of Memory*.

Get & Install Istio

To start downloading and installing Istio, just enter the following command:

```
curl -L https://git.io/getLatestIstio | sh -
```

By the end of the command execution, you will see some message like this one:

```
Add /Users/n.lamouchi/istio-1.0.6/bin to your path; e.g copy paste in your shell and/or ~/.profile:

export PATH="$PATH:/Users/n.lamouchi/istio-1.0.6/bin"
```

This command will add the Istio binaries path to the local variable PATH (as in the console).

Next, we will move to Istio package directory:

```
cd istio-1.0.6/
```

As the first step, you have to install Istio's Custom Resources Definition:

```
kubectl apply -f install/kubernetes/helm/istio/templates/crds.yaml
```

What is Custom Resources Definition ?

Custom resources definition (CRD) is a powerful feature introduced in *Kubernetes* 1.7 which enables users to add their own/custom objects to the *Kubernetes* cluster and use it like any other native *Kubernetes* objects.

Next, we need to install Istio's core components. We have four different options to do this:

- **Option 1**: Install Istio WITHOUT mutual TLS authentication between sidecars
- **Option 2**: Install Istio WITH default mutual TLS authentication
- **Option 3**: Render Kubernetes manifest with Helm and deploy with kubectl
- **Option 4**: Use Helm and Tiller to manage the Istio deployment

For a production setup of Istio, it's recommended to install with the Helm Chart (Option 4), to use all the configuration options. This permits customization of Istio to operator specific requirements.

In this tutorial, we will be using the **Option 1**:

To install Istio:

```
kubectl apply -f install/kubernetes/istio-demo.yaml
```

Verifying the installation

To be sure that the Istio components were correctly installed, these Kubernetes Services needs to be installed: `istio-pilot`, `istio-ingressgateway`, `istio-policy`, `istio-telemetry`, `prometheus`, `istio-galley`, and (optionally) `istio-sidecar-injector`:

```
kubectl get svc -n istio-system
```

```
NAME                     TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)              AGE
grafana                  ClusterIP     10.102.186.225  <none>        3000/TCP             35m
istio-citadel            ClusterIP     10.108.239.218  <none>        8060/TCP,9093/TCP    58m
istio-egressgateway      ClusterIP     10.103.151.29   <none>        80/TCP,443/TCP       58m
istio-galley             ClusterIP     10.96.55.14     <none>        443/TCP,9093/TCP     58m
istio-ingressgateway     LoadBalancer  10.110.91.248   <pending>     80:31380/TCP..       58m
istio-pilot              ClusterIP     10.104.95.143   <none>        15010/TCP..          58m
istio-policy             ClusterIP     10.100.19.140   <none>        9091/TCP..           58m
istio-sidecar-injector   ClusterIP     10.101.13.203   <none>        443/TCP              58m
istio-telemetry          ClusterIP     10.103.135.98   <none>        9091/TCP..           58m
jaeger-agent             ClusterIP     None            <none>        5775/UDP..           35m
jaeger-collector         ClusterIP     10.110.82.2     <none>        14267/TCP,14268/TCP  35m
jaeger-query             ClusterIP     10.101.54.162   <none>        16686/TCP            35m
prometheus               ClusterIP     10.101.210.170  <none>        9090/TCP             58m
servicegraph             ClusterIP     10.99.60.12     <none>        8088/TCP             35m
tracing                  ClusterIP     10.98.62.125    <none>        80/TCP               35m
zipkin                   ClusterIP     10.100.54.120   <none>        9411/TCP             35m
```

For the Kubernetes *Services* already listed, we will find corresponding *Pods*:

```
kubectl get svc -n istio-system
```

\

```
NAME                                      READY   STATUS      RESTARTS   AGE
grafana-59b8896965-n4rvr                  1/1     Running     0          91m
istio-citadel-6f444d9999-xc27q            1/1     Running     0          91m
istio-cleanup-secrets-k2dzg               0/1     Completed   0          91m
istio-egressgateway-6d79447874-8twkk      1/1     Running     0          91m
istio-galley-685bb48846-tht5x             1/1     Running     0          91m
istio-grafana-post-install-rmflr          0/1     Completed   0          91m
istio-ingressgateway-5b64fffc9f-56tm2     1/1     Running     0          91m
istio-pilot-7f558fc848-2fscl              2/2     Running     0          91m
istio-policy-547d64b8d7-skpzm             2/2     Running     0          91m
istio-security-post-install-nlfmd         0/1     Completed   0          91m
istio-sidecar-injector-5d8dd9448d-rdbq8   1/1     Running     0          91m
istio-telemetry-c5488fc49-b8sv8           2/2     Running     0          91m
istio-tracing-6b994895fd-6wxvx            1/1     Running     0          91m
prometheus-76b7745b64-2tgkw               1/1     Running     0          91m
servicegraph-cb9b94c-2x5cb                1/1     Running     1          91m
```

All *Pods* need to be in the **Running** status, except the `istio-cleanup-secrets-*`, `istio-grafana-post-install-*` and `istio-security-post-install-*` *Pods*, which will be in the **Completed** status. These three **Completed** *Pods* are started and executed at a post-installation phase, to do post-installation tasks like cleaning the installation secrets, etc..

Envoy Sidecar Injection

In the service mesh world, a sidecar is a utility container in the pod, and its purpose is to support the main container. In the Istio case, the sidecar will be an Envoy proxy that will be deployed to each pod. The process of adding Envoy into a pod is called **Sidecar Injection**. This action can be done in two ways:

* Automatically using the Istio Sidecar Injector : Automatic injection injects at pod creation time. The controller resource is unmodified. Sidecars can be updated selectively by manually deleting a pods or systematically with a deployment rolling update.
* Manually using the **Istioctl-CLI** tool: Manual injection modifies the controller configuration, e.g. deployment. It does this by modifying the pod template spec such that all pods for that deployment are created with the injected sidecar. Adding, Updating or Removing the sidecar requires modifying the entire deployment.

Automatic Sidecar Injection

To enable the **Automatic Sidecar Inject** just add the `istio-injection` label to the Kubernetes namespace: For example to enable it in the *default* namespace:

```
kubectl label namespace default istio-injection=enabled --overwrite
```

Now, when a pod will be created, the Envoy sidecar is automatically injected inside it.

Manual Sidecar Injection

Inject the sidecar into the deployment using the in-cluster configuration. <1>

```
istioctl kube-inject -f samples/bookinfo/platform/kube/bookinfo.yaml \
           --output bookinfo-injected.yaml

kubectl apply -f bookinfo-injected.yaml
```

BookInfo Sample Application

This example deploys a sample application composed of four separate microservices used to demonstrate various Istio features. The application displays information about a book, similar to a single catalog entry of an online book store. Displayed on the page is a description of the book, book details (ISBN, number of pages, and so on), and a few book reviews.

The Bookinfo application is broken into four separate microservices:

- productpage. The productpage microservice calls the details and reviews microservices to populate the page.
- details. The details microservice contains book information.
- reviews. The reviews microservice contains book reviews. It also calls the ratings microservice.
- ratings. The ratings microservice contains book ranking information that accompanies a book review.

There are 3 versions of the reviews microservice:

- Version v1 doesn't call the ratings service.
- Version v2 calls the ratings service, and displays each rating as 1 to 5 black stars.
- Version v3 calls the ratings service, and displays each rating as 1 to 5 red stars.

These two commands can be done in one command:

```
kubectl create -f <(istioctl kube-inject -f samples/bookinfo/platform/kube/bookinfo.yaml)
```

or also:

```
istioctl kube-inject -f samples/bookinfo/platform/kube/bookinfo.yaml | kubectl apply -f -
```

These commands will inject the Istio Envoy sidecar into the Kubernetes *Deployment* object.

For the sample *Deployment* of the `details-v1` microservice which looks like this:

Sample Deployment before injecting Istio

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: details-v1
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: details
        version: v1
    spec:
      containers:
```

```
    - name: details
      image: istio/examples-bookinfo-details-v1:1.8.0
      imagePullPolicy: IfNotPresent
      ports:
      - containerPort: 9080
```

The *Deployment* after the injection of Istio will look like:

Deployment with the injected Istio

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 creationTimestamp: null
  name: details-v1
spec:
  replicas: 1
 strategy: {}
  template:
    metadata:
     annotations:
       sidecar.istio.io/status: '...'
     creationTimestamp: null
      labels:
        app: details
        version: v1
    spec:
      containers:
      - name: details
        image: istio/examples-bookinfo-details-v1:1.8.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
       resources: {}
     - args:
       - proxy
       - sidecar
       - --configPath
       - /etc/istio/proxy
       - --binaryPath
       - /usr/local/bin/envoy
       - --serviceCluster
       - details
       - --drainDuration
       - 45s
       - --parentShutdownDuration
       - 1m0s
       - --discoveryAddress
       - istio-pilot.istio-system:15007
       - --discoveryRefreshDelay
       - 1s
       - --zipkinAddress
       - zipkin.istio-system:9411
       - --connectTimeout
       - 10s
       - --proxyAdminPort
       - "15000"
```

```
    - --controlPlaneAuthPolicy
    - NONE
    env:
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: INSTANCE_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: ISTIO_META_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: ISTIO_META_INTERCEPTION_MODE
      value: REDIRECT
    - name: ISTIO_METAJSON_LABELS
      value: |
        {"app":"details","version":"v1"}
    image: docker.io/istio/proxyv2:1.0.6
    imagePullPolicy: IfNotPresent
    name: istio-proxy
    ports:
    - containerPort: 15090
      name: http-envoy-prom
      protocol: TCP
    resources:
      requests:
        cpu: 10m
    securityContext:
      readOnlyRootFilesystem: true
      runAsUser: 1337
    volumeMounts:
    - mountPath: /etc/istio/proxy
      name: istio-envoy
    - mountPath: /etc/certs/
      name: istio-certs
      readOnly: true
  initContainers:
  - args:
    - -p
    - "15001"
    - -u
    - "1337"
    - -m
    - REDIRECT
    - -i
    - '*'
    - -x
    - ""
    - -b
    - "9080"
```

```
        - -d
        - ""
        image: docker.io/istio/proxy_init:1.0.6
        imagePullPolicy: IfNotPresent
        name: istio-init
        resources: {}
        securityContext:
          capabilities:
            add:
            - NET_ADMIN
          privileged: true
      volumes:
      - emptyDir:
          medium: Memory
        name: istio-envoy
      - name: istio-certs
        secret:
          optional: true
          secretName: istio.default
status: {}
```

All the extra paramters and configuration are added via `istioctl kube-inject` command.

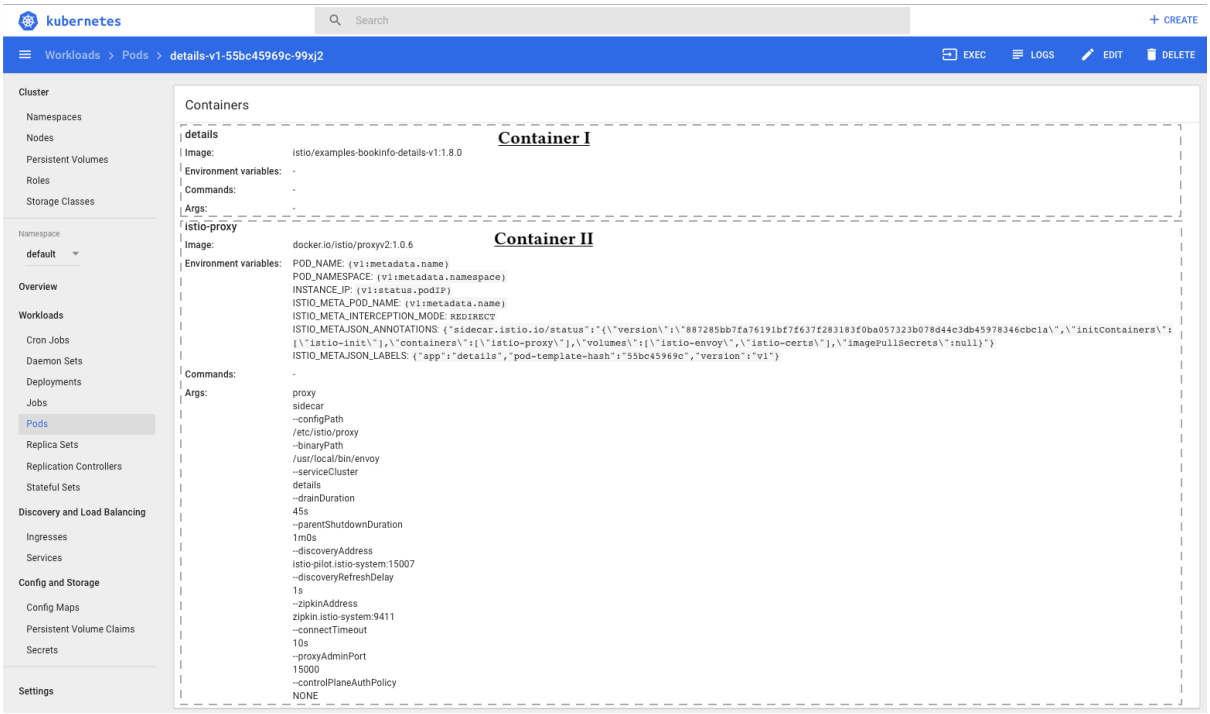
After executing the command:

```
kubectl create -f <(istioctl kube-inject -f samples/bookinfo/platform/kube/bookinfo.yaml)

service/details created
deployment.extensions/details-v1 created
service/ratings created
deployment.extensions/ratings-v1 created
service/reviews created
deployment.extensions/reviews-v1 created
deployment.extensions/reviews-v2 created
deployment.extensions/reviews-v3 created
service/productpage created
deployment.extensions/productpage-v1 created
```

We can verify that the **sidecar** is deployed in the same *Deployment* as the microservice, just type:

```
kubectl get deployment details-v1 -o wide

NAME         READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS            IMAGES    SELECTOR
details-v1   1/1     1            1           63m   details,istio-proxy   ...       ...
```

We can even see that there are two containers in the `details-v1-*` pod:



Istio sidecar in the details-v1-* pod

To be sure that everything is ok, we need to verify that the `BookInfo` services & pods are here:

```
kubectl get svc
```

```
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
details       ClusterIP   10.106.82.61    <none>        9080/TCP   4h51m
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP    11h
productpage   ClusterIP   10.100.126.93   <none>        9080/TCP   4h51m
ratings       ClusterIP   10.98.201.254   <none>        9080/TCP   4h51m
reviews       ClusterIP   10.110.241.59   <none>        9080/TCP   4h51m
```

And :

```
kubectl get pod
```

```
NAME                              READY   STATUS    RESTARTS   AGE
details-v1-55bc45969c-99xj2       2/2     Running   0          4h52m
productpage-v1-5b597ff459-pfpmt   2/2     Running   0          4h52m
ratings-v1-7877895db5-vchw2       2/2     Running   0          4h52m
reviews-v1-699587b49b-bt7z5       2/2     Running   0          4h52m
reviews-v2-cc7cd59cc-k6l6j        2/2     Running   0          4h52m
reviews-v3-6fbcf56df8-nc2lf       2/2     Running   0          4h52m
```

Now, we can go on to next steps and enjoy the great Istio features :)

Traffic Management

Istio Gateway & VirtualService

Now that the Bookinfo services are up and running, we need to make the Services accessible from outside of your Kubernetes cluster. An **Istio Gateway** object is used for this purpose.

An **Istio Gateway** configures a load balancer for *HTTP/TCP* traffic at the edge of the service mesh and enables **Ingress** traffic for an application. Unlike Kubernetes **Ingress**, **Istio Gateway** only configures the *L4-L6* functions (for example, ports to expose, TLS configuration). Users can then use standard Istio rules to control *HTTP* requests as well as *TCP* traffic entering a **Gateway** by binding a **VirtualService** to it.

We can define the **Ingress gateway** for the *Bookinfo* application using the sample gateway configuration:

Sample BookInfo Gateway

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

A **VirtualService** defines the rules that control how requests for a service are routed within an Istio service mesh. For example, a **VirtualService** could route requests to different versions of a service or to a completely different service than was requested. Requests can be routed based on the request source and destination, *HTTP* paths and header fields, and weights associated with individual service versions.

The **VirtualService** configuration looks like:

Sample BookInfo VirtualService

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
  - "*"
  gateways:
  - bookinfo-gateway
  http:
  - match:
```

```
    - uri:
        exact: /productpage
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        prefix: /api/v1/products
    route:
    - destination:
        host: productpage
        port:
          number: 9080
```

Let's create the **Istio Gateway** and the **VirtualService**:

```
kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
```

Confirm the gateway has been created:

```
kubectl get gateway

NAME               AGE
bookinfo-gateway   59m
```

Let's export the `INGRESS_HOST`, the `INGRESS_PORT` and the `GATEWAY_URL`:

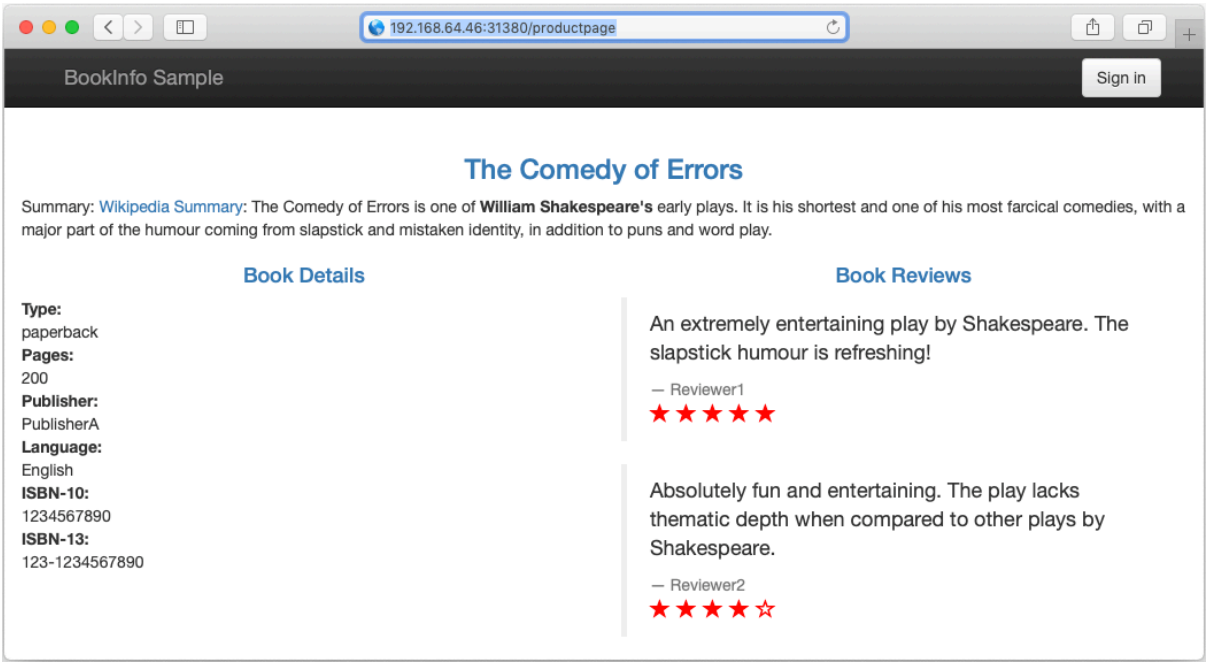
```
export INGRESS_HOST=$(minikube ip)

export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway \
                      -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')

export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
```

To test the Gateway:

```
curl -o /dev/null -s -w "%{http_code}\n" http://$GATEWAY_URL/productpage
```

You must have `200` as response code.

You can also point your browser to `http://$GATEWAY_URL/productpage` to view the Bookinfo web page. If you refresh the page several times, you should see different versions of reviews shown in `productpage`, presented in a round robin style (red stars, black stars, no stars), since we haven't yet used Istio to control the version routing.



BookInfo Sample productpage

If we refresh the web page some times, we will get different versions of the `reviews` structure: One version has red stars (the one that we got in the screenshot), an other one that have back stars and a third one without starts. This is due to the availability of three versions of the `reviews` microservice deployed in our sample `BookInfo` application.

You can verify the availability of the three versions of the `reviews` microservice:

```
kubectl get pods -l app=reviews

NAME                          READY   STATUS    RESTARTS   AGE
reviews-v1-699587b49b-v44wr   2/2     Running   0          5h31m
reviews-v2-cc7cd59cc-rgc86    2/2     Running   0          5h31m
reviews-v3-6fbcf56df8-wndtz   2/2     Running   0          5h31m
```

We got different versions of `reviews` structure while refreshing because Istio, by default, dispatches access to loadbalanced services using a **Round Robin** scheduling. One of the great features of Istio is to route the traffic to some dedicated version of a service, or even using sliced dispatching of requests.

In the next steps, we will see how to route the traffic based on version.

Destination Rules

Before we can use Istio to control the `Bookinfo` version routing, we need to define the available versions of an application, called `subsets`. These `subsets` are defined in an Istio object called **DestinationRule** /The choice of version to display can be decided based on criteria (headers, URL, etc..) defined to each version. We can enjoy this flexibility of criterias to do Blue-green Deployments, A/B Testing, and Canary Releases.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: productpage
spec:
  host: productpage <1>
  subsets:
  - name: v1 <2>
    labels:
      version: v1 <2>
```

1. The Kubernetes Service name on which we will be routing the traffic
2. The `subset` element name
3. The `subset` element version

We will create the default destination rules for the Bookinfo services, using the sample `destination-rule-all.yaml`:

samples/bookinfo/networking/destination-rule-all.yaml

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: productpage
spec:
  host: productpage
  subsets:
  - name: v1
    labels:
      version: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
    labels:
      version: v3
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ratings
spec:
  host: ratings
  subsets:
  - name: v1
```

```
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v2-mysql
    labels:
      version: v2-mysql
  - name: v2-mysql-vm
    labels:
      version: v2-mysql-vm
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: details
spec:
  host: details
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
---
```

Run the following command to create default Destination Rules:

```
kubectl apply -f samples/bookinfo/networking/destination-rule-all.yaml
```

Wait a few seconds for the destination rules to propagate.

You can verify that the destination rules are correctly created, using the following command:

```
kubectl get destinationrules -o yaml
```

Now, we will change the default round-robin behavior for traffic routing: we will route all traffic to `reviews-v1`, using a new **VirtualServices** configuration that looks like this:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
```

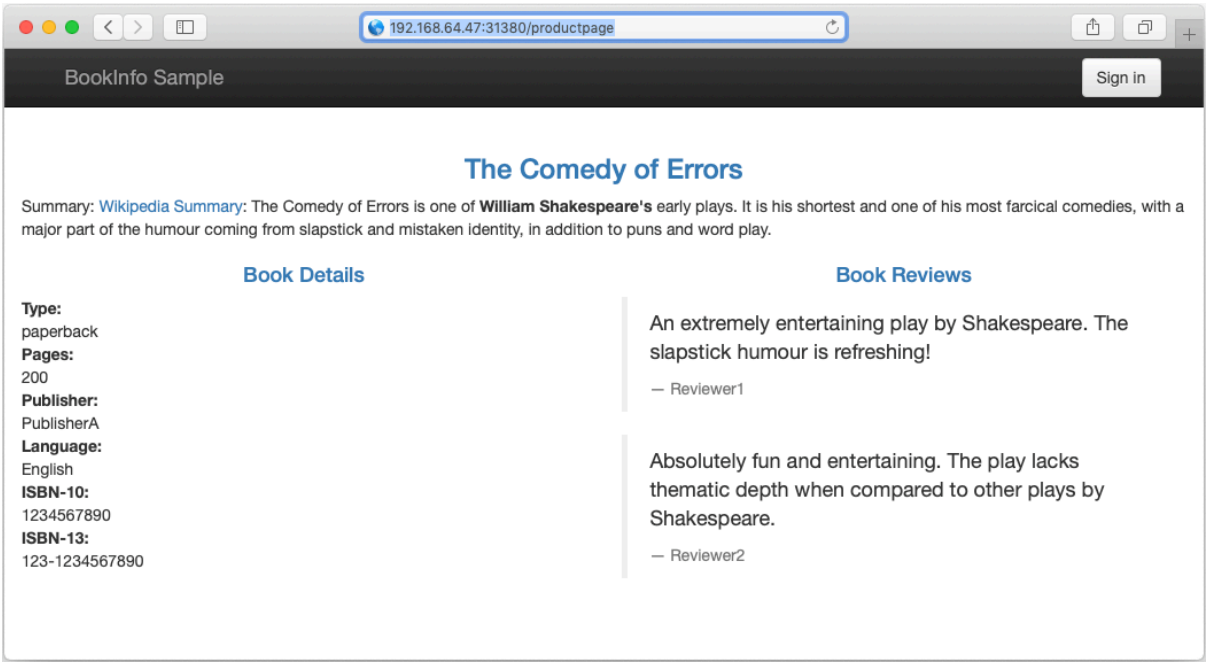
The `BookInfo` sample brings a `virtual-service-all-v1.yaml` that holds the necessary configuration of the new **VirtualServices**. To deploy it, just type:

```
kubectl apply -f samples/bookinfo/networking/virtual-service-all-v1.yaml
```

You can verify that the `review` **VirtualService** is correctly created, using the following command:

```
kubectl get virtualservices reviews -o yaml
```

Try now to reload the page multiple times, and note how only `reviews:v1` is displayed each time:



Product page with reviews-v1

Next, we will change the route configuration so that all traffic from a specific user is routed to a specific service version. In this case, all traffic from a user named Jason will be routed to the service `reviews:v2`.

Next, we will be routing traffic for a specific user to a specific service version. In our example, we will be routing all the traffic for a user named `jason` to the service `reviews:v2`.

The `virtual-service-reviews-test-v2.yaml` configuration covers this case:

samples/bookinfo/networking/virtual-service-reviews-test-v2.yaml

```
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: jason
    route:
    - destination:
        host: reviews
```

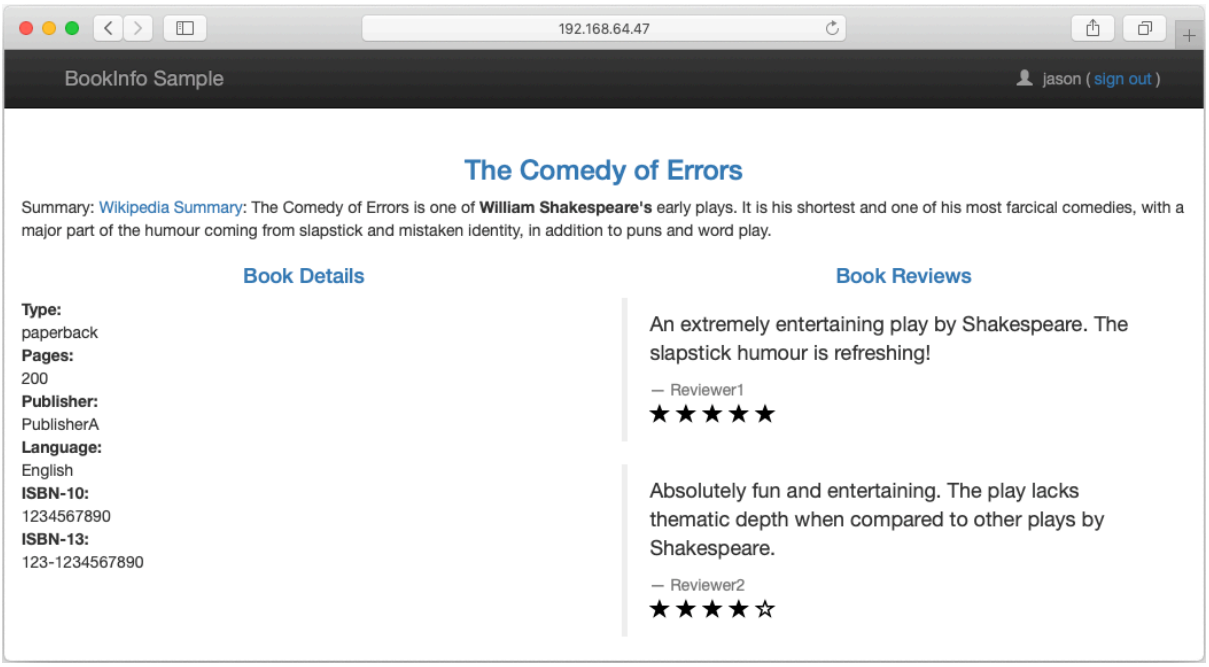
```
      subset: v2
  - route:
    - destination:
      host: reviews
      subset: v1
```

```
kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-test-v2.yaml
```

You can verify that the `review` **VirtualService** is correctly created, using the following command:

```
kubectl get virtualservices reviews -o yaml
```

To test the new **VirtualService**, just click on the *Sign* button and login using `jason` as username and any value as password. Now, we will see `reviews:v2`. If you logout, we will get back the `reviews:v1`:



Product page with reviews-v2 when user is jason

Next, we will see how to gradually migrate traffic from one version of a microservice to another one. In our example, we will send 50% of traffic to `reviews:v1` and 50% to `reviews:v3`:

```
kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-50-v3.yaml
```

Let's verify the content of the new **VirtualService**:

```
kubectl get virtualservice reviews -o yaml
```

The `subset` is set to **50%** of traffic to the `v1` and **50%** to the `v3`:

```
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 50
    - destination:
        host: reviews
        subset: v3
      weight: 50
```

Try now to reload the page multiple times, you will see diversely `reviews:v1` and `reviews:v3`.

Observability

Distributed Tracing

Hello Jaeger

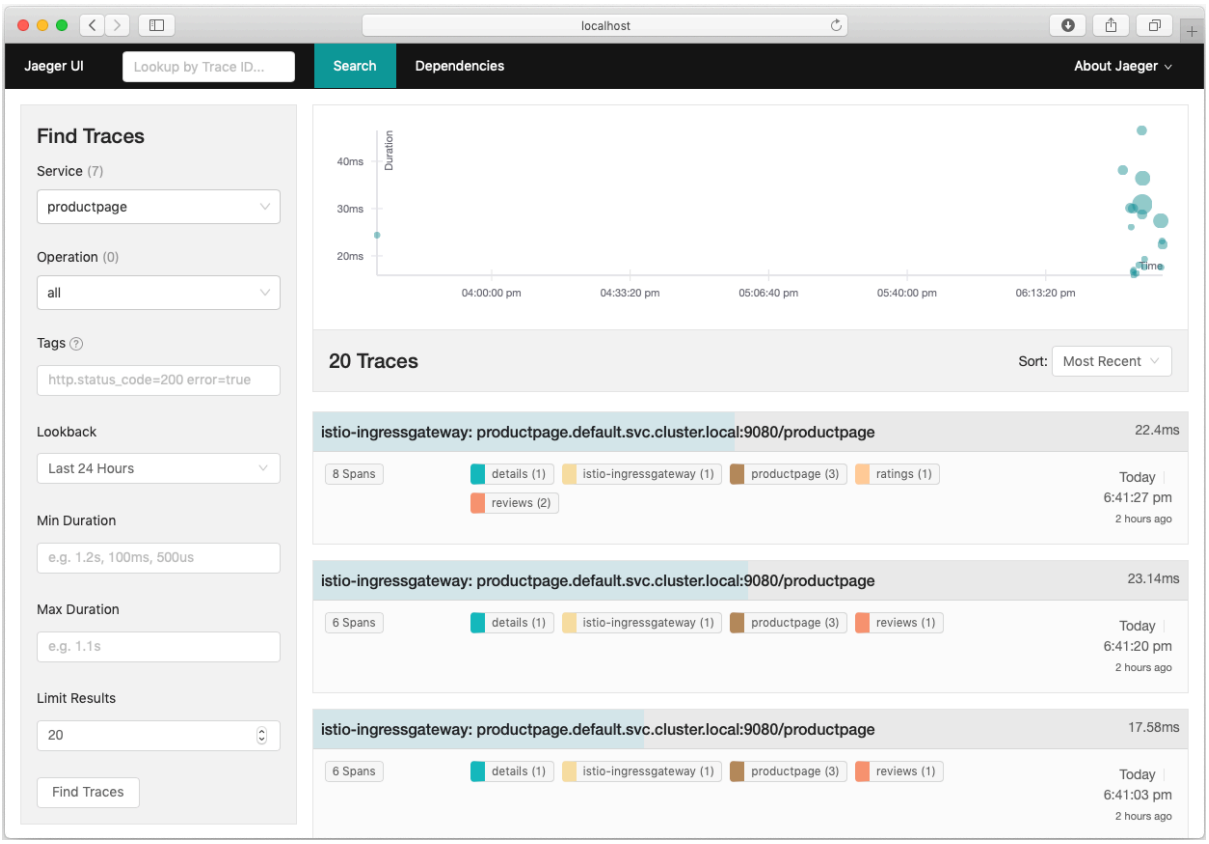
To access the **Jaeger Dashboard**, establish port forwarding from local port *16686* to the *Tracing* instance:

```
kubectl port-forward -n istio-system \
  $(kubectl get pod -n istio-system -l app=jaeger -o jsonpath='{.items[0].metadata.name}') \
  16686:16686
```

In your browser, go to http://127.0.0.1:16686[5]

From the *Services* menu, select **productpage** service.

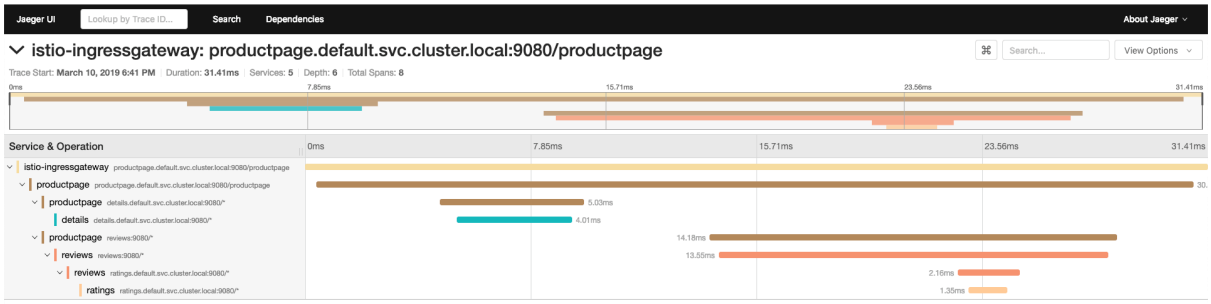
Scroll to the bottom and click on *Find Traces* button to see traces:



Jaeger Dashboard - Traces

[5] http://127.0.0.1:16686

If you click on a trace, you should see more details. The page should look something like this:



Jaeger Dashboard - Trace details

When invoking the `/productpage`, many BookInfo services are called. These services correspond to `spans` and the page call itself corresponds to the `trace`.

Although Istio proxies are able to automatically send spans, they need some hints to tie together the entire trace. Applications need to propagate the appropriate HTTP headers so that when the proxies send span information, the spans can be correlated correctly into a single trace.

To do this, an application needs to collect and propagate the following headers from the incoming request to any outgoing requests:

- x-request-id
- x-b3-traceid
- x-b3-spanid
- x-b3-parentspanid
- x-b3-sampled
- x-b3-flags
- x-ot-span-context

When you make downstream calls in your applications, make sure to include these headers.

Trace sampling

When using the Bookinfo sample application above, every time you access `/productpage` you see a corresponding trace in the Jaeger dashboard. This sampling rate (which is 100% in the BookInfo example) is suitable for a test or low traffic mesh, which is why it is used as the default for the demo installs.

In other configurations, Istio defaults to generating trace spans for 1 out of every 100 requests (sampling rate of of 1%).

To control the trace sampling percentage: in a running mesh, edit the `istio-pilot` deployment and change the environment variable with the following steps:

1. To open your text editor with the deployment configuration file loaded, run the following command:

```
kubectl -n istio-system edit deploy istio-pilot
```

2. Find the `PILOT_TRACE_SAMPLING` environment variable, and change the value: to your desired percentage.

In both cases, valid values are from 0.0 to 100.0 with a precision of 0.01.

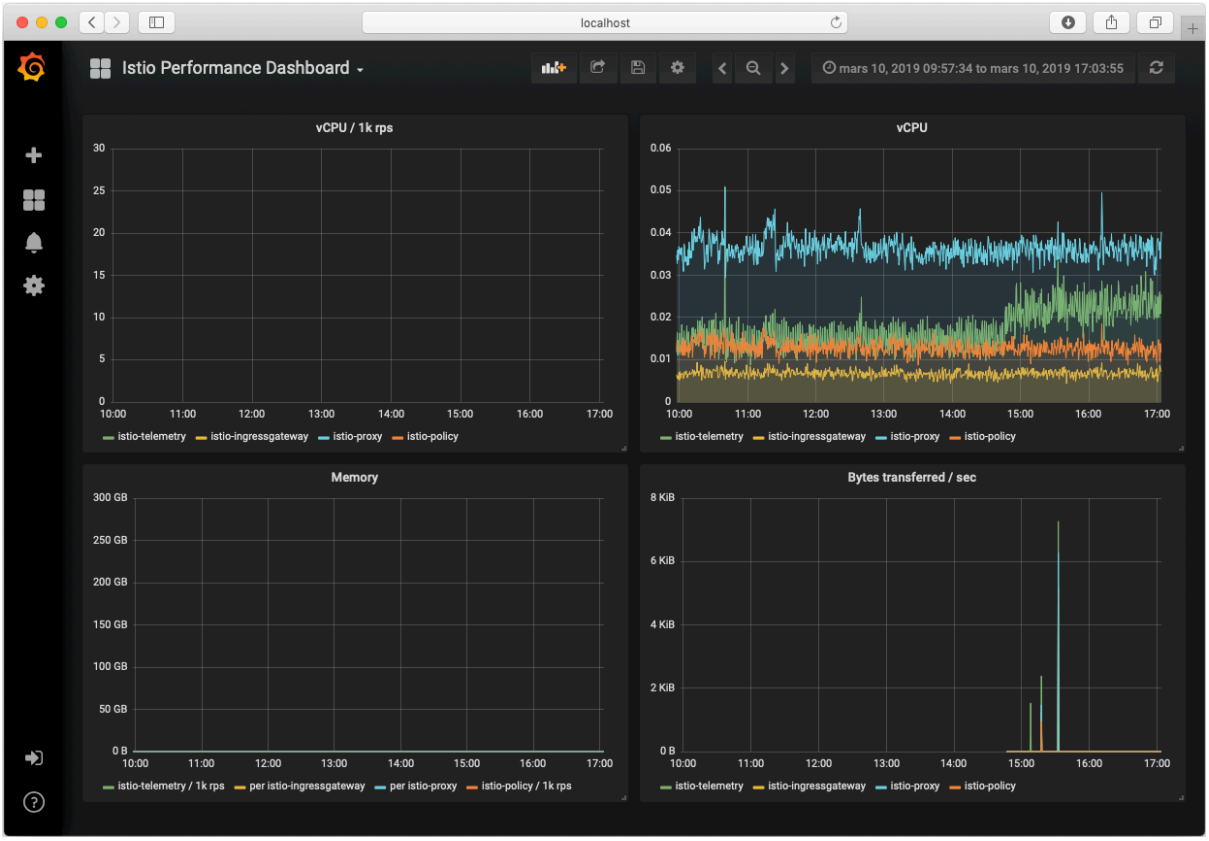
Grafana

The **Grafana** add-on in Istio is a preconfigured instance of **Grafana**.
The base image (`grafana/grafana:5.0.4`) has been modified to start with both a **Prometheus** data source and the *Istio Dashboard* installed. The base install files for *Istio*, and *Mixer* in particular, ship with a default configuration of global (used for every service) metrics. The *Istio Dashboard* is built to be used in conjunction with the default Istio metrics configuration and a **Prometheus** backend.

Establish port forwarding from local port 3000 to the **Grafana** instance:

```
kubectl -n istio-system port-forward \
  $(kubectl -n istio-system get pod -l app=grafana -o jsonpath='{.items[0].metadata.name}') \
  3000:3000 &
```

Browse to http://localhost:3000[6] and navigate to the **Istio Mesh Dashboard**:



Istio - Grafana Dashboard

Prometheus

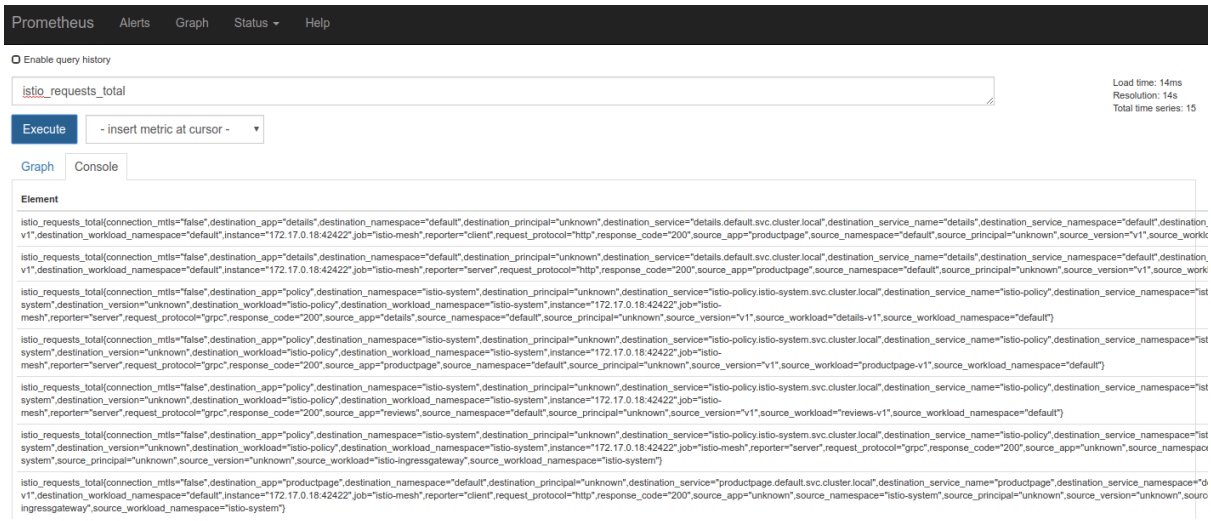
Mixer comes with a built-in **Prometheus** adapter that exposes an endpoint serving generated metric values. The **Prometheus** add-on is a **Prometheus** server that comes preconfigured to scrape Mixer endpoints to collect the exposed metrics. It provides a mechanism for persistent storage and querying of Istio metrics.

To access the **Prometheus Dashboard**, establish port forwarding from local port *9090* to *Prometheus* instance:

```
kubectl -n istio-system port-forward \
  $(kubectl -n istio-system get pod -l app=prometheus -o jsonpath='{.items[0].metadata.name}') \
  9090:9090
```

Browse to http://localhost:9090/graph[7], and in the *"Expression"* input box, enter: `istio_request_byte_count`. Click *Execute*:

[6] http://localhost:3000
[7] http://localhost:9090/graph



Istio - Prometheus Dashboard

Service Graph

The **ServiceGraph** service provides endpoints for generating and visualizing a graph of services within a mesh. It exposes the following endpoints:

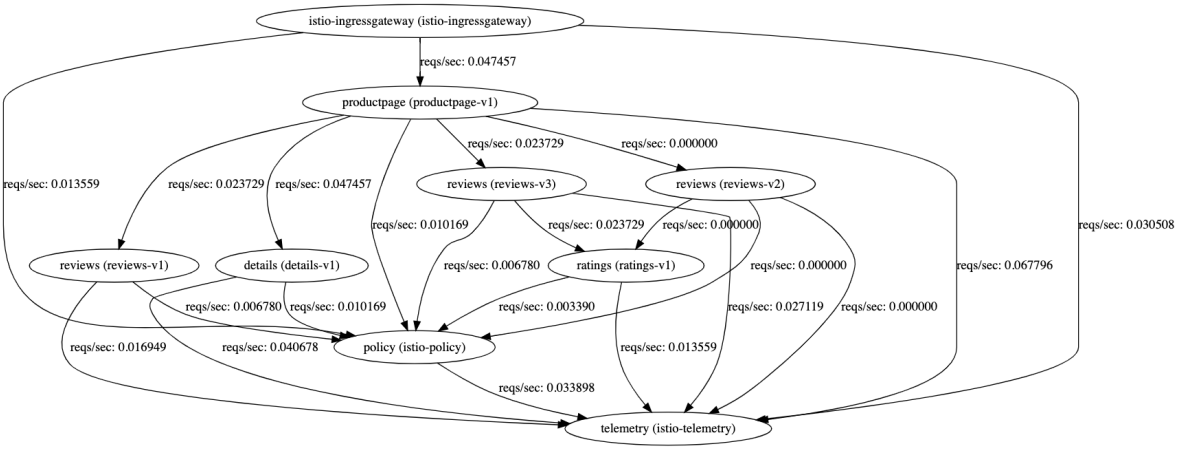
- `/force/forcegraph.html` As explored above, this is an interactive **D3.js** visualization.
- `/dotviz` is a static **Graphviz** visualization.
- `/dotgraph` provides a **DOT** serialization.
- `/d3graph` provides a JSON serialization for **D3** visualization.
- `/graph` provides a generic JSON serialization.

To access the **Service Graph Dashboard**, establish port forwarding from local port *8088* to *Service Graph* instance:

```
kubectl -n istio-system port-forward \
  $(kubectl -n istio-system get pod -l app=servicegraph -o jsonpath='{.items[0].metadata.name}') \
  8088:8088
```

Browse to http://localhost:8088/dotviz[8]:

[8] http://localhost:8088/dotviz



Services Graphviz visualization

The **ServiceGraph** example is built on top of **Prometheus** queries and depends on the standard Istio metric configuration.

Conclusion

That's all folks! We have been presenting the main concepts and components of Istio Service Mesh. This introducing chapter is just for installation and core concepts of Istio. As we saw in the Traffic Management section, there are infinite scenarios and use cases that you want to cover, like fault injection, controlling Ingress and Egress traffic, circuit breakers, etc..

We did not have the opportunity to cover the Security capabilities of Istio, just because it needs so much chapters to be covered.

Stay tuned, I will be writing some quickies about more great capabilities of Istio !

Bonus 2: Service Mesh: Linkerd

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/playing-with-java-microservices-on-k8s-and-ocp.