
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 2018-11-24

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

I would like to express my gratitude to the many people who saw me through this book.

To all those who provided support, read, wrote, offered comments and assisted in the editing and design.

To my wife and my family thank you for all your great support and encouragements.

To my friends thank you for always being there when I need you.

To Jérome Salles, thank you for teaching me the Java EE Art and for your support and guidance.

Last and not least, I beg forgiveness of all those who have been with me over the course of the years and whose names I have failed to mention.

Preface

Playing with Java Microservices on Kubernetes and OpenShift will teach you how to build and design microservices using Java and the Spring platform.

This book covers topics related to creating Java microservices and deploy them to Kubernetes and OpenShift.

Traditionally, Java developers have been used to developing large, complex monolithic applications. The experience of developing and deploying monoliths has been always slow and painful. This book will help Java developers to quickly get started with the features and the concerns of the microservices architecture. It will introduce Docker, Kubernetes and OpenShift to help them deploying their microservices.

The book is written for Java developers who wants to build microservices using the Spring Boot/Cloud stack and who wants to deploy them to Kubernetes and OpenShift.

You will be guided on how to install the appropriate tools to work properly. For those who are new to Enterprise Development using Spring Boot, you will be introduced to its core principles and main features thru a deep step-by-step tutorial on many components. For experts, this book offers some recipes that illustrate how to split monoliths and implement microservices and deploy them as containers to Kubernetes and OpenShift.

The following are some of the key challenges that we will address in this book:

- Introducing Spring Boot/Cloud for beginners
- Splitting a monolith using the Domain Driven Design approach
- Implementing the cloud & microservices patterns
- Rethinking the deployment process
- Introducing containerization, Docker, Kubernetes and OpenShift

By the end of reading this book, you will have practical hands-on experience of building microservices using Spring Boot/Cloud and you will master deploying them as containers to Kubernetes and OpenShift.

What this book covers

Chapter 1, Introduction to the Monolithic architecture, provides an introduction to the Monolithic architecture with a focus on its advantages and drawbacks.

Chapter 2, Coding the Monolithic application, offers a step-by-step tutorial for modeling and creating the monolith using Maven, Java 8 & Spring Boot 2.x.

Chapter 3, Upgrading the Monolithic application, lists some upgrades for our Monolithic application such as changing the DBMS..

Chapter 4, Building & Deploying the Monolithic application, covers how to build and package our example application before showing how to deploy it in the runtime environment.

Chapter 5, Microservices Architecture Pattern, presents the drawbacks of the Monolith architecture pattern and shows the motivations for seeking something new, like the Microservices architecture pattern.

Chapter 6, Splitting the Monolith: Bombarding the domain, presents the Domain Driven Design concepts to successfully understand the Business Domain of the Monolithic Application, to be able to split it into sub-domains.

Chapter 7, Applying DDD to the code, shows how to apply all the DDD concepts to the Monolithic Application source code and to successfully split the monolith into **Bounded Contexts**.

Chapter 8: Meeting the microservices concerns and patterns, covers a deep presentation of the concerns and the cloud patterns related to the Microservices Architecture Pattern.

Chapter 9: Implementing the patterns, shows how to implement the presented cloud patterns using Java & Spring Boot/Cloud stack.

Chapter 10: Building the standalone microservices, covers how to build our real standlone business microservices.

Chapter 11. Getting started with Docker, presents the containerization and Docker technology.

Chapter 12. Getting started with Kubernetes, presents in deep the greatest container orchestrator: Kubernetes

Chapter 13: The Kubernetes style, shows how to apply the Kubberentes concepts and features to our microservices.

Chapter 14: Getting started with Openshift, presents OpenShift Container Platform, a Kubernetes based **PaaS**.

As this book is published online, I will be providing regular updates for chapters, adding new ones, covering more items, etc..

If you want that I write about any item of the Java/Kubernetes ecosystem, feel free to get in touch with me directly on lnibrass@gmail.com[1].

[1] lnibrass@gmail.com

Part One: The Monolithics Era

Chapter 1: Introduction to the Monolithic architecture

Introduction to an actual situation

Let's imagine that we are developing an e-commerce backend application using Java & Spring Boot. Our e-commerce will have different clients including desktop browsers, mobile browsers and native mobile applications. The application can expose many APIs to be used by 3rd parties. The application handles many requests by executing business logic; accessing a database; exchanging messages with a broker and returning HTTP responses.

The e-commerce has many modules managing customers, orders, products, reviews, carts, authentication, etc..

Many team of developers are working on our application. Every team is working on a module.

Our application is a Java EE application. It is deployed as a huge `WAR` file that is wrapping all e-commerce modules. To deliver our application, the project manager has to be sure that all teams have delivered all the code of each module; if one of the teams had some delays for any reason, all the application will be delayed for sure, as it cannot be delivered with unfinished parts.
–> How can we solve this ?

Our technical stack is somehow sexy :D almost all Java Developers love working with Java 8 and Spring Boot. So when recruiting a new developer, we expect that our new developer will be happy to start working on the e-commerce. But does this will be true ? Our developer will be happy to meet all the code of the e-commerce when he is coming to work on the products module only ? Does this additional complexity is mandatory ?
–> How can we solve this ?

We are hearing about agility, does our project can adopt agility ? can we get benefit from it ?
–> How can we solve this ?

We know that there are many occasions and periods that our e-commerce will be highly loaded by customers, like Christmas, Valentine's day, etc.. So, we asked our operators to provide us with high-availability for our application. The solution were to have many running instances of our application to be sure to handle all the load. But does this solution is the best one ? Does our current architecture has benefits in matter of high availability ?
–> How can we solve this ?

Presenting the context

The actual architecture of our application is **Monolithic**. It consists of a single application that is :

- Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications
- Simple to deploy - we simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime

* Simple to scale - we can scale the application by running multiple copies of the application behind a load balancer

However, once the application becomes large and the team grows in size, this approach has a number of drawbacks that become increasingly significant:

- The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. It will be difficult to understand how to correctly implement a change, which will decrease the quality of the code.

- Overloaded development machines - large application will cause heavy load on the development machines, which decreases productivity.

- Overloaded servers - large application are hard to monitor and need huge server resources, which impacts productivity.

- Continuous deployment is difficult - a large monolithic application is also hard to deploy. In order to update one component you have to redeploy the entire application. The risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.

- Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension: just by creating copies of the monolith. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently.

- Obstacle to scaling development - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size it's useful to divide up the engineering organization into teams that focus on specific functional areas. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.

How to solve these issues ?

Here we will be talking about **Microservices**: the main subject of this book and one of the most trendy words of nowadays, at least now, when I am writing this words.. This is what we will be talking about in the next few chapters..

I wish you happy reading :D Good luck !

Chapter 2: Coding the Monolithic application

Presenting our domain

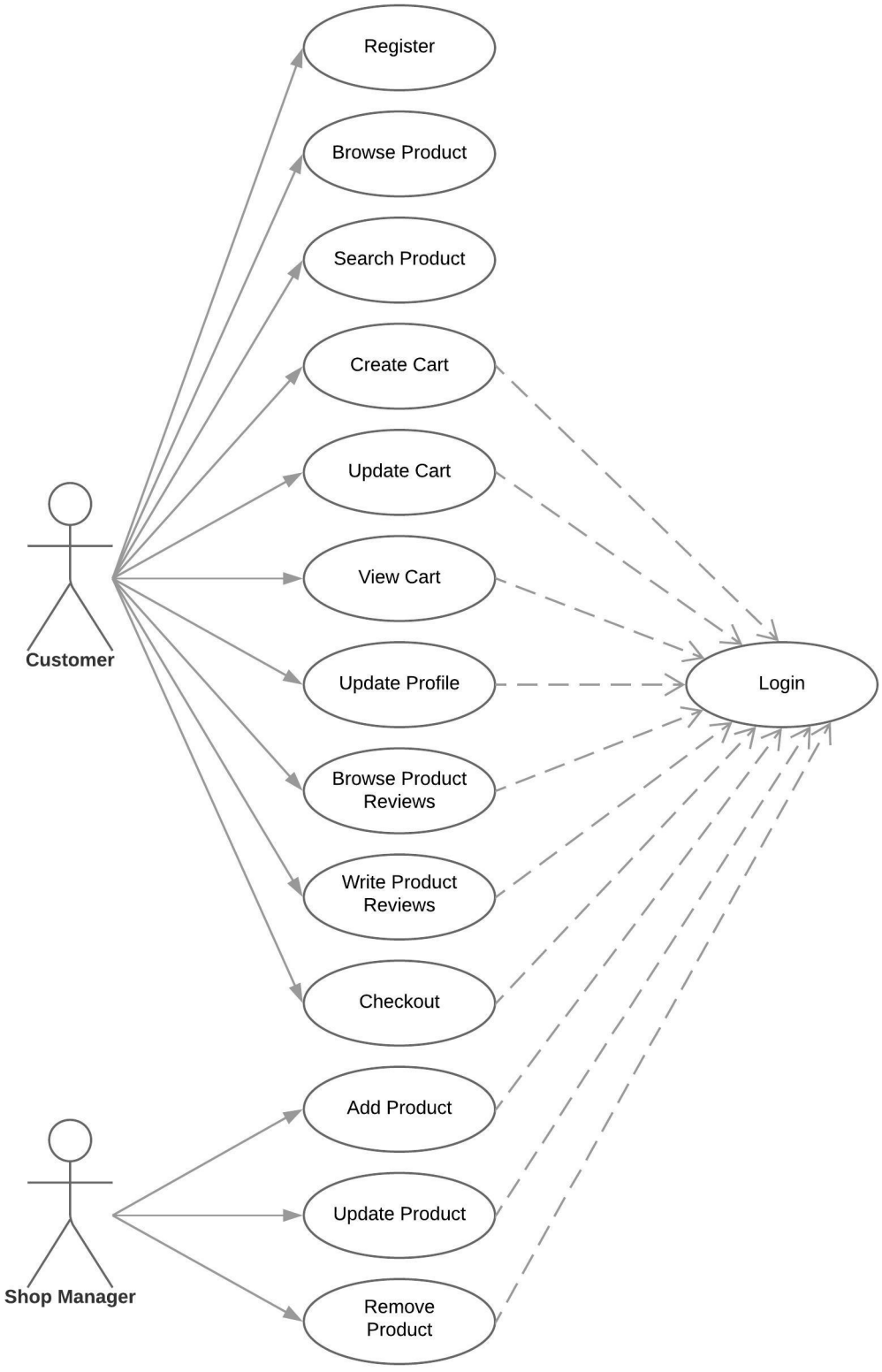
Our use-case will be one of the most classic examples. No, it will not be the “Hello World” example :smile: We will be dealing with a small e-commerce application.

Our application will be an online shop, where registered customers can purchase products that they found in our catalogue. A customer can read and post reviews about articles that he already purchased. To simplify the use-case, for payments we will be using a Stripe or Paypal payment gateway.

Use Case Diagram

The functionalities offered by the application:

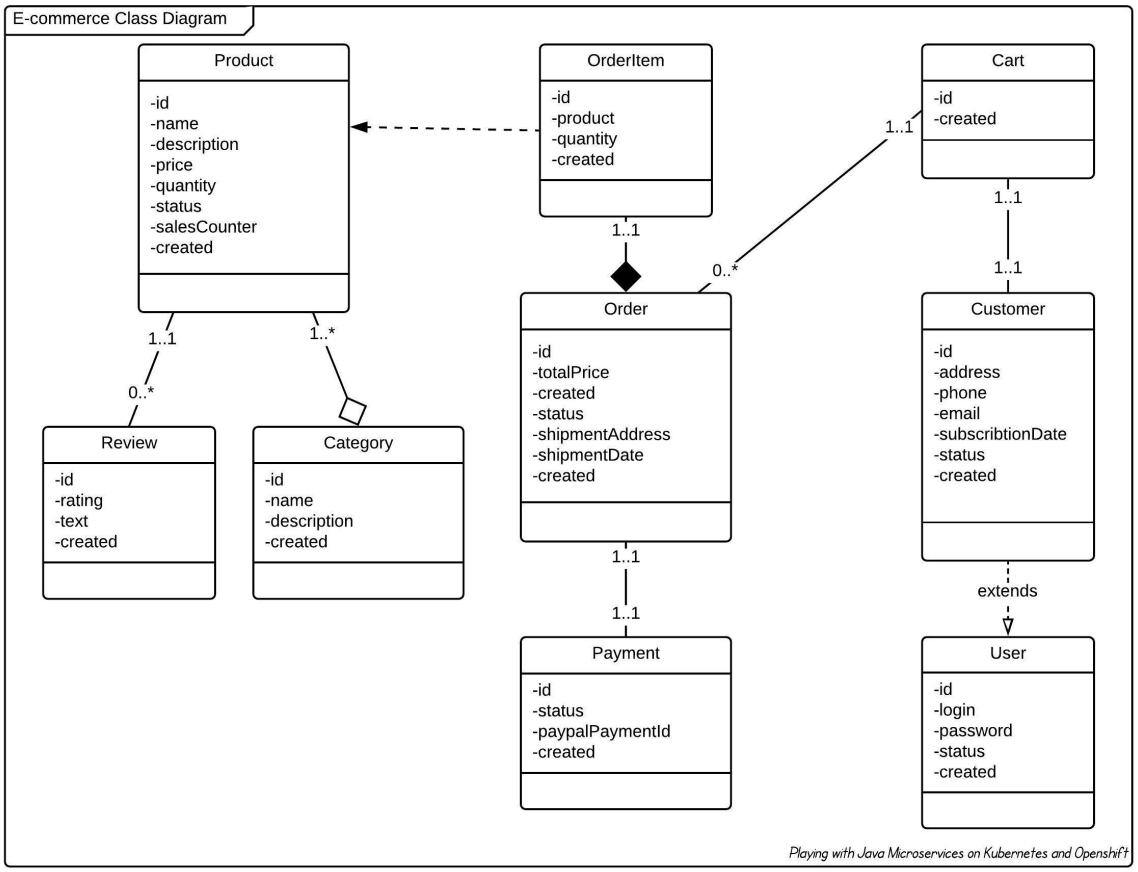
- For the Customer:
- Search and Browse Product
- Browse and Write Reviews
- Create, View and Update Cart
- Update Profile
- Checkout and pay
- For the Shop Manager:
- Add product
- Update product
- Remove product



Use Case Diagram for the e-commerce application

Class Diagram

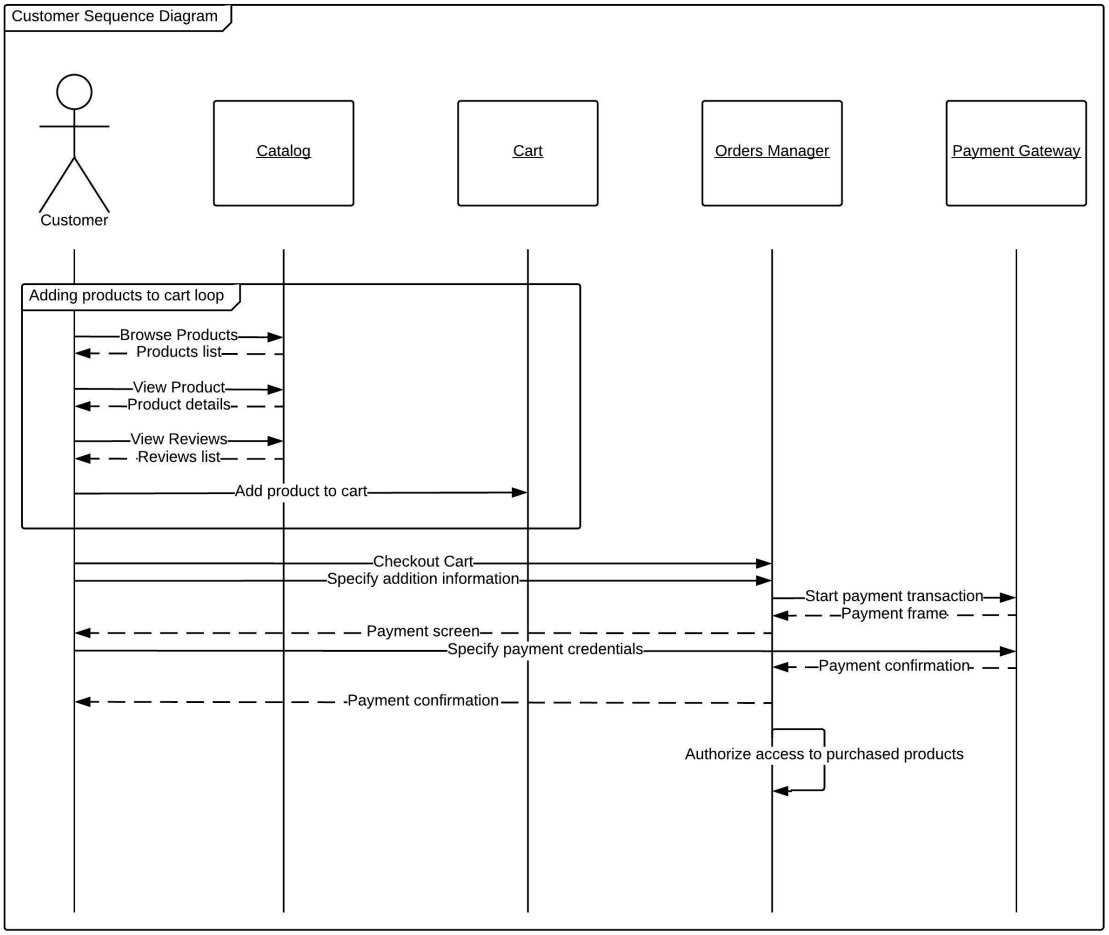
Our class diagram will look like this:



Class Diagram for the e-commerce application

Sequence Diagram

The sequence diagram for a classic shopping trip will look be:



Sequence Diagram for the customer

Coding the application

After we elaborated some UML parts, we can start coding the application.

But before attacking the code, let's list the technical stack that we will be using.

Presenting the technology stack

In this book we are implementing only the backend of an ecommerce application, so our stack will be:

- Java 1.8+
- Maven 3.0+
- Spring Boot 2.x

Java 8

Java Standard Edition 8 is a major feature edition, released in 18th March 2014.

Java 8 came with:

- Lambda expressions
- Method references
- Default Methods (Defender methods)
- A new Stream API.
- Optional
- A new Date/Time API.
- Nashorn, the new JavaScript engine
- Removal of the Permanent Generation
- and many other features…

Yes, you can think why we used Java 8 and not Java 9 or even 10? The choice was made on Java 8 because is the version most compatible with the frameworks and libraries we will be dealing with.

Maven

We will be based on the official introduction of Maven from its official website[2]:

Maven, a Yiddish word meaning accumulator of knowledge, was originally started as an attempt to simplify the build processes in the Jakarta Turbine project. There were several projects each with their own Ant build files that were all slightly different and JARs were checked into CVS. We wanted a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information and a way to share JARs across several projects.

The result is a tool that can now be used for building and managing any Java-based project. We hope that we have created something that will make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

[2] http://maven.apache.org

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal there are several areas of concern that Maven attempts to deal with:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features

Apache Maven Logo

Spring Boot

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

Features:

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' POMs to simplify your Maven configuration
- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

Spring Boot follows the “Convention over Configuration” paradigm. Spring Boot replaced all the XML based configurations and provided Annotations for using the Spring Framework. You can start your application with a very minimum annotation in no time. This would be very helpful to the developers, the team productivity would greatly be impacted positively.

Spring by Pivotal

NetBeans IDE

Apache NetBeans IDE is the original free Java IDE. It is actually the best IDE available for Java and web development. The IDE is designed to limit coding errors and optimize the productivity. There is no need for configuration or plugins, every thing is ready for you to start coding and making miracles. No more lost time for preparing the environment: Apache NetBeans can interact with all your environment components such as databases, automation servers, application servers, quality tools, …

Apache NetBeans IDE has a very awesome Debugger functionnalities to optimize the refactoring and debugging processes. It includes a very powerful text editor with refactoring tools and code completion, snippets and analyzers, and even multilanguages spell corrector. NetBeans offers also very powerful multisystem versioning using out-of-the-box integration with tools such as Git, SVN, CVS, Mercurial… Other than Java and JavaFX, Apache NetBeans IDE is supporting C/C++, PHP, HTML5/JavaScript. The Apache NetBeans IDE is available for all the operating systems that supports a compatible JVM.

Apache NetBeans Community is organizing many free conferences and meetings all over the world (USA, France, UK, Netherlands, Brazil…). You can assist to the events and acquire many skills on the Apache NetBeans and for sure programming.

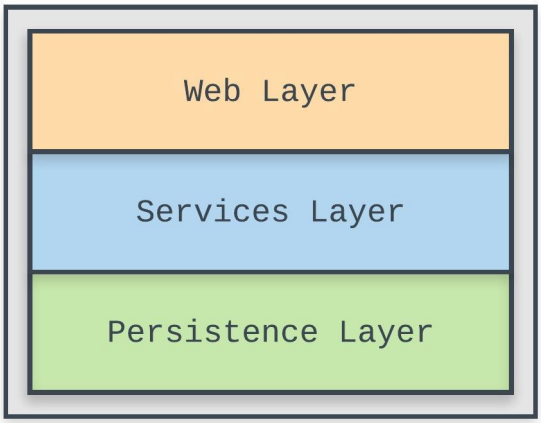
You can download the latest version of Apache NetBeans IDE on the official NetBeans website[3]

Apache NetBeans IDE Logo

[3] https://netbeans.org/downloads/index.html

Implementing the Boutique

Now, we will start the implementation of our application. We will split the implementation using the layers composing a typical Spring Application.



Application Layers

- **Persistence Layer** holds the Entities, Repositories and related classes.
- **Services Layer** holds the Services, Configuration, Batches, etc..
- **Web Layer** holds the webservices endpoints.

Generating the project skull

Here we go ! We will start using the great features and options offered by the Spring Framework Teams at Pivotal Software, Inc.

To avoid the hard parts when creating new project and getting it started, the Spring Team has created the **Spring Initializr** Project.

The Spring Initializr is a useful project that can generate a basic Spring Boot project structure easily. You can choose either your project to be based on Maven or Gradle, to use Java or Kotlin or Groovy, and to choose which version of Spring Boot you want to pick.

Spring Initializr can be used:

- Using a web-based interface http://start.spring.io
- Using Spring Tool Suite or other different IDEs like NetBeans & IntelliJ
- Using the Spring Boot CLI

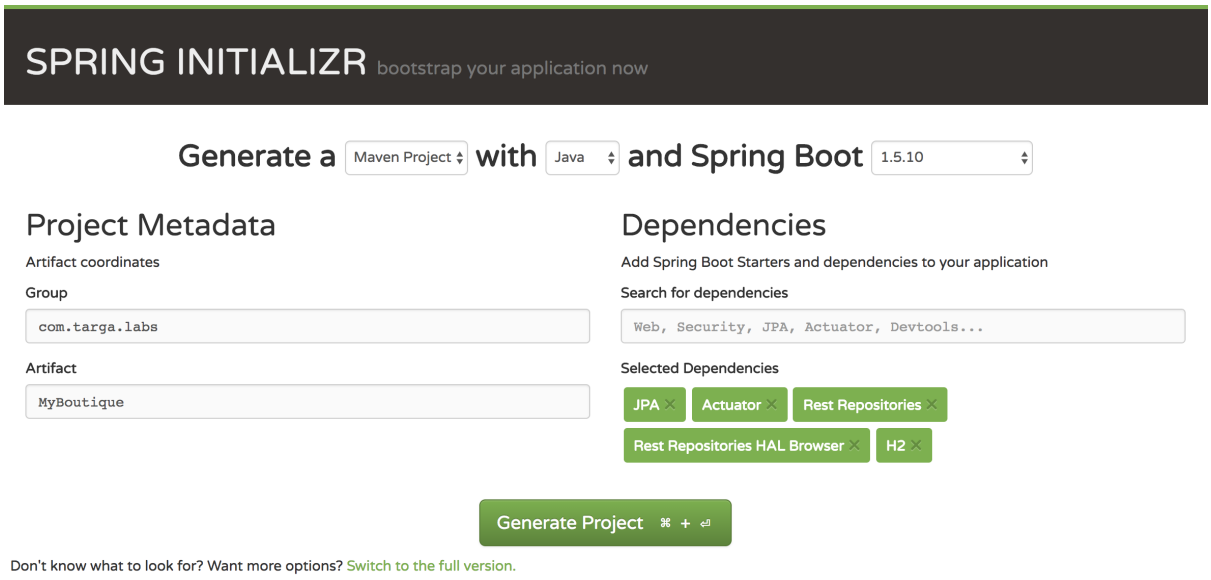
Spring Initializr gives you the ability to add the core dependencies that you need, like JDBC drivers or Spring Boot Starters.

Hey! What is Spring Boot Starters?
Spring Boot Starters are a set of convenient dependency descriptors that you can include in your application. You get a one-stop-shop for all the Spring and related technology that you need, without having to hunt through sample code and copy paste loads of dependency descriptors. For example, if you want to get started using Spring and JPA for database access, just include the spring-boot-starter-data-jpa dependency in your project,and you are good to go.
The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

You can learn more about Spring Boot Starters here: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter

For our Spring Boot project, the *Dependencies* we will choose:

- JPA
- Actuator
- H2
- Web



Spring Initializr - Bootstrapping MyBoutique

What is these dependenices?

- JPA: Support for the **Java Persistence API** including *spring-data-jpa*, *spring-orm* and *Hibernate*.
- Actuator: Production ready features to help you monitor and manage your application
- H2: very popular in-memory databases that has a very good integration in *Spring Boot*.
- Rest Repositories: Exposing *Spring Data* repositories over REST via *spring-data-rest-webmvc*.
- Rest Repositories HAL Browser: Browsing *Spring Data REST* repositories in your browser.

As we walk into our example we will add dependencies when we need them.

One we created the blank project, we will start building our monolith layers. We will begin by the **Persistence Layer**.

Creating the Persistence Layer

If we look back to our class diagram (link to class diagram), we have:

- Address
- Cart
- Category
- Customer
- Order
- OrderItem
- Payment
- Product
- Review

For every *entity* of the list, we will create:

- A JPA Entity
- A Spring Data JPA Repository

Our entities will share some attributes that are always commonly used, like 'id', 'created date', etc.. These attributes will be centralized in the 'AbstractEntity' class that will be 'extended' by our entities.

The `AbstractEntity` will look like:

```
@Getter <1>
@Setter <1>
@MappedSuperclass <2>
@EntityListeners(AuditingEntityListener.class) <3>
public abstract class AbstractEntity implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @CreatedDate <4>
    @Column(name = "created_date", nullable = false)
    @JsonIgnore
    private Instant createdDate = Instant.now();

    @LastModifiedDate <4>
```

```
    @Column(name = "last_modified_date")
    @JsonIgnore
    private Instant lastModifiedDate = Instant.now();

}
```

1. We are using Lombok annotations to generate getters and setters for the `AbstractEntity` class.
2. Annotating a base class which holds properties will be inherited.
3. Activates the Auditing in Spring Data JPA.
4. The annotated fields will be filled automatically based on the corresponding annotation by the `AuditingEntityListener`.

Cart

The class `Cart` entity will look like:

```
@Data <1>
@NoArgsConstructor <2>
@AllArgsConstructor <3>
@EqualsAndHashCode(callSuper = false) <4>
@Entity <5>
@Table(name = "cart") <5>
public class Cart extends AbstractEntity {

    private static final long serialVersionUID = 1L;

    @OneToOne
    @JoinColumn(unique = true)
    private Order order;

    @ManyToOne
    private Customer customer;

    @NotNull <6>
    @Enumerated(EnumType.STRING)
    private CartStatus status;

    public Cart(Customer customer) {
        this.customer = customer;
        this.status = CartStatus.NEW;
    }
}
```

1. This Lombok annotation generates getters for all fields, a useful `toString` method, and hashCode and equals implementations that check all non-transient fields. Will also generate setters for all non-final fields, as well as a constructor.
2. This Lombok annotation generates a no-args constructor.
3. This Lombok annotation generates an all-args constructor. An all-args constructor requires one argument for every field in the class.
4. Column definition: name, length and a definition of a Nullability Constraint.
5. This is an `@Entity` and its corresponding `@Table` will be named `cart`.

6. Validation annotations used to control the integrity of the data.

The difference between the **Validation annotations** and **Constraints** defined in the `@Column`, is that the **Validation annotations** is application-scoped and the **Constraints** are DB scoped.

The *CartRepository* will be:

```
@Repository <1>
public interface CartRepository extends JpaRepository<Cart, Long> <2> {

    List<Cart> findByStatus(CartStatus status); <3>

    List<Cart> findByStatusAndCustomerId(CartStatus status, Long customerId); <3>

}
```

1. Indicates that an annotated class is a "Repository", originally defined by Domain-Driven Design (Evans, 2003) as "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects".
2. JPA specific extension of Repository. This will enable Spring Data to find this interface and automatically create an implementation for it.
3. These methods are implementing queries automatically using Spring Data Query Builder Mecanism.

What is a JpaRepository ?

`JpaRepository` extends `PagingAndSortingRepository` which in turn extends `CrudRepository`.

Their main functions are:

- `CrudRepository` mainly provides CRUD functions.
- `PagingAndSortingRepository` provide methods to do pagination and sorting records.
- `JpaRepository` provides some JPA related method such as flushing the persistence context and delete record in a batch.

Because of the inheritance mentioned above, `JpaRepository` will have all the functions of `CrudRepository` and `PagingAndSortingRepository`. So if you don't need the repository to have the functions provided by `JpaRepository` and `PagingAndSortingRepository` , use `CrudRepository`.

What is Spring Data Query Builder Mecanism?

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes `find…By`, `read…By`, `query…By`, `count…By`, and `get…By` from the method and starts parsing the rest of it. The introducing clause can contain further expressions, such as a Distinct to set a distinct flag on the query to be created. However, the first By acts as delimiter to indicate the start of the actual criteria. At a very basic level, you can define conditions on entity properties and concatenate them with `And` and `Or`.

Address

The class `Address` will look like:

```
@Data
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
@Embeddable
public class Address {

    @Column(name = "address_1")
    private String address1;

    @Column(name = "address_2")
    private String address2;

    @Column(name = "city")
    private String city;

    @NotNull
    @Size(max = 10)
    @Column(name = "postcode", length = 10, nullable = false)
    private String postcode;

    @NotNull
    @Size(max = 2)
    @Column(name = "country", length = 2, nullable = false)
    private String country;

}
```

The class `Address` will be used as *Embeddable* class. Embeddable classes are used to represent the state of an entity but don't have a persistent identity of their own, unlike entity classes. Instances of an embeddable class share the identity of the entity that owns it. Embeddable classes exist only as the state of another entity.

Category

The `Category` entity:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
@Entity
@Table(name = "category")
public class Category extends AbstractEntity {

    @NotNull
    @Column(name = "name", nullable = false)
    private String name;

    @NotNull
    @Column(name = "description", nullable = false)
    private String description;

    @OneToMany(mappedBy = "category")
    @JsonIgnore
    private Set<Product> products = new HashSet<>();

}
```

The `CategoryRepository`:

```
@Repository
public interface CategoryRepository extends JpaRepository<Category, Long> {
}
```

Customer

The `Customer` entity:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
@Entity
@Table(name = "customer")
public class Customer extends AbstractEntity {

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "email")
    private String email;

    @Column(name = "telephone")
    private String telephone;

    @OneToMany(mappedBy = "customer")
    @JsonIgnore
    private Set<Cart> carts;
}
```

The `CustomerRepository`:

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findAllByEnabled(Boolean enabled);
}
```

Order

The `Order` entity:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
@Entity
@Table(name = "orders")
public class Order extends AbstractEntity {

    @NotNull
    @Column(name = "total_price", precision = 10, scale = 2, nullable = false)
    private BigDecimal totalPrice;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private OrderStatus status;

    @Column(name = "shipped")
    private ZonedDateTime shipped;

    @OneToOne
    @JoinColumn(unique = true)
    private Payment payment;

    @Embedded
    private Address shipmentAddress;

    @OneToMany(mappedBy = "order")
    @JsonIgnore
    private Set<OrderItem> orderItems;

    @OneToOne(mappedBy = "order")
    @JsonIgnore
    private Cart cart;

}
```

The `OrderRepository`:

```
@Repository
public interface OrderRepository extends JpaRepository<Order, Long> {

    public List<Order> findByCartCustomer_Id(Long id);
}
```

OrderItem

The `OrderItem` entity:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
@Entity
@Table(name = "order_item")
public class OrderItem extends AbstractEntity {

    @NotNull
    @Column(name = "quantity", nullable = false)
    private Long quantity;

    @ManyToOne
    private Product product;

    @ManyToOne
    private Order order;

}
```

The `OrderItemRepository`:

```
@Repository
public interface OrderItemRepository extends JpaRepository<OrderItem, Long> {
}
```

Payment

The `Payment` entity:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
@Entity
@Table(name = "payment")
public class Payment extends AbstractEntity {

    @Column(name = "paypal_payment_id")
    private String paypalPaymentId;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
```

```
    private PaymentStatus status;

    @OneToOne
    @JoinColumn(unique = true)
    private Order order;

}
```

The `PaymentRepository`:

```
@Repository
public interface PaymentRepository extends JpaRepository<Payment, Long> {
}
```

Product

The `Product` entity:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
@Entity
@Table(name = "product")
public class Product extends AbstractEntity {

    @NotNull
    @Column(name = "name", nullable = false)
    private String name;

    @NotNull
    @Column(name = "description", nullable = false)
    private String description;

    @NotNull
    @Column(name = "price", precision = 10, scale = 2, nullable = false)
    private BigDecimal price;

    @Column(name = "quantity")
    private Integer quantity;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private ProductStatus status;

    @Column(name = "sales_counter")
    private Integer salesCounter;

    @OneToMany
    @JsonIgnore
    private Set<Review> reviews = new HashSet<>();

    @ManyToOne
    private Category category;

}
```

The `ProductRepository`:

```
@Repository
public interface PaymentRepository extends JpaRepository<Payment, Long> {
}
```

Review

The `Review` entity:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
@Entity
@Table(name = "review")
public class Review extends AbstractEntity {

    @NotNull
    @Column(name = "text", nullable = false)
    private String title;

    @NotNull
    @Column(name = "description", nullable = false)
    private String description;

    @NotNull
    @Column(name = "rating", nullable = false)
    private Long rating;

}
```

The `ReviewRepository`:

```
@Repository
public interface ReviewRepository extends JpaRepository<Review, Long> {
}
```

Creating the Service Layer

Now we created our entities, we will create the services.

A `service` in a **Spring Application** is a components that wraps the Business Logic. It's also the glue between the **Persistence** and the **Web** layers.

At this point, we didn't discussed any *Business Logic*, we just have *CRUD* operations. We will implement these *CRUD* operations in our services. Yeah ! It's obvious, we will have a `service` per `entity`.

Typical Service: CartService

The `CartService` will look like:

```
@RequiredArgsConstructor <1>
@Slf4j <2>
@Service <3>
@Transactional <4>
public class CartService {

    private final CartRepository cartRepository; <5>
    private final CustomerRepository customerRepository; <5>
    private final OrderService orderService; <5>

    public List<CartDto> findAll() {
        log.debug("Request to get all Carts");
        return this.cartRepository.findAll()
                .stream()
                .map(CartService::mapToDto)
                .collect(Collectors.toList());
    }

    public List<CartDto> findAllActiveCarts() {
        return this.cartRepository.findByStatus(CartStatus.NEW)
                .stream()
                .map(CartService::mapToDto)
                .collect(Collectors.toList());
    }

    public CartDto create(Long customerId) {
        if (this.getActiveCart(customerId) == null) {
            Customer customer = this.customerRepository.findById(customerId)
                    .orElseThrow(() -> new IllegalStateException("The Customer does not exist!"));

            Cart cart = new Cart(
                    null,
                    customer,
                    CartStatus.NEW
            );
            Order order = this.orderService.create(cart);
            cart.setOrder(order);

            return mapToDto(this.cartRepository.save(cart));
        } else {
            throw new IllegalStateException("There is already an active cart");
        }
    }

    @Transactional(readOnly = true)
    public CartDto findById(Long id) {
        log.debug("Request to get Cart : {}", id);
        return this.cartRepository.findById(id).map(CartService::mapToDto).orElse(null);
    }

    public void delete(Long id) {
        log.debug("Request to delete Cart : {}", id);
        Cart cart = this.cartRepository.findById(id)
                .orElseThrow(() -> new IllegalStateException("Cannot find cart with id " + id));

        cart.setStatus(CartStatus.CANCELED);
```

```
        this.cartRepository.save(cart);
    }

    public CartDto getActiveCart(Long customerId) {
        List<Cart> carts = this.cartRepository
                .findByStatusAndCustomerId(CartStatus.NEW, customerId);
        if (carts != null) {

            if (carts.size() == 1) {
                return mapToDto(carts.get(0));
            }
            if (carts.size() > 1) {
                throw new IllegalStateException("Many active carts detected !!!");
            }
        }

        return null;
    }

    public static CartDto mapToDto(Cart cart) {
        if (cart != null) {
            return new CartDto(
                    cart.getId(),
                    cart.getOrder().getId(),
                    CustomerService.mapToDto(cart.getCustomer()),
                    cart.getStatus().name()
            );
        }
        return null;
    }

}
```

1. Lombok annotation used to generate a constructor with required arguments. Required arguments are final fields and fields with constraints such as @NonNull.
2. Lombok annotation used to generate a `logger` in the class. When used, you then have a `static final log` field, initialized to the name of your class, which you can then use to write log statements.
3. Indicates that an annotated class is a `Service`, originally defined by **Domain-Driven Design** (Evans, 2003) as "an operation offered as an interface that stands alone in the model, with no encapsulated state."
4. Spring `@Transactional` annotation is used for defining a transaction boundary.
5. We injected the `Repository` in the `Service` using a **constructor injection** and not using the **field** or **setter** injection. The constructor used for the injection is generated using Lombok.

Why I like Constructor Dependency Injection ?

> I prefer to to use constructor-based dependecies injection, for many reasons:
> - Testability: The easiest way to pass `mock` objects in constructor while testing.
> - Safety: This way forces Spring to provide mandatory dependencies, making sure every object created is in a valid state after construction.
> - Readability: Dependencies passed as paramters to the constructor are the *mandatory* dependencies. So dependencies injected using fields or setters are *optional* dependencies.

And the `CartDto` class will look like:

```
@Data
@AllArgsConstructor
public class CartDto {
    private Long id;
    private Long orderId;
    private CustomerDto customerDto;
    private String status;
}
```

AddressService

The `AddressService` class will look like:

```
public class AddressService {

    public static AddressDto mapToDto(Address address) {
        if (address != null) {
            return new AddressDto(
                    address.getAddress1(),
                    address.getAddress2(),
                    address.getCity(),
                    address.getPostcode(),
                    address.getCountry()
            );
        }
        return null;
    }
}
```

And the `AddressDto` class will look like:

```
@Data
@AllArgsConstructor
public class AddressDto {
    private String address1;
    private String address2;
    private String city;
    private String postcode;
    private String country;
}
```

CategoryService

The `CategoryService` class will look like:

```
@RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class CategoryService {

    private final CategoryRepository categoryRepository;

    public List<CategoryDto> findAll() {
        log.debug("Request to get all Categories");
        return this.categoryRepository.findAll()
                .stream()
                .map(CategoryService::mapToDto)
                .collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
    public CategoryDto findById(Long id) {
        log.debug("Request to get Category : {}", id);
        return this.categoryRepository.findById(id).map(CategoryService::mapToDto)
                .orElseThrow(IllegalStateException::new);
    }

    public CategoryDto create(CategoryDto categoryDto) {
        log.debug("Request to create Category : {}", categoryDto);
        return mapToDto(this.categoryRepository.save(
                new Category(
                        categoryDto.getName(),
                        categoryDto.getDescription(),
                        Collections.emptySet()
                )
        ));
    }

    public void delete(Long id) {
        log.debug("Request to delete Category : {}", id);
        this.categoryRepository.deleteById(id);
    }

    public static CategoryDto mapToDto(Category category) {
        if (category != null) {
            return new CategoryDto(
```

```
                    category.getId(),
                    category.getName(),
                    category.getDescription(),
                    category.getProducts().stream().map(ProductService::mapToDto).collect(Collectors.t\
oSet())
            );
        }
        return null;
    }
}
```

And the `CategoryDto` class will look like:

```
@Data
@AllArgsConstructor
public class CategoryDto {
    private Long id;
    private String name;
    private String description;
    private Set<ProductDto> products;
}
```

CustomerService

The `CustomerService` class will look like:

```
@RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerDto create(CustomerDto customerDto) {
        log.debug("Request to create Customer : {}", customerDto);
        return mapToDto(
                this.customerRepository.save(
                        new Customer(
                                customerDto.getFirstName(),
                                customerDto.getLastName(),
                                customerDto.getEmail(),
                                customerDto.getTelephone(),
                                Collections.emptySet(),
                                Boolean.TRUE
                        )
                )
        );
    }

    public List<CustomerDto> findAll() {
        log.debug("Request to get all Customers");
        return this.customerRepository.findAll()
                .stream()
                .map(CustomerService::mapToDto)
```

```
                .collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
    public CustomerDto findById(Long id) {
        log.debug("Request to get Customer : {}", id);
        return this.customerRepository.findById(id).map(CustomerService::mapToDto).orElse(null);
    }

    public List<CustomerDto> findAllActive() {
        log.debug("Request to get all Customers");
        return this.customerRepository.findAllByEnabled(true)
                .stream()
                .map(CustomerService::mapToDto)
                .collect(Collectors.toList());
    }

    public List<CustomerDto> findAllInactive() {
        log.debug("Request to get all Customers");
        return this.customerRepository.findAllByEnabled(false)
                .stream()
                .map(CustomerService::mapToDto)
                .collect(Collectors.toList());
    }

    public void delete(Long id) {
        log.debug("Request to delete Customer : {}", id);

        Customer customer = this.customerRepository.findById(id)
                .orElseThrow(() -> new IllegalStateException("Cannot find Customer with id " + id));

        customer.setEnabled(false);
        this.customerRepository.save(customer);
    }

    public static CustomerDto mapToDto(Customer customer) {
        if (customer != null) {
            return new CustomerDto(
                    customer.getId(),
                    customer.getFirstName(),
                    customer.getLastName(),
                    customer.getEmail(),
                    customer.getTelephone()
            );
        }
        return null;
    }

}
```

And the `CustomerDto` class will look like:

```
@Data
@AllArgsConstructor
public class CustomerDto {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String telephone;
}
```

OrderItemService

The `OrderItemService` class will look like:

```
RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class OrderItemService {

    private final OrderItemRepository orderItemRepository;
    private final OrderRepository orderRepository;
    private final ProductRepository productRepository;

    public List<OrderItemDto> findAll() {
        log.debug("Request to get all OrderItems");
        return this.orderItemRepository.findAll()
                .stream()
                .map(OrderItemService::mapToDto)
                .collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
    public OrderItemDto findById(Long id) {
        log.debug("Request to get OrderItem : {}", id);
        return this.orderItemRepository.findById(id).map(OrderItemService::mapToDto).orElse(null);
    }

    public OrderItemDto create(OrderItemDto orderItemDto) {
        log.debug("Request to create OrderItem : {}", orderItemDto);
        Order order = this.orderRepository.findById(orderItemDto.getOrderId()).orElseThrow(() -> new I\
llegalStateException("The Order does not exist!"));
        Product product = this.productRepository.findById(orderItemDto.getProductId()).orElseThrow(() \
-> new IllegalStateException("The Product does not exist!"));

        return mapToDto(
                this.orderItemRepository.save(
                        new OrderItem(
                                orderItemDto.getQuantity(),
                                product,
                                order
                        )));
    }

    public void delete(Long id) {
        log.debug("Request to delete OrderItem : {}", id);
```

```
        this.orderItemRepository.deleteById(id);
    }

    public static OrderItemDto mapToDto(OrderItem orderItem) {
        if (orderItem != null) {
            return new OrderItemDto(
                    orderItem.getId(),
                    orderItem.getQuantity(),
                    orderItem.getProduct().getId(),
                    orderItem.getOrder().getId()
            );
        }
        return null;
    }

}
```

And the `OrderItemDto` class will look like:

```
@AllArgsConstructor
@Data
public class OrderItemDto {
    private Long id;
    private Long quantity;
    private Long productId;
    private Long orderId;
}
```

OrderService

The `OrderService` class will look like:

```
@RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class OrderService {

    private final OrderRepository orderRepository;

    public List<OrderDto> findAll() {
        log.debug("Request to get all Orders");
        return this.orderRepository.findAll()
                .stream()
                .map(OrderService::mapToDto)
                .collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
    public OrderDto findById(Long id) {
        log.debug("Request to get Order : {}", id);
        return this.orderRepository.findById(id).map(OrderService::mapToDto).orElse(null);
    }

    public List<OrderDto> findAllByUser(Long id) {
```

```
        return this.orderRepository.findByCartCustomer_Id(id)
                .stream()
                .map(OrderService::mapToDto)
                .collect(Collectors.toList());
    }

    public OrderDto create(OrderDto orderDto) {
        log.debug("Request to create Order : {}", orderDto);
        return mapToDto(
                this.orderRepository.save(
                        new Order(
                                BigDecimal.ZERO,
                                OrderStatus.CREATION,
                                null,
                                null,
                                null,
                                Collections.emptySet(),
                                null
                        )
                )
        );
    }

    public Order create(Cart cart) {
        log.debug("Request to create Order with a Cart : {}", cart);
        return this.orderRepository.save(
                new Order(
                        BigDecimal.ZERO,
                        OrderStatus.CREATION,
                        null,
                        null,
                        null,
                        Collections.emptySet(),
                        cart
                )
        );
    }

    public void delete(Long id) {
        log.debug("Request to delete Order : {}", id);
        this.orderRepository.deleteById(id);
    }


    public static OrderDto mapToDto(Order order) {
        if (order != null) {
            return new OrderDto(
                    order.getId(),
                    order.getTotalPrice(),
                    order.getStatus().name(),
                    order.getShipped(),
                    PaymentService.mapToDto(order.getPayment()),
                    AddressService.mapToDto(order.getShipmentAddress()),
                    order.getOrderItems().stream().map(OrderItemService::mapToDto).collect(Collectors.\
toSet())
            );
        }
```

```
        return null;
    }
}
```

And the `OrderDto` class will look like:

```
@Data
@AllArgsConstructor
public class OrderDto {
    private Long id;
    private BigDecimal totalPrice;
    private String status;
    private ZonedDateTime shipped;
    private PaymentDto payment;
    private AddressDto shipmentAddress;
    private Set<OrderItemDto> orderItems;
}
```

PaymentService

The `PaymentService` class will look like:

```
@RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class PaymentService {

    private final PaymentRepository paymentRepository;
    private final OrderRepository orderRepository;

    public List<PaymentDto> findAll() {
        log.debug("Request to get all Payments");
        return this.paymentRepository.findAll()
                .stream()
                .map(PaymentService::mapToDto)
                .collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
    public PaymentDto findById(Long id) {
        log.debug("Request to get Payment : {}", id);
        return this.paymentRepository.findById(id).map(PaymentService::mapToDto).orElse(null);
    }

    public PaymentDto create(PaymentDto paymentDto) {
        log.debug("Request to create Payment : {}", paymentDto);

        Order order = this.orderRepository.findById(paymentDto.getOrderId()).orElseThrow(() -> new Ill\
egalStateException("The Order does not exist!"));

        return mapToDto(this.paymentRepository.save(
                new Payment(
                        paymentDto.getPaypalPaymentId(),
                        PaymentStatus.valueOf(paymentDto.getStatus()),
```

```
                        order
                )
        ));
    }

    public void delete(Long id) {
        log.debug("Request to delete Payment : {}", id);
        this.paymentRepository.deleteById(id);
    }

    public static PaymentDto mapToDto(Payment payment) {
        if (payment != null) {
            return new PaymentDto(
                    payment.getId(),
                    payment.getPaypalPaymentId(),
                    payment.getStatus().name(),
                    payment.getOrder().getId()
            );
        }
        return null;
    }
}
```

And the `PaymentDto` class will look like:

```
@Data
@AllArgsConstructor
public class PaymentDto {
    private Long id;
    private String paypalPaymentId;
    private String status;
    private Long orderId;
}
```

ReviewService

The `ReviewService` class will look like:

```
@RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class ReviewService {

    private final ReviewRepository reviewRepository;

    public List<ReviewDto> findAll() {
        log.debug("Request to get all Reviews");
        return this.reviewRepository.findAll()
                .stream()
                .map(ReviewService::mapToDto)
                .collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
```

```
    public ReviewDto findById(Long id) {
        log.debug("Request to get Review : {}", id);
        return this.reviewRepository.findById(id).map(ReviewService::mapToDto).orElse(null);
    }

    public ReviewDto create(ReviewDto reviewDto) {
        log.debug("Request to create Review : {}", reviewDto);
        return mapToDto(this.reviewRepository.save(
                new Review(
                        reviewDto.getTitle(),
                        reviewDto.getDescription(),
                        reviewDto.getRating()
                )
        ));
    }

    public void delete(Long id) {
        log.debug("Request to delete Review : {}", id);
        this.reviewRepository.deleteById(id);
    }

    public static ReviewDto mapToDto(Review review) {
        if (review != null) {
            return new ReviewDto(
                    review.getId(),
                    review.getTitle(),
                    review.getDescription(),
                    review.getRating()
            );
        }
        return null;
    }
}
```

And the `ReviewDto` class will look like:

```
@Data
@AllArgsConstructor
public class ReviewDto {
    private Long id;
    private String title;
    private String description;
    private Long rating;
}
```

Creating the Web Layer

In this section, we will expose the operations that we implemented in the `Services` as REST webservices.

In Spring Framework, a REST webservice can be implemented using a `RestController`.

Typical RestController: CartResource

The `CartResource` will look like:

```
@RequiredArgsConstructor <1>
@RestController <2>
@RequestMapping(API + "/carts") <2>
public class CartResource {

    private final CartService cartService;

    @GetMapping
    public List<CartDto> findAll() { <3>
        return this.cartService.findAll();
    }

    @GetMapping("/active")
    public List<CartDto> findAllActiveCarts() {
        return this.cartService.findAllActiveCarts();
    }

    @GetMapping("/customer/{id}")
    public CartDto getActiveCartForCustomer(@PathVariable("id") Long customerId) {
        return this.cartService.getActiveCart(customerId);
    }

    @GetMapping("/{id}")
    public CartDto findById(@PathVariable Long id) {
        return this.cartService.findById(id);
    }

    @PostMapping("/customer/{id}")
    public CartDto create(@PathVariable("id") Long customerId) {
        return this.cartService.create(customerId);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        this.cartService.delete(id);
    }

}
```

1. Lombok annotation used to generate a Required-Args-Constructor. We are using it for the Dependency Injection.
2. The `@RestController` and `@RequestMapping("/api/carts")` are used to declare the `CartResource` class as a REST Webservice reacheable at `http://SERVER_URL:SERVER_PORT/api/carts`.
3. We are returning `CartDto` object as a response of the REST Webservice, so when invoked, Spring Boot will use Jackson for serializing the object as JSON.

In this `RestController`, we are declaring:

- Listing all `Carts`: **HTTP GET** on `/api/carts`
- Listing active `Carts`: **HTTP GET** on `/api/carts/active`
- Listing active `Cart` for a customer: **HTTP GET** on `/api/carts/customer/{id}` with `{id}` is a holder of the Customer's ID.

- Listing all details of a `Cart`: **HTTP GET** on `api/carts/{id}` with `{id}` is a holder of the Cart's ID.
- Creating a new `Cart` for a giving `Customer`: **HTTP POST** on `/api/carts/customer/{id}` with `{id}` is a holder of the Customer's ID.
- Deleting a `Cart`: **HTTP DELETE** on `/api/carts/{id}` with `{id}`.

These operations will be described using *SWAGGER* API Framework. The API description will be generated automatically. We will talk about this later.

CategoryResource

The `CategoryResource` class will look like:

```
@RequiredArgsConstructor
@RestController
@RequestMapping(API + "/categories")
public class CategoryResource {

    private final CategoryService categoryService;

    @GetMapping
    public List<CategoryDto> findAll() {
        return this.categoryService.findAll();
    }

    @GetMapping("/{id}")
    public CategoryDto findById(@PathVariable Long id) {
        return this.categoryService.findById(id);
    }

    @PostMapping
    public CategoryDto create(CategoryDto categoryDto) {
        return this.categoryService.create(categoryDto);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        this.categoryService.delete(id);
    }
}
```

CustomerResource

The `CustomerResource` class will look like:

```
@RequiredArgsConstructor
@RestController
@RequestMapping(API + "/customers")
public class CustomerResource {

    private final CustomerService customerService;

    @GetMapping
    public List<CustomerDto> findAll() {
        return this.customerService.findAll();
    }

    @GetMapping("/{id}")
    public CustomerDto findById(@PathVariable Long id) {
        return this.customerService.findById(id);
    }

    @GetMapping("/active")
    public List<CustomerDto> findAllActive() {
        return this.customerService.findAllActive();
    }

    @GetMapping("/inactive")
    public List<CustomerDto> findAllInactive() {
        return this.customerService.findAllInactive();
    }

    @PostMapping
    public CustomerDto create(CustomerDto customerDto) {
        return this.customerService.create(customerDto);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        this.customerService.delete(id);
    }

}
```

OrderItemResource

The `OrderItemResource` class will look like:

```
@RequiredArgsConstructor
@RestController
@RequestMapping(API + "/order-items")
public class OrderItemResource {

    private final OrderItemService itemService;

    @GetMapping
    public List<OrderItemDto> findAll() {
        return this.itemService.findAll();
    }

    @GetMapping("/{id}")
```

```
    public OrderItemDto findById(@PathVariable Long id) {
        return this.itemService.findById(id);
    }

    @PostMapping
    public OrderItemDto create(@RequestBody OrderItemDto orderItemDto) {
        return this.itemService.create(orderItemDto);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        this.itemService.delete(id);
    }

}
```

OrderResource

The `OrderResource` class will look like:

```
@RequiredArgsConstructor
@RestController
@RequestMapping(API + "/orders")
public class OrderResource {

    private final OrderService orderService;

    @GetMapping
    public List<OrderDto> findAll() {
        return this.orderService.findAll();
    }

    @GetMapping("/customer/{id}")
    public List<OrderDto> findAllByUser(@PathVariable Long id) {
        return this.orderService.findAllByUser(id);
    }

    @GetMapping("/{id}")
    public OrderDto findById(@PathVariable Long id) {
        return this.orderService.findById(id);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        this.orderService.delete(id);
    }
}
```

PaymentResource

The `PaymentResource` class will look like:

```
@RequiredArgsConstructor
@RestController
@RequestMapping(API + "/payments")
public class PaymentResource {

    private final PaymentService paymentService;

    @GetMapping
    public List<PaymentDto> findAll() {
        return this.paymentService.findAll();
    }

    @GetMapping("/{id}")
    public PaymentDto findById(@PathVariable Long id) {
        return this.paymentService.findById(id);
    }

    @PostMapping
    public PaymentDto create(@RequestBody PaymentDto orderItemDto) {
        return this.paymentService.create(orderItemDto);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        this.paymentService.delete(id);
    }
}
```

ProductResource

The `ProductResource` class will look like:

```
@RequiredArgsConstructor
@RestController
@RequestMapping(API + "/products")
public class ProductResource {

    private final ProductService productService;

    @GetMapping
    public List<ProductDto> findAll() {
        return this.productService.findAll();
    }

    @GetMapping("/{id}")
    public ProductDto findById(@PathVariable Long id) {
        return this.productService.findById(id);
    }

    @PostMapping
    public ProductDto create(@RequestBody ProductDto productDto) {
        return this.productService.create(productDto);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
```

```
        this.productService.delete(id);
    }
}
```

ReviewResource

The `ReviewResource` class will look like:

```
@RequiredArgsConstructor
@RestController
@RequestMapping(API + "/reviews")
public class ReviewResource {

    private final ReviewService reviewService;

    @GetMapping
    public List<ReviewDto> findAll() {
        return this.reviewService.findAll();
    }

    @GetMapping("/{id}")
    public ReviewDto findById(@PathVariable Long id) {
        return this.reviewService.findById(id);
    }

    @PostMapping
    public ReviewDto create(@RequestBody ReviewDto reviewDto) {
        return this.reviewService.create(reviewDto);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        this.reviewService.delete(id);
    }
}
```

Automated API documentation

Swagger 2 is an open source project used to describe and document RESTful APIs. **Swagger 2** is language-agnostic and is extensible into new technologies and protocols beyond HTTP. The current version defines a set HTML, JavaScript and CSS assets to dynamically generate documentation from a **Swagger**-compliant API. These files are bundled by the **Swagger UI** project to display the API on browser. Besides rendering documentation, **Swagger UI** allows other API developers or consumers to interact with the API's resources without having any of the implementation logic in place.

The **Swagger 2** specification, which is known as **OpenAPI** specification has several implementations. We will be using the **Springfox** implementation in our project.

Springfox is automating the generation of the API Documentation. **Springfox** works by examining an application, once, at runtime to infer API semantics based on spring configurations, class structure and various compile time java Annotations.

To enable the Swagger capabilities to our project, we will be:

- Adding the Maven Dependencies
- Adding the Java Configuration

Let's put everything in action !

Maven Dependencies

We need to add some Maven Dependencies to our `pom.xml` file:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.0</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.0</version>
</dependency>
```

Java Configuration

For the Java Configuration, we need to create a **Spring** `@Bean` that configure **Swagger** in the application context:

```
@Configuration
public class SwaggerConfiguration {

    @Bean
    public Docket productApi() {
        return new Docket(DocumentationType.SWAGGER_2).select()
                .apis(RequestHandlerSelectors.any())
                .paths(PathSelectors.any())
                .build();
    }
}
```

Also, we need to add the `@EnableSwagger2` annotation to the SpringBootApplication Main Class:

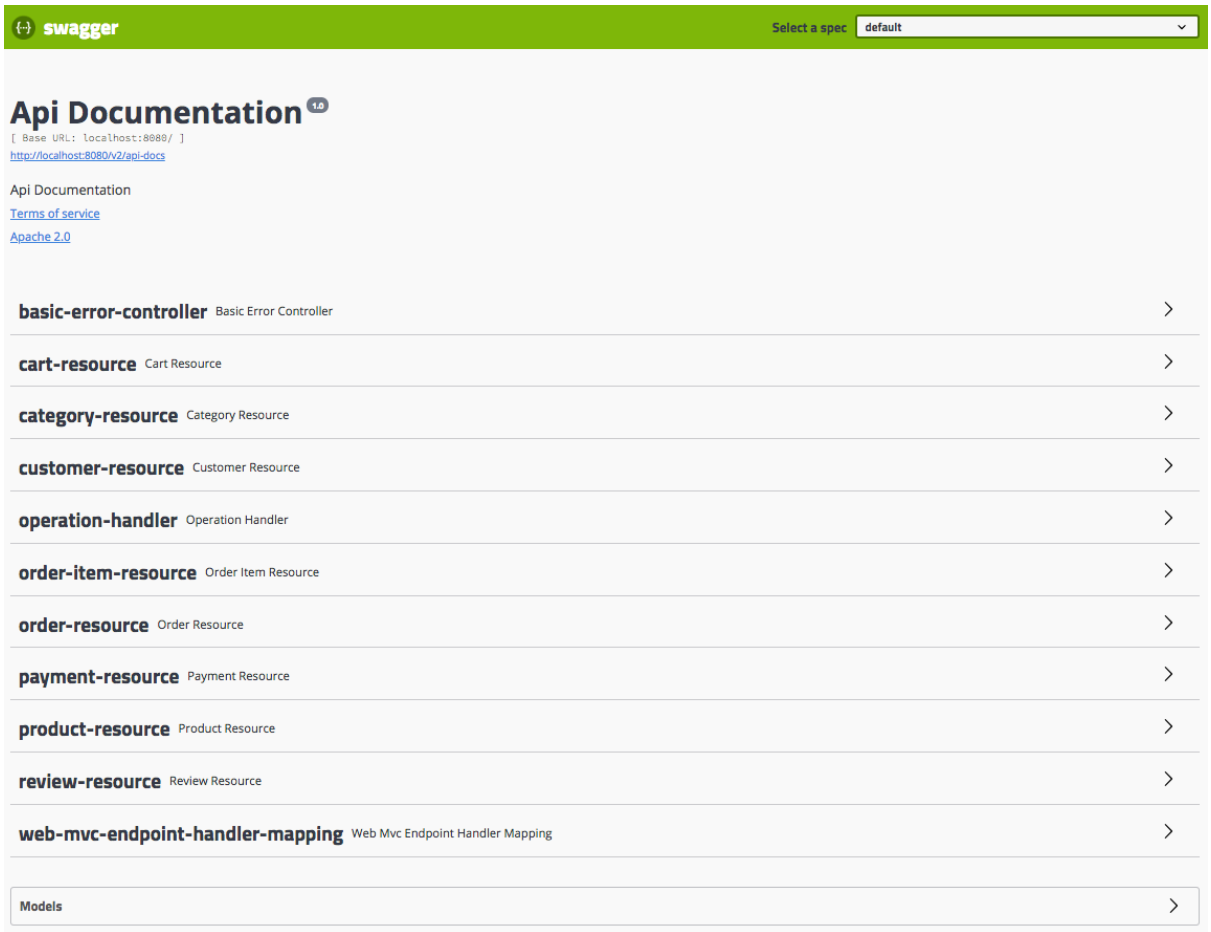
```
@EnableSwagger2
@SpringBootApplication
public class MyboutiqueApplication {

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

Hello World Swagger !

To run the Spring Boot Application, you just run: `mvn spring-boot:run`

The application will be running on the 8080 port.

We will enjoy the game now: To access the Swagger UI, just go to `http://localhost:8080/swagger-ui.html`



Swagger UI

Chapter 3 : Upgrading the Monolithic application

Refactoring the database

Now we want to move from **H2 Databse** to **PostgreSQL**. So we have to configure the application to use it by mentioning some properties like JDBC driver, url, username, password… in the `application.properties` file and to add the **PostgreSQL JDBC Driver** in the `pom.xml`.

Requirements

This part requires an installed running **PostgreSQL** instance on your machine. We are assuming that it is running on its default port 5432. Along with the port, you will need a schema name and the credentials (username & password) to access the DB.

First of all, we start by replacing the **H2 Driver** dependency by the **PostgreSQL Driver** dependency to our `pom.xml`:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Next, we do some modifications to the `application.properties`:

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:postgresql://localhost:5432/demo <1>
spring.datasource.username=developer <2>
spring.datasource.password=p4SSW0rd <2>
```

1. The Datasource URL is pointing to a local **PostgreSQL** instance, with a DB called `demo`.
2. The `username` and `password` of our **PostgreSQL** instance.

Now, if you restart your Spring Boot application, you will be using the **PostgreSQL** instance instead of the embedded **H2** instance.

Chapter 4: Building & Deploying the Monolithic application

Spring Boot makes it easy to create stand-alone Java web applications as a JAR or a WAR file. But generally, in production, a server already exists. So how do we build and deploy our apps in these cases? and how can we run our application with other already existing applications?

Building the monolith

In our stack, for our Spring Boot application, we are using Maven as a build system that supports dependency management.

Maven helps manage builds, documentation, reporting, dependencies, software configuration management (SCM), releases and distribution.

Many integrated development environments (IDEs) provide plug-ins or add-ons for Maven, thus enabling Maven to compile projects from within the IDE.

Using Maven is extremely easy, once you learn a few of the main concepts. Each project contains a file called a POM (Project Object Model), which is just an XML file containing details of the project. Some of these details might include project name, version, package type, dependencies, Maven plugins, etc.

At this stage our `pom.xml` file might look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc\
e"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0\
.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.targa.labs</groupId>
    <artifactId>myboutique</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>myboutique</name>
    <description>Demo project for Playing with Java Microservices on Kubernetes and OpenShift</descrip\
tion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.2.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
```

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>

    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger-ui</artifactId>
        <version>2.9.2</version>
    </dependency>
    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger2</artifactId>
        <version>2.9.2</version>
    </dependency>

    <dependency>
                    <groupId>com.h2database</groupId>
                    <artifactId>h2</artifactId>
                    <scope>runtime</scope>
            </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
```

```
    </build>

</project>
```

In the `Build` section, we are using the `spring-boot-maven-plugin` to build the project or even running the application using the command: `mvn spring-boot:run`.

Which package type: WAR vs JAR

We have to start by defining the difference between a WAR and a JAR. WAR stands for **W**eb Application **AR**chive and is deployed on a Servlet Container like Tomcat or Jetty. JAR stands for **J**ava **AR**chive and this will contain and embedded servlet container like Tomcat. These are two very different ways of deploying an application. The Spring Team recommends the **JAR** package type.

Josh Long, the Spring Developer Advocate at Pivotal always says: "Make JARs not WARs". This recommendation is based on many facts of the JAR package type.

Build a JAR

To build the project, just run `mvn package` from the command line, as follows:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< com.targa.labs:myboutique >-----------------------
[INFO] Building myboutique 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:3.0.1:resources (default-resources) @ myboutique ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.7.0:compile (default-compile) @ myboutique ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:3.0.1:testResources (default-testResources) @ myboutique ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Users/n.lamouchi/Desktop/myboutique/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.7.0:testCompile (default-testCompile) @ myboutique ---
[INFO] Changes detected - recompiling the module!
[INFO]
[INFO] --- maven-surefire-plugin:2.21.0:test (default-test) @ myboutique ---
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ myboutique ---
[INFO] Building jar: /Users/n.lamouchi/Desktop/myboutique/target/myboutique-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.0.2.RELEASE:repackage (default) @ myboutique ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

```
[INFO] Total time: 2.931 s
[INFO] Finished at: 2018-05-20T18:09:39+02:00
[INFO] ------------------------------------------------------------------------
```

Next, go to the `target` directory, you should see `myboutique-0.0.1-SNAPSHOT.jar`. The file should be around 44 MB in size.
You should also see a much smaller file named `myboutique-0.0.1-SNAPSHOT.jar.original` in the target directory. This is the original jar file, which has around 75 Kb, that Maven created before it was repackaged by Spring Boot.

To run that application, use the `java -jar` command, as follows:

```
$ java -jar target/myboutique-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.2.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started MyboutiqueApplication in 10.025 seconds (JVM running for 10.872)
```

To exit the application, just press `ctrl-c`.

Build WAR if you

To be able to build a WAR, we need to convert our packaging from JAR to WAR:

Step 1: Adding the Servlet Initializer Class

In our application, we are using JAR as the package type, we only need the main application class that looks like this.

```
package com.targa.labs.myboutique;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@EnableSwagger2
@SpringBootApplication
public class MyboutiqueApplication {

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

To have WAR deployment, we need to add another class that will help configure our web application. We are going to add a new class called `ServletInitializer` that looks like this:

```
package com.targa.labs.myboutique.configuration;

import com.targa.labs.myboutique.MyboutiqueApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(MyboutiqueApplication.class);
    }
}
```

We are using our main application class (`MyboutiqueApplication.class`) as an argument to the sources method.

Step 2: Exclude the embedded container from the WAR

In our `pom.xml`, in the Spring Boot Starter Web dependency, there is a Tomcat dependency declared. As the application will be deployed in a Tomcat server, we need to change that dependency to say that this will be provided for us.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
</dependency>
```

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

```
<groupId>com.targa.labs</groupId>
<artifactId>myboutique</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
```

Step 4: Package your application

With those changes in place, you can now package your application.

```
mvn package
```

If you look inside of the /target folder now you should see the war file. You can now take that file and deploy on your servlet container.

Deploying the monolith

Deploying the JAR

In additional to running Spring Boot applications by using `java -jar`, it is also possible to make fully executable applications for Unix systems. A fully executable jar can be executed like any other executable binary or it can be registered with `init.d` or `systemd`. This makes it very easy to install and manage Spring Boot applications in common production environments.

Fully executable jars work by embedding an extra script at the front of the file. Currently, some tools do not accept this format, so you may not always be able to use this technique. It is recommended that you make your jar or war fully executable only if you intend to execute it directly, rather than running it with `java -jar` or deploying it to a servlet container.

To create a *fully executable* jar with Maven, use the following plugin configuration:

```
<plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
                <executable>true</executable>
        </configuration>
</plugin>
```

You can then run your application by typing `./myboutique-0.0.1-SNAPSHOT.jar`. The directory containing the jar is used as your application's working directory.

Deploying the WAR

To have our WAR file deployed and running in **Tomcat**, we need to complete the following steps:

1. Download **Apache Tomcat** and unpackage it into a *tomcat* folder
2. Copy our WAR file from `target/myboutique.war` to the *tomcat/webapps/* folder
3. From a terminal navigate to *tomcat/bin* folder and execute
    1. **catalina.bat run** (on Windows)
    2. **catalina.sh run** (on Unix-based systems)
4. Our application will be reachable at *http://localhost:8080/myboutique/*

Continuous integration and deployment

TBD

Part Two: The Microservices Era

Chapter 5: Microservices Architecture Pattern

The Monolithic Architecture

What is a Monolithic Architecture ?

The word *monolith* was originally used by Ancient Greeks to describe a single, mountain-sized block of stone. Though the word is used more broadly today, the idea remains the same – a monolithic software product is a single, indivisible unit that will generally grow to a large size.

In a typical client-server architecture, a monolithic product lives on the server, where it handles HTTP requests, executes logic, and interacts with the database. It contains a single executable that performs all of the server-side functions for an application.

For example, in *MyBoutique* to update the behavior of a product page, a developer would access the same code base as they would to add a new customer service feature, or to change the functionality of the Cart. In a monolith, everything is managed and served from the same place. The size and simplicity of monolithic software products are both their strengths and their weaknesses.

Our application is built as a single unit, a single-codebase. This server-side application will handle HTTP requests, execute some domain specific logic, retrieve and update data from the database, and handle payloads to be sent to the REST Client. It is a monolith – a single logical executable. To make any alterations to the system, we must rebuild and redeploy an updated version of our application.

The primary benefit of a monolithic application is the simplicity of its structure. A unique simple application which is easy to deploy and scale.

To deploy a monolithic application, only one file or directory has to be handled. This makes deployment fairly straightforward. Since the entire application's codebase is in one place, only one environment has to be configured to build and deploy the software. In most cases, this means less time spent figuring out how to get deploy and deliver your application to your end users.

Monoliths are a convenient way to start a new software project with minimal concern for setup on a server or cloud environment. While the complexity may grow over time, appropriate management of the code base can help maintain productivity over the lifetime of a monolithic application.

Monolithic applications tend to become more cumbersome over time. Without close attention to how code is being written and maintained, a monolith can become dangerously brittle. This magnifies each bump in the road for your business as new challenges and demands arise for your products.

On the development side, monoliths can hinder agility. As mentioned prior, monolithic applications are very tightly coupled and can become a complex web of code as the product evolves. Thus, it can be extremely difficult for developers to manage over time.

> "The way to build a complex system that works is to build it from very simple systems that work."
>
> - Kevin Kelly

Also, it is common for each developer to understand only part of a monolith, meaning very few developers can explain the entirety of the application. This can be the case when it's a small or a medium application, but it will never will be the case when it's a huge application. Since the monoliths must be developed and deployed as one unit, it can be difficult to break up development efforts into independent teams. Each code change must be carefully coordinated, which slows down development.

This situation can be a little-bit difficult to new developers, who dont hope to deal with a massive code base that has evolved over the years. As a result, more time is spent finding the correct line of code, and making sure it doesn't have side effects. So, less time is spent writing new features that will actually improve the application.

Developers who are used to modern development environments may be disappointed with the rigidity of monoliths, which are generally confined to their original legacy stack. Adopting new technology in a monolith can mean rewriting the whole application, which is a costly and time-intensive drudgery, that doesn't always can be done.

Monoliths are popular because they are simpler to begin building than their alternative, microservices.

Microservices Architecture

What is a Microservices Architecture ?

While a monolith is a single, large unit, a microservice architecture uses small, modular units of code that can be deployed independently of the rest of a product's components.

> "Simple can be harder than complex. You have to work hard to get your thinking clean to make it simple."
>
> - Steve Jobs

Microservices architecture allow big application to divide into small, loosely coupled services. It comes up with lot of benefits:

1. We have our e-commerce application which was developed using monolithic architecture, and over the time it has grown with lot of new features. Now our codebase became huge and few set of people going to join our team. It is very difficult for them to get started with given application as there is no clear separation between various components of application. Microservices alllows us to see our application as small, loosely coupled components, anyone can get started with existing application in short span of time and, it wouldn't be too difficult to add new fix to codebase.
2. Imagine today we have the *Black Friday*, so we will get huge amount of traffic. And somehow our *Product Catalog* will be highly requested. It will affect whole application, and it will cause whole application to go down. If the same happens in microservices architecture, we won't face such failures as it runs multiple services independently so even if one goes down, it won't affect other services.
3. We are some developers who spent a lot of time in terms of understanding our application codebase. We are existed about adding new features for example products search engine. You have to redeploy whole application to show this features to end users. And we would be working on many such features, just think about the time it is going to take every time in deployment. So if our application is huge, it takes a lot of effort and time which definitely cause loss of productivity. Nobody likes to wait to see the result of code changes. In microservices architecture we will try to make codebase as small as possible in each service so it doesn't take so much time in terms of building, deployments etc.
4. Today we are getting huge amount of traffic, and nothing got broken, servers are up. Lets say we need to scale some component of our application, the *Product Catalog* for example. We can't scale individual component in monolithic architecture. But, in microservices architecture, we can scale any component separately with adding more number of nodes dynamically.
5. Tomorrow, we want to move to new framework, or technology stack. It is so hard to upgrade the monolith as it has become very big, complex over the time. It is not that difficult to achieve in microservices architecture, as components are small and flexible.

That's how microservices architecture makes our life easy. There are various strategies available which help us in diving application into small services ex. decompose by domain driven design, decompose by business capability etc.

What is really a Microservice?

A Microservice, aka service, typically implements a set of distinct features or functionality, such as order management, customer management, etc. Each microservice is a mini-application that has its own business logic along with its boundaries, such as REST webservices. Some microservices would expose an API that's consumed by other microservices or by the application's clients. Other microservices might implement a web UI.

Each functional area of the application is now implemented by its own microservice. Moreover, the web application is split into a set of smaller web applications.

Each backend service generally exposes a REST API and most services consume APIs provided by other services. The UI services invoke the other services in order to render web pages. Services might also use asynchronous message-based communication.

Some REST APIs are also exposed clients apps, that don't have direct access to the backend services. Instead, communication is mediated by an intermediary known as an API Gateway. The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring.

The Microservices Architecture pattern significantly impacts the relationship between the application and the database. Rather than sharing a single database schema with other services, each service has its own database schema. On the one hand, this approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data. However, having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling.

Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs, the so-called polyglot persistence architecture.

On the surface, the Microservices Architecture pattern is similar to SOA. With both approaches, the architecture consists of a set of services. However, one way to think about the Microservices Architecture pattern is that it's SOA without the commercialization and perceived baggage of web service specifications (WS) and an Enterprise Service Bus (ESB). Microservice-based applications favor simpler, lightweight protocols such as REST, rather than heavy protocols. They also very much avoid using ESBs and instead implement ESB-like functionality in the microservices themselves. The Microservices Architecture pattern also rejects other parts of SOA, such as the concept of a canonical schema.

Making the Switch

So is it possible to switch from a monolithic to microservices without starting from scratch? Is it worth it? Big names like Netflix, Amazon, and Twitter have all shifted from monolithic to microservices and have no intention of going back.

Changing architectures can be done in stages. Basically, extracting microservices one-by-one out of your monolithic application until your done. A good architecture choice can transform your applications and organization for the better. If you're thinking about switching over, you will get a great migration guide by the incoming chapters.

Chapter 6: Splitting the Monolith: Bombarding the domain

We've already defined what is a microservice and what is the problems solved by this architecture. We listed also many the benefits of adopting microservices architecture. But how can I migrate my monolithic application to microservices architecture? How can we apply this pattern? How can we split our monolith withou rewriting all the application?

We will use Domain-Driven Design as an approach to split our monolith. Domain-Driven Design (DDD), is an approach to software development that simplifies the software modeling and design.

What is Domain-Driven Design ?

The Domain-Driven Design has many advantages:

- Focus on the core business domain and business logic.
- Best way to ensure that the design is based on a model of the domain.
- A tight collaboration way between technical and business teams.

To understand Domain-Driven Design, we need to explain and define many concepts.

Context

The circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood.

Domain

The subject area to which the user applies a program is the domain of the software.

Model

A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

Ubiquitous Language

A common language to communicate within a project. As we have seen, designing a model is the collective effort of software designers, domain experts, and developers; therefore, it requires a common language to communicate with. DDD makes it necessary to use ubiquitous language. Domain models use ubiquitous language in their diagrams, descriptions, presentations, speeches, and meetings. It removes the misunderstanding, misinterpretation, and communication gap among them. Therefore, it must be included in all diagrams, description, presentations, meetings, and so on—in short, in everything.

Strategic Design

Domain-Driven Design addresses the strategy behind the direction of the business and not just the technical aspects. It helps define the internal relationships and early warning feedback systems. On the technical side, strategic design protects each business service by providing the motivation for how an service-oriented architecture should be achieved.

There are various principles that could be followed to maintain the integrity of the domain model, and these are listed as follows:

Bounded context

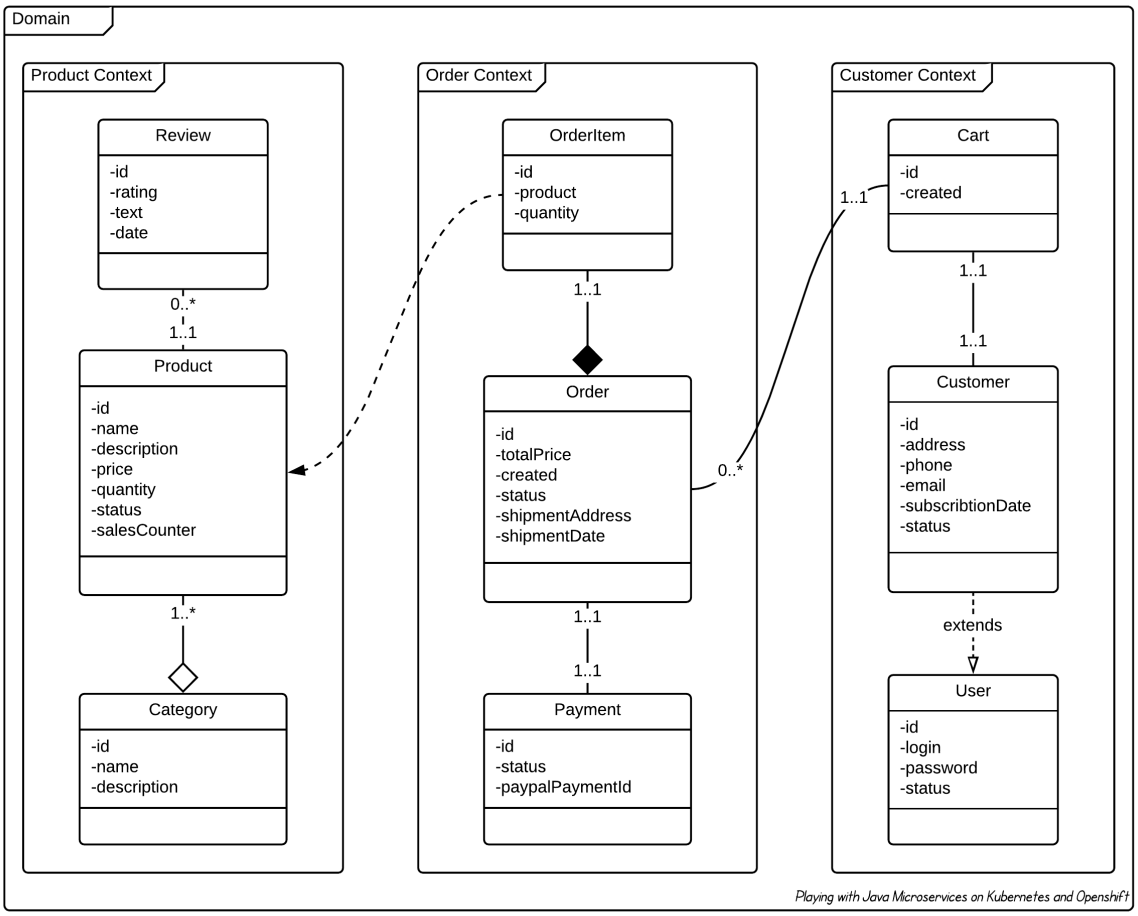
When you have different submodels, it is difficult to maintain the code when all submodels are combined. You need to have a small model that can be assigned to a single team. You might need to collect the related elements and group them. Context keeps and maintains the meaning of the domain term defined for its respective submodel by applying this set of conditions.

These domain terms define the scope of the model that creates the boundaries of the context.

Bounded context seems very similar to the module that you learned about in the previous section. In fact, the module is part of the bounded context that defines the logical frame where a submodel takes place and is developed. Whereas, the module organizes the elements of the domain model, and is visible in the design document and the code.

Now, let's examine the table reservation example we've been using. When you started designing the system, you would have seen that the guest would visit the application, and would request a table reservation at a selected restaurant, date, and time. Then, there is the backend system that informs the restaurant about the booking information, and similarly, the restaurant would keep their system updated in regard to table bookings, given that tables can also be booked by the restaurant themselves. So, when you look at the system's finer points, you can see three domain models:

- The Product domain model
- The Order domain model
- The Customer domain model



MyBoutique Domain Models

They have their own bounded context and you need to make sure that the interface between them works fine.

Bombarding La Boutique

Splitting our application will be done in many steps.

Codebase

We will start by creating packages for every bounded context, and then move related classes into them. With NetBeans (and other IDEs) we can move and refactor code easily with some clicks. While moving code, you can find some new bounded context that you may missed.

At this point, the only way to guarantee that our refactoring did not break our application, is **TESTS !!**

Doing the refactor and moving the code in a small codebase, like our application, is very easy and will take some minutes or even hours. But when dealing with huge applications will be hard and will take severals weeks or months. You may not need to sort all code into domain-oriented

packages before splitting out your first service, and indeed it can be more valuable to concentrate your effort in one place. There is no need for this to be a big-bang approach. It is something that can be done bit by bit, day by day, and we have a lot of tools at our disposal to track our progress.

We can use structure analysis tools like Stan4j and Structure101.

Dependencies and Commons

While we are moving parts of code to the corresponding package, we will meet some common classes (DTOs or Utilities).. these commonly used classes must be moved to a dedicated **COMMONS** package.

The structure analysis tools are very useful at this point.

Next, we will be talking about the most important element of the Domain Splitting: Entities and Relationships

Entities

Until now, we had grouped our code into packages representing our bounded contexts; we want to do the same for our database access code.

For our database access code, we have already a **Repository** for each **Entity**.
The relationships between entities shows us the foreign key relationships between table, which is database-level constraints.

All this helps you understand the coupling between tables that may span what will eventually become service boundaries. But how do we cut those ties? And what about cases where the same tables are used from multiple different bounded contexts? Handling problems like these is not easy, and there are many answers, but it is doable.

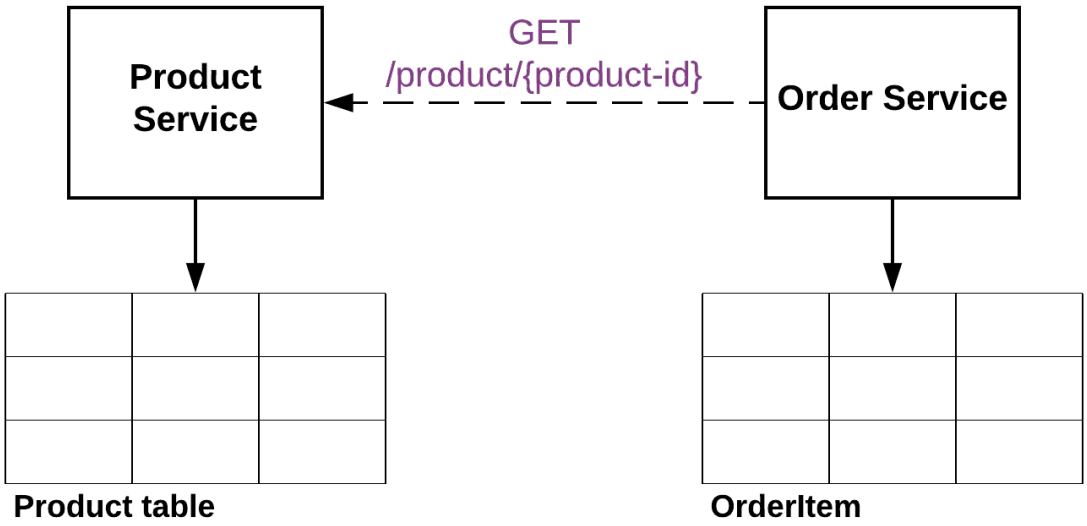
Coming back to some concrete examples, let's consider our Boutique again. We have identified three bounded contexts, and want to move forward with making them three distinct, interacting services. We are going to look at the problems we might face, and what will be the potential solutions.

Example: Breaking Foreign Key Relationships

In our application, we are storing the products information in a dedicated table, which is mapped and managed by the ORM.
The OrderItem object stores in his dedicated table, the reference of the ordered product with the ordered quantity. The reference of the Product record in the OrderItem table consists a forein key constraint.

So how can we break this relationship? Well, we need to make a change in two places. First, we need to stop the Order code from reaching into the Product table, as this table really belongs to the Product Bounded Context, and we don't want database integration happening once Product and Order are services in their own rights. The quickest way to address this is rather than having

the code in Order reach into the Product table, we'll expose the data via an API call in the Product service that the Order code can call. This API call will be the forerunner of a call we will make over the wire, as we see in Figure 5-3.



Post removal of the foreign key relationship

Figure 5-3. Post removal of the foreign key relationship

We have two database calls instead of one, as these two services are separated, while we have only one call in the monolith. So for sure you are thinking now about performance issues?! What I can told you, that the extra call will take some extra time, but will not so huge. You can test performance before splitting, then you should have visibility while making changes.

But what about the foreign key relationship? Well, we lose this altogether. This becomes a constraint we need to now manage in our resulting services rather than in the database level. This may mean that we need to implement our own consistency check across services, or else trigger actions to clean up related data. Whether or not this is needed is often not a technologist's choice to make. For example, if our Order service contains a list of IDs for Products, what happens if a Product is removed and an Order now refers to an invalid Product ID? Should we allow it? If we do, then how is this represented in the order when it is displayed? If we don't, then how can we check that this isn't violated? These are questions you'll need to get answered by the people who define how your system should behave for its users.

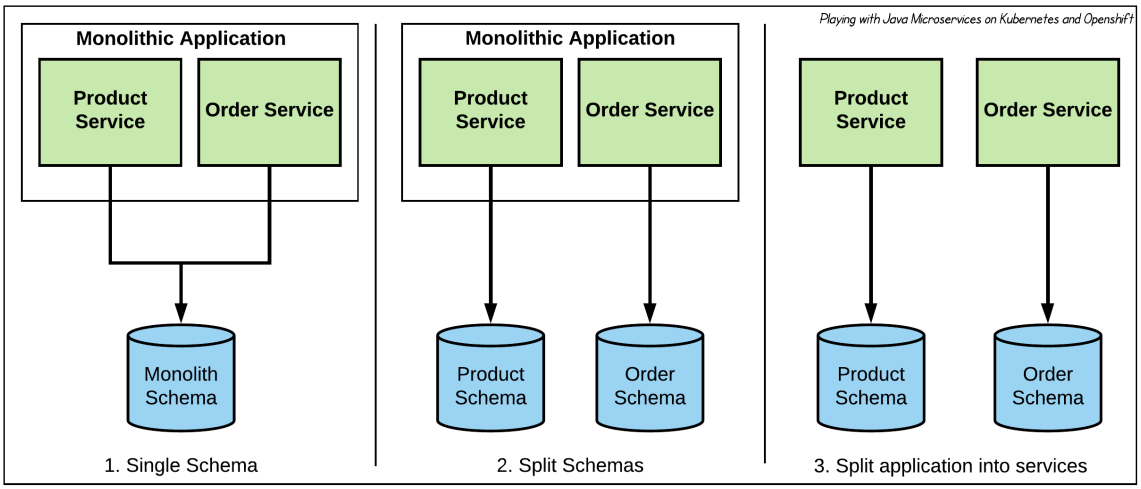
Refactoring Databases

Staging the Break

We used the defined bounded contexts to define the splitting plans of the database.

As any refactoring must be in steps, no big-bang effect will happen in our migration from a monolithic application with one database schema to many services having its own schema each.

We recommend that you split out the schema but keep the service together before splitting the application code out into separate microservices.



Staging a service separation

Figure 5-9. Staging a service separation

With a separate schema, we'll be potentially increasing the number of database calls to perform a single action. Where before we might have been able to have all the data we wanted in a single *SELECT* statement, now we may need to pull the data back from two locations and join in memory. Also, we end up breaking transactional integrity when we move to two schemas, which could have significant impact on our applications; we'll be discussing this next. By splitting the schemas out but keeping the application code together, we give ourselves the ability to revert our changes or continue to tweak things without impacting any consumers of our service. Once we are satisfied that the DB separation makes sense, we can then think about splitting out the application code into two services.

Transactional Boundaries

Transactions are useful things. They allow us to say *these events either all happen together, or none of them happen*. They are very useful when we're inserting data into a database; they let us update multiple tables at once, knowing that if anything fails, everything gets rolled back, ensuring our data doesn't get into an inconsistent state. Simply put, a transaction allows us to group together multiple different activities that take our system from one consistent state to another—everything works, or nothing changes.

Transactions don't just apply to databases, although we most often use them in that context. Message brokers, for example, have long allowed you to post and receive messages within transactions too.

With a monolithic schema, all our create or updates will probably be done within a single transactional boundary. When we split apart our databases, we lose the safety afforded to us by having a single transaction.

Within a single transaction in our existing monolithic schema, inserting records happens within a single transaction.

But if we have pulled apart the schema into two separate schemas, we have lost this transactional safety. We now spans many separate transactional boundaries. If one insertion fails, we can clearly stop everything, leaving us in a consistent state. But what happens when the insert into the first table works, but the insert into the second one fails?

Try Again Later

The fact that the insertion was done in the first table might be enough for us, and we may decide to retry the insertion into the second table at a later date. We could queue up this part of the operation in a queue or logfile, and try again later. For some sorts of operations this makes sense, but we have to assume that a retry would fix it.

In many ways, this is another form of what is called eventual consistency. Rather than using a transactional boundary to ensure that the system is in a consistent state when the transaction completes, instead we accept that the system will get itself into a consistent state at some point in the future. This approach is especially useful with business operations that might be long-lived.

Abort the Entire Operation

Another option is to reject the entire operation. In this case, we have to put the system back into a consistent state. The second table is easy, as that insert failed, but we have a committed transaction in the first table. We need to unwind this. What we have to do is issue a compensating transaction, kicking off a new transaction to wind back what just happened. For us, that could be something as simple as issuing a *DELETE* statement to remove the records inserted in the first table. Then we'd also need to report back via the UI that the operation failed. Our application could handle both aspects within a monolithic system, but we'd have to consider what we could do when we split up the application code. Does the logic to handle the compensating transaction live in the customer service, the order service, or somewhere else?

But what happens if our compensating transaction fails? It's certainly possible. In this situation, you'd either need to retry the compensating transaction, or allow some backend process to clean up the inconsistency later on. This could be something as simple as a maintenance screen that admin staff had access to, or an automated process.

Now think about what happens if we have not one or two operations we want to be consistent, but three, four, or five. Handling compensating transactions for each failure mode becomes quite challenging to comprehend, let alone implement.

Distributed Transactions

An alternative to manually orchestrating compensating transactions is to use a distributed transaction. Distributed transactions try to span multiple transactions within them, using some overall governing process called a transaction manager to orchestrate the various transactions being done by underlying systems. Just as with a normal transaction, a distributed transaction

tries to ensure that everything remains in a consistent state, only in this case it tries to do so across multiple different systems running in different processes, often communicating across network boundaries.

The most common algorithm for handling distributed transactions—especially short-lived transactions, as in the case of handling our customer order—is to use a two-phase commit. With a two-phase commit, first comes the voting phase. This is where each participant (also called a cohort in this context) in the distributed transaction tells the transaction manager whether it thinks its local transaction can go ahead. If the transaction manager gets a yes vote from all participants, then it tells them all to go ahead and perform their commits. A single no vote is enough for the transaction manager to send out a rollback to all parties.

This approach relies on all parties halting until the central coordinating process tells them to proceed. This means we are vulnerable to outages. If the transaction manager goes down, the pending transactions never complete. If a cohort fails to respond during voting, everything blocks. And there is also the case of what happens if a commit fails after voting. There is an assumption implicit in this algorithm that this cannot happen: if a cohort says yes during the voting period, then we have to assume it will commit. Cohorts need a way of making this commit work at some point. This means this algorithm isn't foolproof—rather, it just tries to catch most failure cases.

This coordination process also mean locks; that is, pending transactions can hold locks on resources. Locks on resources can lead to contention, making scaling systems much more difficult, especially in the context of distributed systems.

Distributed transactions have been implemented for specific technology stacks, such as Java's Transaction API, allowing for disparate resources like a database and a message queue to all participate in the same, overarching transaction. The various algorithms are hard to get right, so I'd suggest you avoid trying to create your own. Instead, do lots of research on this topic if this seems like the route you want to take, and see if you can use an existing implementation.

So What to Do?

All of these solutions add complexity. As you can see, distributed transactions are hard to get right and can actually inhibit scaling. Systems that eventually converge through compensating retry logic can be harder to reason about, and may need other compensating behavior to fix up inconsistencies in data.

When you encounter business operations that currently occur within a single transaction, ask yourself if they really need to. Can they happen in different, local transactions, and rely on the concept of eventual consistency? These systems are much easier to build and scale (we'll discuss this more in Chapter 11).

If you do encounter state that really, really wants to be kept consistent, do everything you can to avoid splitting it up in the first place. Try really hard. If you really need to go ahead with the split, think about moving from a purely technical view of the process (e.g., a database transaction) and actually create a concrete concept to represent the transaction itself. This gives you a handle, or a hook, on which to run other operations like compensating transactions, and a way to monitor and manage these more complex concepts in your system. For example, you might create the

idea of an “in-process-order” that gives you a natural place to focus all logic around processing the order end to end (and dealing with exceptions).

Summary

We decompose our system by focusing on **Bounded Contexts**, and this can be an incremental approach. By getting good at finding these boundaries and working to reduce the cost of splitting out services in the first place, we can continue to grow and evolve our systems to meet whatever requirements come down the road. As you can see, some of this work can be so hard. But the very fact that it can be done incrementally means there is no need to fear this work.

So now we can split our services out, but we’ve introduced some new problems too. We have many more moving parts to get into production now! So next up we’ll dive into the world of deployment.

Chapter 7: Applying DDD to the code

Introduction

We saw, in the previous chapter, how to use Domain Driven Design to split our Domain into **Bounded Contexts**.

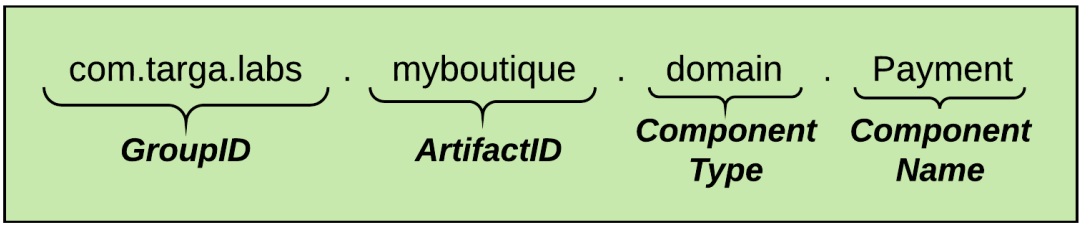
Now we will apply the splitting of our code, using the cut lines that we defined in the previous chapter.

Applying Bounded Contexts to Java Packages

Our classes are already classified by the **component type**, such as Repository, Service, etc..

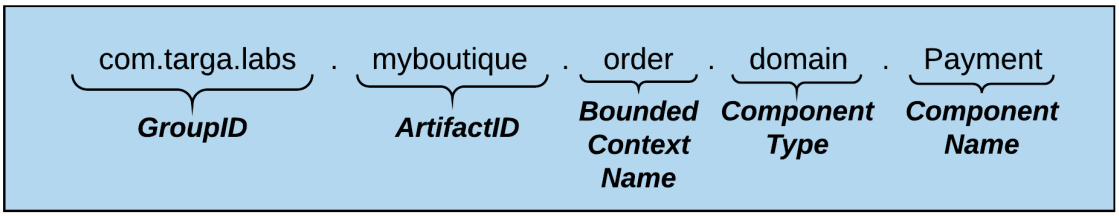
We will move every component in a new package, that includes the **Bounded Context** name.

The name of component is actually has the format:



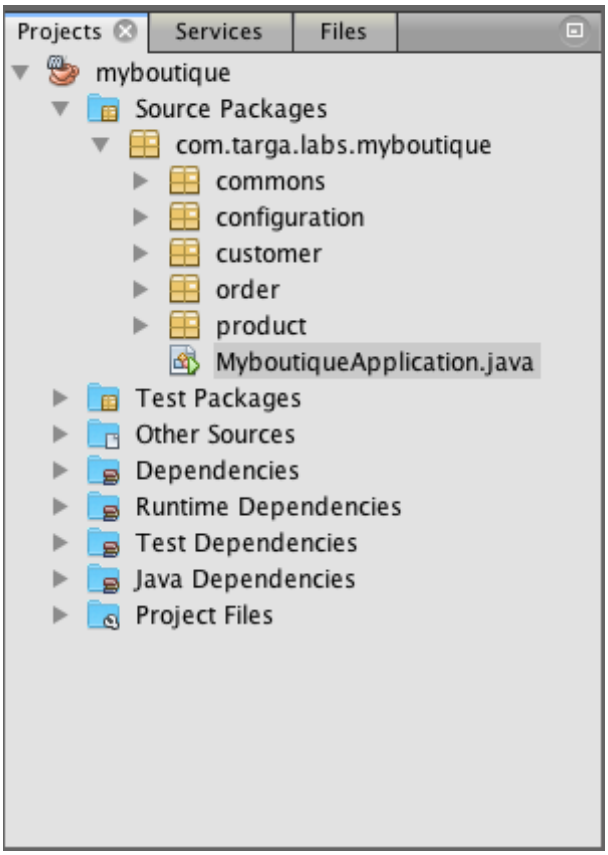
Actual name format of a component

So, after the refactoring, the name will be:



New name format of a component

After renaming all the components, we will get a Project Tree looking like this:

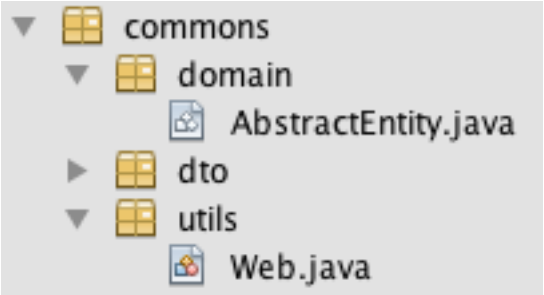


Project structure with Bounded Contexts as packages

Wait ! Wait ! What is the content of that `commons` and `configuration` packages ?

The birth of the commons package

I didn't tell you ? Euh, ok :) I'm sure that we said that the common components have to be collected into a dedicated `commons` package, that will be shared by all the **BC**s. Ok, I will show you what I have in my `commons` package:



Commons package content

Ohhhh ! What is this `dto` package? We already said that components have to be moved to their corresponding **Bounded Context**?
So if the `Payment` class belongs exclusively to the `Order` **BC**, then automatically the `PaymentDTO` class has to reside in the `order` package and not the `commons`. So why our `DTO`s are not respecting this recommandation? Are we kidding ?

The answer is obviously NO! We are not kidding. I did this **intentionally** and not by mistake. The reason is very simple:
When we was talking about microservices, we said that after the splitting, our microservices will be talking with each other. So, let we imagine the case: The `Order` wants to get a `Product` from the `Product` microservice using a given Product ID. The REST API in the `Product` will return a `ProductDTO` object populated by the data of a `Product` record, which will be serialized to JSON and will be returned back to the requesting service, which is the `Order`. But how can handle the received JSON if its format is unknown to him ? The answer is *it can't*, when the `ProductDTO` is exclusively stored in the `Product` microservice.

An other idea, is to store the `DTO` classes in both services, but this idea seems to unefficient for me, as it will result of duplicated code to handle. Imagine the case that we need to update the `DTO` classes, in this case we will be doing the same work twice..

So the best idea that I found is to store the `DTO`s in the `commons`. This solution will make sharing and upgrades so easy.

Other than the `dto` package, we find also a `domain` package.

What ? a Domain in the common area?

No, it's not really a domain, it has only **AbstractEntity** class which a common class, used a parent class for all the `Entity` classes in every **BC**. As the entities belong to the Domain, I thought that the most suitable place to put the common **AbstractEntity** is the `commons.domain` package.

Finally, there is a `utils` package, that will host the **Utility** classes, for example we have actually a `Class` that contains **constants** which looks like this:

```
public class Web {
    public static final String API = "/api";
}
```

The birth of the configuration package

Our microservices will be a standalone **Spring Boot** applications. These applications has many non-functional requirements like activating the **Swagger** console, logging, etc.. These requirements will be filled by components that we will store in the `configuration` package. So ours will have until now only the **SwaggerConfiguration** class.

Locating & breaking the BC Relationships

Locating the BC Relationships

This step aims to break dependencies between **Bounded Context**s. To be able to break them, we need first to find these dependencies. There are many ways to find them. For example reviewing the *Class Diagrams*, the source code, etc.. In the previous chapter we talked about tools that can help us highlight the dependencies between blocs in our monolith.

I am mainly using **STAN** (Aka STAN4J), a powerful structure analysis for Java. **STAN** supports a set of carefully selected metrics, suitable to cover the most important aspects of structural quality. Special focus has been set on visual dependency analysis, a key to structure analysis.

STAN is available in two variants:

1. as a standalone application for Windows and Mac OS, which is targeted to architects and project managers who are typically not using the IDE
2. as an extension to the Eclipse Integrated Development Environment (IDE), that allows the developer to quickly explore the structure for any bunch of code she desires.

Yeah! You are right! We will be using the second choice: I got a fresh install of Eclipse IDE and I installed the plugin as described in http://stan4j.com/download/ide/[4].

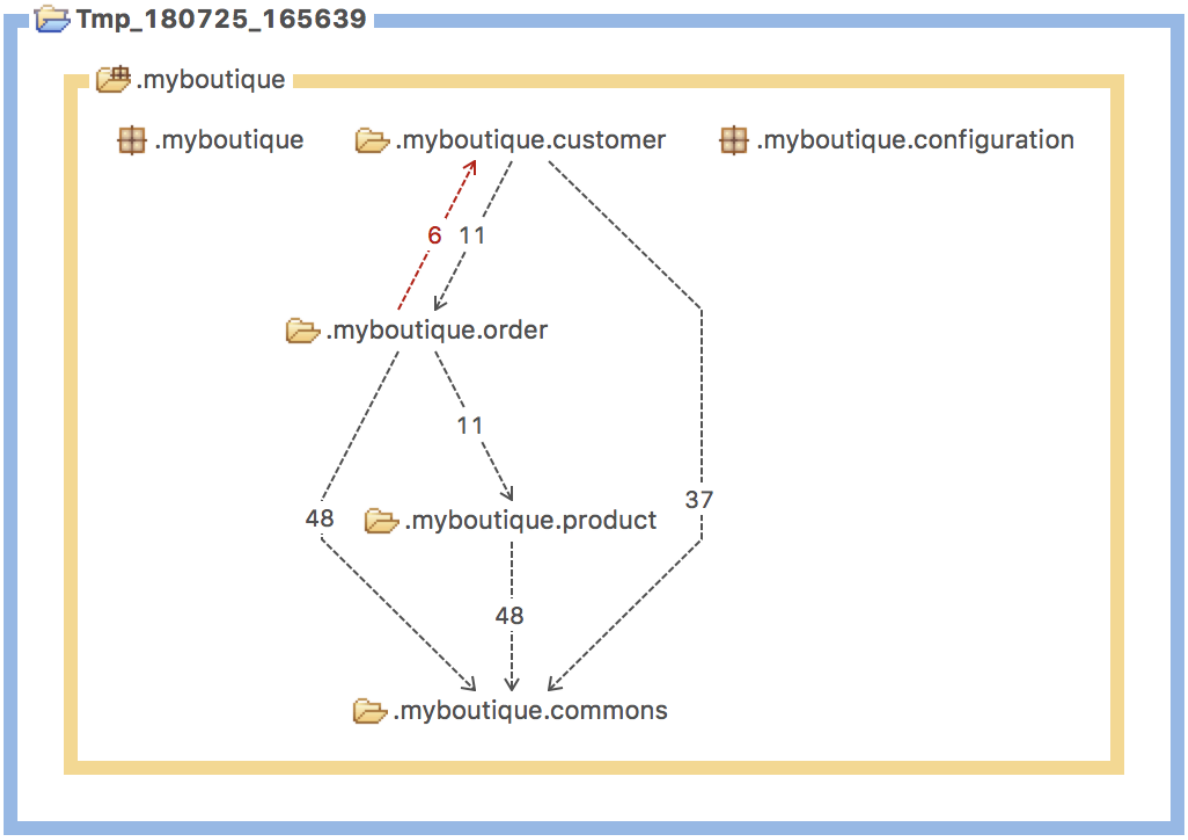
Our monolith is a Maven-based project, so it can be easily imported into Eclipse IDE. After doing the import, just do a :

1. Right click on the project
2. Run As > Maven Build
3. Choose the Maven configuration, if you already have one
4. In Goals, just enter: `clean install -DskipTests` and Run
5. Done !

Next, to make a structure analysis of our project:

[4] http://stan4j.com/download/ide/

1. Right click on the project
2. Run As > Structure Analysis
3. Yoppi ! The structure diagram is here !



First structure analysis with STAN

Breaking the BC Relationships

Now, we will break the relationship between the Bounded Contexts. We can start by the relations between entitites. Generally the relations are more effective between tables in the database. Here the entity classes are treated as relational tables (concept of JPA), therefore the relationships between Entity classes are as follows:

- @ManyToOne Relation
- @OneToMany Relation
- @OneToOne Relation
- @ManyToMany Relation

To be more precise, we will not break relationships between entities that belong to the same **BC**, we will break inter-**BC** relationships. For example we have the relationship between the `OrderItem`, which belongs to the `Order Context`, and the `Product`, which belongs to the `Product Context`.

In Java, this relationship is represented by:

```
public class OrderItem extends AbstractEntity {

    @NotNull
    @Column(name = "quantity", nullable = false)
    private Long quantity;

    @ManyToOne
    private Product product;

    @ManyToOne
    private Order order;

}
```

The `@ManyToOne` annotating the `product` field is our target here.

How to break the relationship ? It's simple, the bloc :

```
@ManyToOne
private Product product;
```

Will be changed by :

```
private Long productId;
```

Ok, why we picked up the **Long** type to replace the **Product** type ? It's simple ! **Long** is the type of the **Product**'s ID.

Great ! We have to do the same in the `Product` class, if the relationship between `OrderItem` and `Product` is a bidirectional relationship.

Ok :) When we try to build the project, using `mvn clean install`, we will get a Compilation problems. This is obvious, as we edited our `OrderItem`, many components using this class has to be aware about these modifications. Here, it is the `OrderItemService` that is making trouble.

Let's see the blocks of the `OrderItemService` class that have errors, the first one:

```
public OrderItemDto create(OrderItemDto orderItemDto) {
    log.debug("Request to create OrderItem : {}", orderItemDto);

    Order order = this.orderRepository
                        .findById(orderItemDto.getOrderId())
                        .orElseThrow(() -> new IllegalStateException("The Order does not exist!"));

    Product product = this.productRepository
                        .findById(orderItemDto.getProductId())
                        .orElseThrow(() -> new IllegalStateException("The Product does not exist!"));
    return mapToDto(
        this.orderItemRepository.save(
            new OrderItem(
                    orderItemDto.getQuantity(),
                    product,
                    order
            )));
}
```

Here we see that we are getting a `Product` instance from the `ProductRepository`, and we are passing it to the `OrderItem` constructor.
As we have no more the `Product` field in the `OrderItem` class, and it has been changed to *Product ID*, we need to pass the ID to the `OrderItem` constructor, instead of the `Product`.

But we already said that we will have no more the `Product` object in the `Order` **Bounded Context**. How can we talk about dealing with `Product` and `ProductRepository` classes until now?

Did you forget? We have in the scope of `Order` **Bounded Context**, the `commons` module, which contains the `ProductDTO` and we can use for free :D

For the `ProductRepository`, it's obvious, it will be replaced by a REST Client that will gather the `Product` data from the `Product` microservice, and populate the known `ProductDTO` object when needed. But here, we need only the *Product ID* to create the `OrderItem` instance. So, in this case, we dont need to get the Product record.

The second block is the method that maps the `OrderItem` to an `OrderItemDto`:

```
public static OrderItemDto mapToDto(OrderItem orderItem) {
    if (orderItem != null) {
        return new OrderItemDto(
                orderItem.getId(),
                orderItem.getQuantity(),
                orderItem.getProduct().getId(),
                orderItem.getOrder().getId()
        );
    }
    return null;
}
```

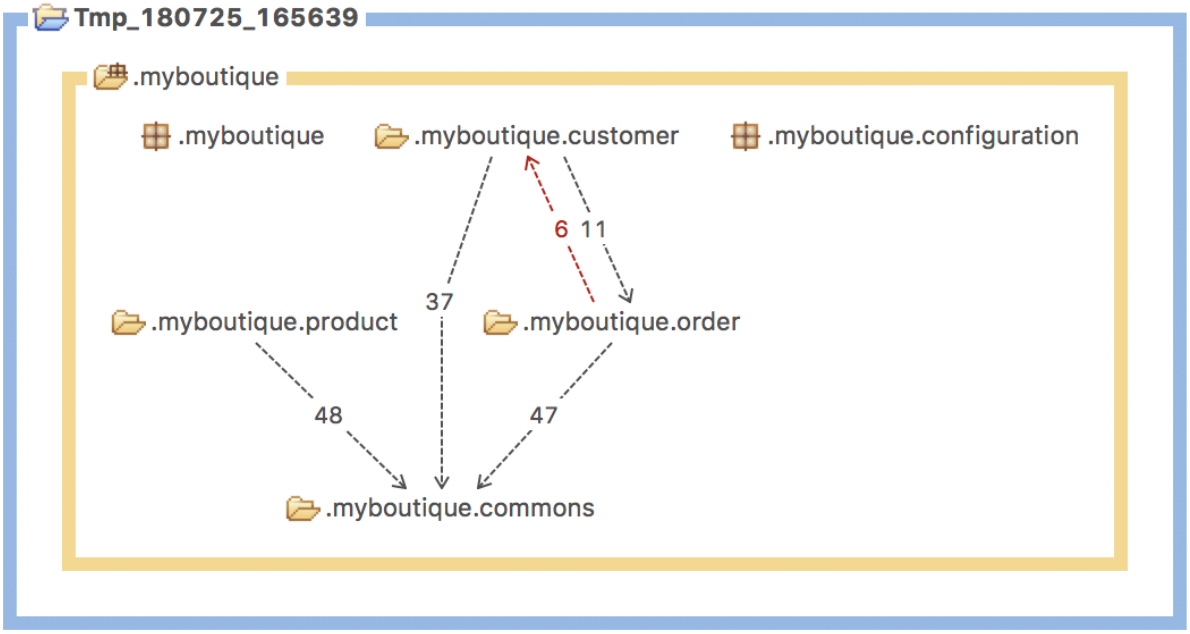
The reference of the `Product` has to be deleted from this method, as it was removed from the `OrderItem` class. We dont have no more a product as member, but we have the Product ID in the `OrderItem` class. So the line `orderItem.getProduct().getId()` has to be changed to `orderItem.getProductId()`.

The resulting `OrderItemService` will look like:

```
@RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class OrderItemService {

    private final OrderItemRepository orderItemRepository;
    private final OrderRepository orderRepository;

    public List<OrderItemDto> findAll() {
        log.debug("Request to get all OrderItems");
        return this.orderItemRepository.findAll()
                .stream()
                .map(OrderItemService::mapToDto)
                .collect(Collectors.toList());
    }
```

```
    @Transactional(readOnly = true)
    public OrderItemDto findById(Long id) {
        log.debug("Request to get OrderItem : {}", id);
        return this.orderItemRepository
                    .findById(id).map(OrderItemService::mapToDto).orElse(null);
    }

    public OrderItemDto create(OrderItemDto orderItemDto) {
        log.debug("Request to create OrderItem : {}", orderItemDto);
        Order order = this.orderRepository
                            .findById(orderItemDto.getOrderId())
                            .orElseThrow(() -> new IllegalStateException("The Order does not exist!"));

        return mapToDto(
                this.orderItemRepository.save(
                        new OrderItem(
                                orderItemDto.getQuantity(),
                                orderItemDto.getProductId(),
                                order
                        )));
    }

    public void delete(Long id) {
        log.debug("Request to delete OrderItem : {}", id);
        this.orderItemRepository.deleteById(id);
    }

    public static OrderItemDto mapToDto(OrderItem orderItem) {
        if (orderItem != null) {
            return new OrderItemDto(
                    orderItem.getId(),
                    orderItem.getQuantity(),
                    orderItem.getProductId(),
                    orderItem.getOrder().getId()
            );
        }
        return null;
    }

}
```

After this refactoring, we can use our great tool **STAN** to check how our modifications changed the structure of our project:



Project structure after breaking the relationship between Order and Product

In the generated schema, you can see that there is no more link between the **product** and **order** packages. Yoppi !!

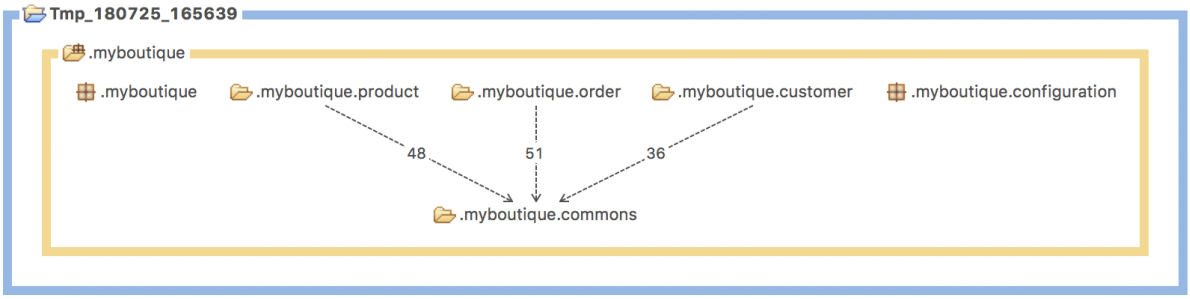
In same schema, we can see that there is a tight relationship between the **customer** and **order** packages. Let's do the same job in both of them.

After some checks, the modifications to do are:

1. in the **customer** package:
    a. Cart: the `private Order order;` will be changed by `private Long orderId;`
    b. CartService: the `Order` reference will be changed by `OrderDto`
2. in the **order** package:
    a. Order: the `private Cart cart;` will be changed by `private Long cartId;`
    b. OrderService: the `Cart` reference will be changed by `CartDto`

After these modifications, we will still have an `OrderService` reference in the `CartService`. As we said before, this `OrderService` will be replaced by a REST Client that will make calls to the API exposed by the `Order` microservice. You can comment the lines referencing the `OrderService` before analyzing the structure.

After this refactoring, we can check how our modifications changed the structure of our project:



Project structure after the full break of the relationships between BCs

This is good ! We just finished one of the heaviest tasks in our migration trip !

Now, we can start building our standalone microservices. But, we need to learn more about the best practices and microservices patterns.

Chapter 8: Meeting the microservices concerns and patterns

Introduction

When we deal with architecture and design, we immediately start thinking about recipes and patterns. These design patterns are useful for building reliable, scalable, secure applications in the cloud.

A pattern is a reusable solution to a problem that occurs in particular context. It's an idea, which has its origins in real-world architecture, that that has proven to be useful in software architecture and design.

The Microservices architecture is being proclaimed as the most powerful architectural pattern. We already discussed this pattern in details in Chapter 7.

When presenting a pattern, we will start by defining the context and the problem that we can face, and the solution given by the pattern.

Cloud Patterns

We will deal with these patterns:

- Externalized configuration
- Service discovery and registration
- Circuit Breaker
- Database per service
- API gateway
- CQRS
- Event sourcing
- Log aggregation
- Distributed tracing
- Audit logging
- Application metrics
- Health check API

Service discovery and registration

Context and problem

In the microservices world, Service Registry and Discovery plays an important role because we most likely run multiple instances of services and we need a mechanism to call other services without hardcoding their hostnames or port numbers. In addition to that, in Cloud environments service instances may come up and go down anytime. So we need some automatic service registration and discovery mechanism.

In a monolithic application, calls between components are made through language-level calls. But, in microservices architecture, services typically need to call each another using HTTP/REST (or other) calls. In order to make a request, a service needs to know the network location (IP address and port) of a given service instance. As we said in previous chapters, microservices have dynamically assigned network locations, due to many factors such as, different deployment frequencies for example. Moreover, a service can have more than one instance that can keep change dynamically because of autoscaling, failures, etc..

So, we must implement a mechanism that enables the clients of a given service to make requests to a dynamically changing set of the service instances.

How does the client of a service discover the location of a service instance?
How does the client of a service know about the available instances of a service?

Solution

We can create a service registry, which is a database of available service instances. Which works like this:

- The network location of a microservice instance is registered with the service registry when the instance starts up.
- The network location of a microservice instance is removed from the service registry when the instance terminates.
- The microservice instance availability is typically refreshed periodically using a heartbeat mechanism.

We have two varieties of this pattern:

- **Client-side Discovery pattern**: The requesting service (client) is responsible for seeking the network locations of available service instances. The client queries a service registry and then selects, using some **load-balancing** algorithms, one of the available service instances and makes a request. This mecanism follows the **client-side discovery pattern**.
- **Server-side Discovery pattern**: The **server-side discovery pattern** advises that the client makes a request to a service via a **load balancer**. It's the responsability of the load balancer to query the service registry and to forward each request to an available service instance. So, in case we want to follow this pattern, we need to have (or implement) the discussed **load balancer**.

Externalized configuration

Context and problem

An application typically uses one or more infrastructure (a message broker and a database server, etc..) and 3rd party services (a payment gateway, email and messaging, etc..). These services need configuration information (credentials for example) to be used. This configuration information is stored in files that are deployed with the application.

In some cases, it's possible to edit these files to change the application behavior after it's been deployed. However, changes to the configuration require the application be redeployed, often resulting in unacceptable downtime and other administrative overhead.

Local configuration files also limit the configuration to a single application, but sometimes it would be useful to share configuration settings across multiple applications. Examples include database connection strings or the URLs of queues and storage used by a related set of applications.

Our monolith is divised into many microservices. All these microservices need the configuration information that was provided to the monolith. Imagine that we need to update the database url. This task needs to be done for all the microservices. If we forget to update the data somewhere, it can result in instances using different configuration settings while the update is being deployed.

Solution

Externalize all application configuration including the database credentials and network location. We can for example store the configuration information externally, and provide an interface that can be used to quickly and efficiently read and update configuration settings. We can call this configuration store as **Configuration Server**.

When a microservice starts up, it reads the configuration from the given **Configuration Server**.

Circuit Breaker

Context and problem

Microservices are communicating using mainly HTTP REST requests. When a microservice synchronously another one, there is always the risk that the other service is unavailable or unreachable high latency it is essentially unusable. The unsuccessful calls might lead to resource exhaustion, which would make the calling service unable to handle other requests. The failure of one service can potentially cascade to other services throughout the application.

Solution

A requesting microservice should invoke a remote service via a proxy that works in a similar mechanism to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

The Circuit Breaker pattern, popularized by **Michael Nygard** in his book, **Release It!**, can prevent an application from repeatedly trying to execute an operation that's likely to fail. Allowing it to

continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long lasting. The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.

Database per service

Context and problem

In the microservices architecture world, the services must be loosely coupled so that they can be developed, deployed and scaled independently.

Most services need to persist data in some kind of database. In our application, , the **Order Service** stores information about orders and the **Customer Service** stores information about customers.

What's the database architecture in a microservices application?

Solution

Keep each microservice's persistent data private to that service and accessible only via its API.

The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

There are a few different ways to keep a service's persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:

- Private-tables-per-service – each service owns a set of tables that must only be accessed by that service
- Schema-per-service – each service has a database schema that's private to that service
- Database-server-per-service – each service has it's own database server.

Private-tables-per-service and schema-per-service have the lowest overhead. Using a schema per service is appealing since it makes ownership clearer. Some high throughput services might need their own database server.

It is a good idea to create barriers that enforce this modularity. You could, for example, assign a different database user id to each service and use a database access control mechanism such as grants. Without some kind of barrier to enforce encapsulation, developers will always be tempted to bypass a service's API and access it's data directly.

API gateway

Context and problem

For our boutique, imagine that we are implementing the product details page. Let's imagine that we need to develop multiple versions of ther product details user interface:

- HTML5/JavaScript-based UI for desktop and mobile browsers - HTML is generated by a server-side web application
- Native Android and iPhone clients - these clients interact with the server via REST APIs

In addition, **MyBboutique** must expose product details via a REST API for use by 3rd party applications.

A product details UI can display a lot of information about a product. For example:

- Basic information about the product such as name, description, price, etc..
- Your purchase history for the product
- Availability
- Buying options
- Other items that are frequently bought with this product
- Other items bought by customers who bought this product
- Customer reviews

Since MyBoutique follows the Microservice architecture pattern, the product details data is spread over multiple services:

- Product Service - basic information about the product such as name, description, price, customer reviews, product availability..
- Order service - purchase history for product..
- Customer service - customers, carts..

Consequently, the code that displays the product details needs to fetch information from all of these services.

How do the clients of a Microservices-based application access the individual services?

Solution

Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.

Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. The API gateway might also implement security, e.g. verify that the client is authorized to perform the request.

CQRS

Context and problem

In traditional data management systems, both commands (updates to the data) and queries (requests for data) are executed against the same set of entities in a single data repository. These entities can be a subset of the rows in one or more tables in a relational database such as SQL Server.

Typically in these systems, all create, read, update, and delete (CRUD) operations are applied to the same representation of the entity. For example, a data transfer object (DTO) representing a customer is retrieved from the data store by the data access layer (DAL) and displayed on the screen. A user updates some fields of the DTO (perhaps through data binding) and the DTO is then saved back in the data store by the DAL. The same DTO is used for both the read and write operations.

Traditional CRUD designs work well when only limited business logic is applied to the data operations. Scaffold mechanisms provided by development tools can create data access code very quickly, which can then be customized as required.

However, the traditional CRUD approach has some disadvantages:

- It often means that there's a mismatch between the read and write representations of the data, such as additional columns or properties that must be updated correctly even though they aren't required as part of an operation.
- It risks data contention when records are locked in the data store in a collaborative domain, where multiple actors operate in parallel on the same set of data. Or update conflicts caused by concurrent updates when optimistic locking is used. These risks increase as the complexity and throughput of the system grows. In addition, the traditional approach can have a negative effect on performance due to load on the data store and data access layer, and the complexity of queries required to retrieve information.
- It can make managing security and permissions more complex because each entity is subject to both read and write operations, which might expose data in the wrong context.

Solution

Command and Query Responsibility Segregation (CQRS) is a pattern that segregates the operations that read data (queries) from the operations that update data (commands) by using separate interfaces. This means that the data models used for querying and updates are different. The models can then be isolated.

Compared to the single data model used in CRUD-based systems, the use of separate query and update models for the data in CQRS-based systems simplifies design and implementation. However, one disadvantage is that unlike CRUD designs, CQRS code can't automatically be generated using scaffold mechanisms.

The query model for reading data and the update model for writing data can access the same physical store, perhaps by using SQL views or by generating projections on the fly.

The read store can be a read-only replica of the write store, or the read and write stores can have a different structure altogether. Using multiple read-only replicas of the read store can greatly increase query performance and application UI responsiveness, especially in distributed scenarios where read-only replicas are located close to the application instances.

Event sourcing

Context and problem

Most applications work with data, and the typical approach is for the application to maintain the current state of the data by updating it as users work with it. For example, in the traditional create, read, update, and delete (CRUD) model a typical data process is to read data from the store, make some modifications to it, and update the current state of the data with the new values—often by using transactions that lock the data.

The CRUD approach has some limitations:

- CRUD systems perform update operations directly against a data store, which can slow down performance and responsiveness, and limit scalability, due to the processing overhead it requires.
- In a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data.
- Unless there's an additional auditing mechanism that records the details of each operation in a separate log, history is lost.

Solution

The Event Sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they're persisted. Each event represents a set of changes to the data (such as `AddedItemToOrder`).

The events are persisted in an event store that acts as the system of record (the authoritative data source) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action that's required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events.

Typical uses of the events published by the event store are to maintain materialized views of entities as actions in the application change them, and for integration with external systems. For example, a system can maintain a materialized view of all customer orders that's used to populate parts of the UI. As the application adds new orders, adds or removes items on the order, and adds shipping information, the events that describe these changes can be handled and used to update the materialized view.

In addition, at any point it's possible for applications to read the history of events, and use it to materialize the current state of an entity by playing back and consuming all the events related to that entity. This can occur on demand to materialize a domain object when handling a request, or through a scheduled task so that the state of the entity can be stored as a materialized view to support the presentation layer.

Log aggregation

Context and problem

In the microservices architecture, our application consists of multiple services and service instances that are running on different servers and locations. Requests often cross multiple service instances.

When we had the monolith, the application is generating one log stream, which is generally stored in one log file/directory. Now, each service instance generates its own log file.

How to understand the behavior of an application and troubleshoot problems when log is splitted ?

Solution

Use a centralized logging service that aggregates logs from each service instance. When the log is aggregated, the users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs.

Distributed tracing

Context and problem

In the microservices architecture, requests often span multiple services. Each service handles a request by performing one or more operations, e.g. database queries, publishes messages, etc.

When a request fails, how to understand the behaviour and troubleshoot the problems ?

Solution

Instrument services with code that

- Assigns each external request a unique external Request ID
- Passes the external Request ID to all services that are involved in handling the request
- Includes the external Request ID in all log messages
- Records information (e.g. start time, end time) about the requests and operations performed when handling a external request in a centralized service

Audit logging

Context and problem

In a microservices architecture, we need more visibility about our services and how things are going, in addition to the logging system.

How to understand the behavior of users and the application and troubleshoot problems?

Solution

We can for example record the user activity in a database, or some special dedicated logging system.

Application metrics

Context and problem

In a microservices architecture, we need more visibility about our services and what's going on, in addition to indicators that we already have.

How to understand and articulate the application behavior ?

Solution

The recommended solution is to have a centralized metrics service that gathers and stocks the decision-enabling statistics of each of the service operations. Microservices can push their metrics information to the metrics service. On the other side, the metrics service can pull metrics from the microservice.

Health check API

Context and problem

It's a good practice, and often a business requirement, to monitor web applications and back-end services, to ensure they're available and performing correctly. However, it's more difficult to monitor services running in the cloud than it is to monitor on-premises services. There are many factors that affect applications such as network latency, the performance and availability of the underlying compute and storage systems, and the network bandwidth between them. The service can fail entirely or partially due to any of these factors. Therefore, you must verify at regular intervals that the service is performing correctly to ensure the required level of availability.

Solution

Implement health monitoring by sending requests to an endpoint on the application. The application should perform the necessary checks, and return an indication of its status.

A health monitoring check typically combines two factors:

- The checks (if any) performed by the application or service in response to the request to the health verification endpoint.
- Analysis of the results by the tool or framework that performs the health verification check.

The response code indicates the status of the application and, optionally, any components or services it uses. The latency or response time check is performed by the monitoring tool or framework.

Security between services: Access Token

Context and problem

In a microservices architecture and with the use of the API Gateway pattern. The application is composed of numerous services. The API gateway is the single entry point for client requests. It authenticates requests, and forwards them to other services, which might in turn invoke other services.

How to communicate the identity of the requestor to the services that handle the request?

Solution

The API Gateway authenticates the request and passes an access token (e.g. JSON Web Token) that securely identifies the requestor in each request to the services. A service can include the access token in requests it makes to other services.

What's next?

Now that we splitted our monolith and we discovered some useful patterns, we can start building our standalone microservices.

Chapter 9: Implementing the patterns

Introduction

Now, we can make our standalone microservices that wraps our **Bounded Context**s and that follow the patterns that we already studied.

To implement the patterns in our microservices architecture ecosystem, we will be using some libraries and frameworks that will optimize our task and make it easy.

We will be mainly using **Spring Cloud**, which is based on Spring Boot and provides a set of libraries to develop cloud native applications. Most of the technologies provided by the **Spring Team** come from the **Netflix** stack.

Spring Cloud comes with many production-ready components that we can use them directly, instead of implementing them from scratch.

Externalized configuration

In this step, we need a centralized configuration store. We can implement it from scratch but we just said that we can use production-ready components instead of the from scratch implementation.

Yeah! **Spring Cloud** comes with an implementation of a centralized configuration store: **Spring Cloud Config Server**.

The **Spring Cloud Config Server** enables a centralized management of configurations for the applications connected to it. The applications will retrieve their properties on startup from the Cloud Config server. The server itself gets its properties from several sources e.g. from file system or git repository.

Step 1: Generating the Config Server project skull

We will create `config-server`: a new Spring Boot application using Spring Initializr[5] we need to add the following dependencies:

- Config Server
- Actuator

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

Step 2: Defining the properties of the Config Server

In the `config-server/src/main/resources/application.yml` file inside the resources folder we have to add the following properties:

config-server/src/main/resources/application.yml

```
server:
  port: 8888 <1>

spring:
  profiles:
    active: native <2>
  application:
    name: config-server <3>
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/configurations <4>
```

1. The port that will be dedicated to the Config Server.
2. The "native" profile in the Config Server loads the config files from the local classpath or file system, and it doesn't use Git.
3. The application name - that the service will use to be identified to other services.
4. The directory where the configuration files are stored.

Step 3: Creating a centralized configuration

We need to put our centralized configuration files under the folder `src/main/resources/configurations/` of the **Config Server**. The property files have to be named like the application using them, but we will add the specific files successively when we need them. So far we will just create one property file (`application.yml` in the `src/main/resources/configurations/` of the `config-server`) which is global, inheriting its properties to all specific property files we will add later. Inside this master property file we will define where other services will be found and we tell **Cloud Config** that it should not override system properties made in the specific microservices:

config-server/src/main/resources/configurations/application.yml

```
name:
  value: nebrass
spring:
  cloud:
    config:
      override-system-properties: false
```

Step 4: Enabling the Config Server engine

Spring Initializr has already created the main class for the Spring Boot application. To make our application a **Config Server**, we just need to annotate the main class with `@EnableConfigServer`, in addition to the `@SpringBootApplication` annotation.

ConfigserverApplication.java

```
@EnableConfigServer
@SpringBootApplication
public class ConfigserverApplication {

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

Step 5: Run it !

Our Config Server is ready to start. Just run the main method and you should reach the configurations by using the application name and the spring-profile name as path parameters: http://localhost:8888/APP/PROFILE[6].

To get the global configurations you can use any non-existing application name and any profile name. If a file for the given application and the given profile exists, it will be merged with the global properties.

Getting the global configurations using a non-existing application name and profile

Step 6: Spring Cloud Config Client

Our microservices will bes clients for the **Config Server**. To do this we need to add Config Client starter dependency to the pom.xml of the service:

[6] http://localhost:8888/APP/PROFILE

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

And we need to tell the service where to find the **Config Server**. To do this we need to define the `spring.cloud.config.uri` property.

When we talk about defining application properties, the first thing that comes to my mind is defining them in the `application.properties` or the `application.yml` file. But, when dealing with Spring Cloud and your application's configuration is stored on a remote configuration server, such as **Config Server** which is our case, it's not the same, we use a `bootstrap.yml` or `bootstrap.properties` file.

What is exactly this file? What is the difference between storing properties in `application.yml` vs `bootstrap.yml`?

Spring Cloud application operates by creating a "**bootstrap**" context, which is a parent context for the main application. Out of the box it is responsible for loading configuration properties from the external sources, and also decrypting properties in the local external configuration files.

Note that the `bootstrap.yml` or `bootstrap.properties` file can contain generally the bootstrap config. Typically it contains two properties:

- Location of the configuration server (spring.cloud.config.uri)
- Name of the application (spring.application.name)

Upon startup, **Spring Cloud Config Client** makes an HTTP call to the **Config Server** with the name of the application and retrieves back it's configuration.

The `application.yml` or `application.properties` file contains standard application configuration - typically default configuration since any configuration retrieved during the bootstrap process will override configuration defined here.

So in our case, for a given microservice, the content of `bootstrap.yml` will look like:

bootstrap.yml

```
spring:
  cloud:
    config:
      uri: http://localhost:8888
  application:
    name: microservice-name
```

If we have some default values for some properties we can define them in the `application.properties` or the `application.yml` file.

Service Discovery and Registration

For the Service Discovery Pattern, we will be using **Eureka**, a component from the **Spring Cloud Neflix** project.

Eureka is comparable to a registry, where we can find the services that we want reach. The instances of the our microservices will be registered with a Eureka instance with their host, port and other meta-data. If any other service needs to communicate with one of the microservices, it just has to ask Eureka for the location to establish the connection.

How does this work? The clients have to register themselves with Eureka. Afterwards they have to renew their status by sending an heartbeat at regular intervals. If the heartbeat of one Eureka client is missing, the Eureka server marks the instance in its registry as "DOWN". On a normal shutdown of a client, it has to send a cancel request to get removed from the registry. To lookup other services, the clients have to fetch the registry. For visual feedback of the service statuses, Eureka ships with a very useful dashboard.

Eureka is composed of microservices:

- a service registry (**Eureka Server**),
- a REST service which registers itself at the registry (**Eureka Client**)

Step 1: Generating the Eureka Server project skull

We will create `discovery-service`: a new Spring Boot application using Spring Initializr[7] we need to add the following dependencies:

- Eureka Server
- Actuator
- Config Client <== The Config Client is necessary to retrieve the configurations from the **Config Server**.

Step 2: Defining the properties of the Eureka Server

The content of the `bootstrap.yml` of the **Eureka Server**:

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

discovery-service/src/main/resources/bootstrap.yml

```
spring:
  cloud:
    config:
      uri: http://localhost:8888 <1>
  application:
    name: discovery-service <2>
```

1. Our application needs to retrieve configuration from the **Config Server**, so our service needs the Config Server uri.
2. The application name - that the service will use to be identified to other services.

Step 3: Creating the main configuration of the Eureka Server

The main config file for service-discovery has to be created on the Config Server side with the name used forthe property value of `spring.application.name`.

In the **Step 2**, we defined `discovery-service` as the application name for the **Eureka Server**. So, the main configuration file that will be stored on the **Config Server** will be saved as:

config-server/src/main/resources/configurations/discovery-service.yml

```
server:
  port: 8761 <1>

eureka:
  instance:
    hostname: localhost <2>
  client:
    registerWithEureka: false <3>
    fetchRegistry: false <3>
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

1. Eureka Server port
2. Eureka Server hostname
3. Eureka Server doesn't have to register and fetch anything from Eureka because it is the Eureka Server itself.

Step 4: Enabling the Eureka Server engine

Spring Initializr has already created the main class for the Spring Boot application. To make our application an Eureka Server, we just need to annotate the main class with `@EnableEurekaServer`, in addition to the `@SpringBootApplication` annotation.

EurekaApplication.java

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaApplication {

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

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

Now we have a **Service Registry** on which all of our services need to know.

So this step aims to add the **Eureka Server** properties to the global `application.yml` of the Config Server.

config-server/src/main/resources/configurations/application.yml

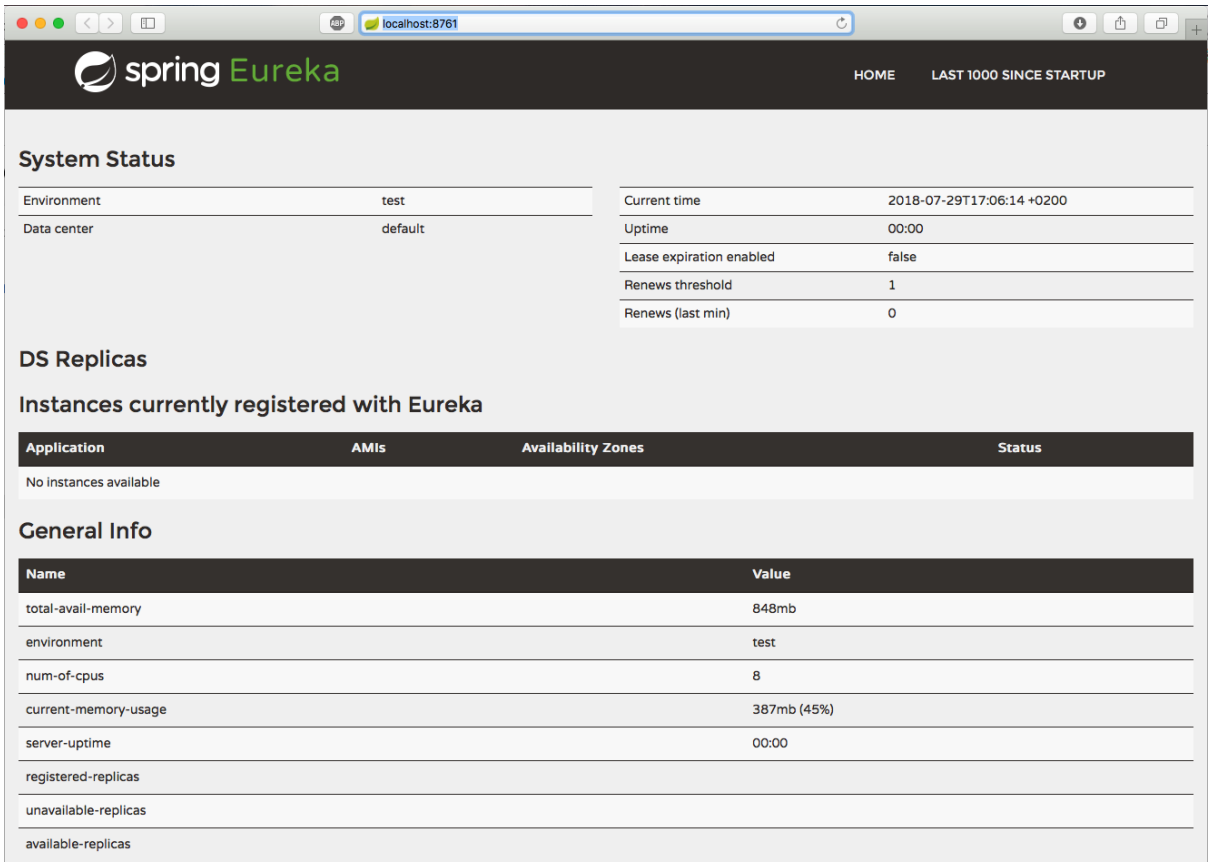
```
eureka:
  instance:
    hostname: localhost
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/

name:
  value: nebrass

spring:
  cloud:
    config:
      override-system-properties: false
```

Step 6: Run it !

Our Eureka Server is ready to go. But, it can be started only after restarting the Config Server, so that the newly inserted `service-discovery.yml` is applied.

After starting the **Eureka Server**, you can access its dashboard, go to http://localhost:8761/[8].

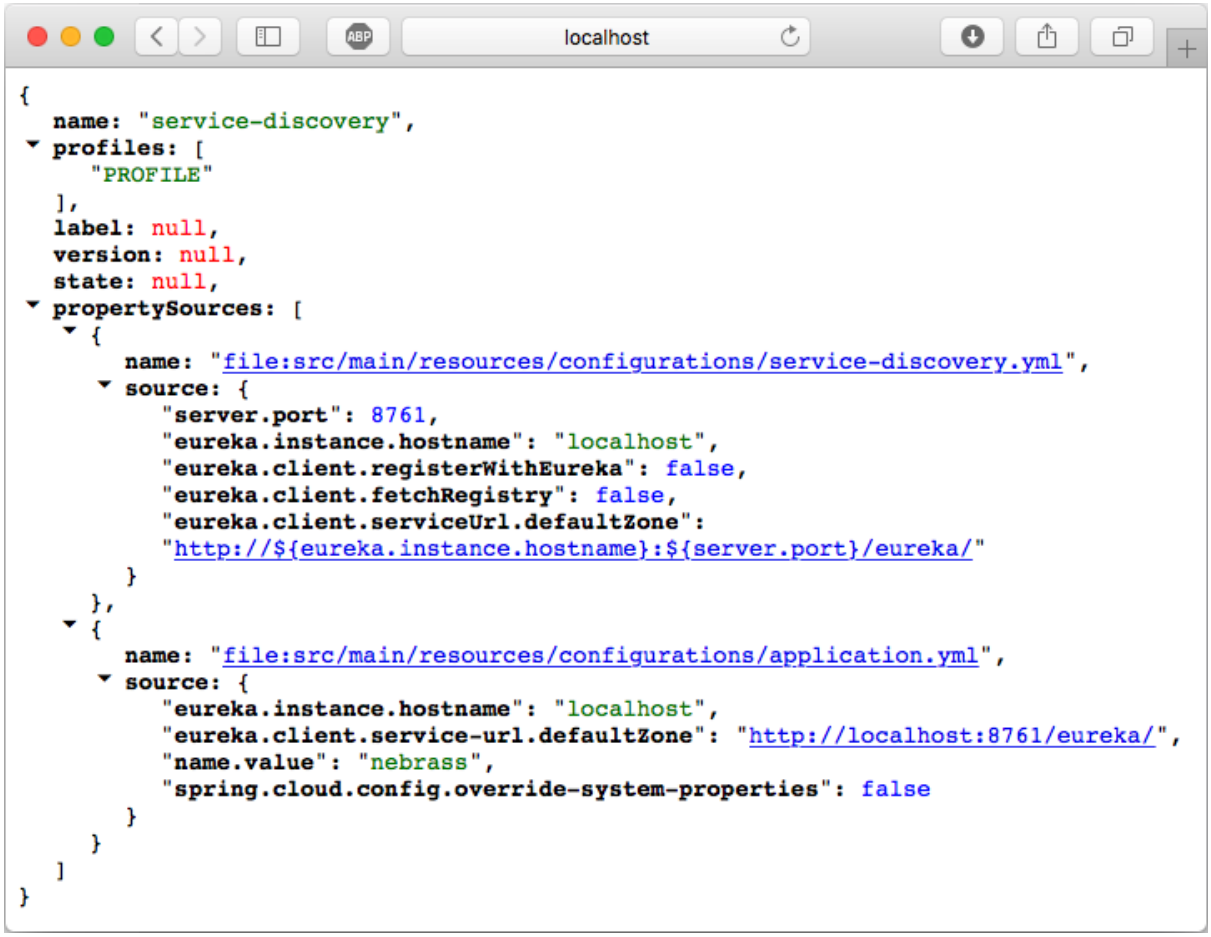
[8] http://localhost:8761/



Eureka Server Dashboard

To see how the global properties are merged within the specific properties of service-discovery go to http://localhost:8888/service-discovery/PROFILE[9].

[9] http://localhost:8888/service-discovery/PROFILE



The generated properties of service-discovery

Step 7: Client Side Load Balancer with Ribbon

In the **Order Service**, as we mentionned in the *chapter 7*, will communicate with the **Product Service** using REST calls instead of the language-level calls in the old monolith. The REST call will be using the Spring's **RestTemplate** component.

The *RestTemplate* needs as input the *URL* of the REST resource. A very important element of the **URL** is the **HOST** and **PORT** of the service, which are dynamically changing and not guaranteed to be static. But, whenever a service got a new IP address, after a failure or restart or even by the arrival of a new instance of that service, the **Eureka Server** will be always aware of these changes, and can tell us where is located a given service.

At this step, we can benefit of a great functionnality offered by the Registry: the resolution of the **Service Name** to its **Network Location**.

We can register *RestTemplate* as a **Spring Bean** with **@LoadBalanced** annotation. The **RestTemplate** with **@LoadBalanced** annotation will internally use **Ribbon LoadBalancer** to resolve the *Service Name* and invoke REST endpoint using one of the available instances.

ServiceConfiguration.java

```
@Configuration
public class ServiceConfiguration {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

}
```

Ribbon is a client-side load balancer that gives you a lot of control over the behavior of HTTP and TCP clients. It comes with the **Eureka Client** Starter bundle.

With this kind of automatic Service Registration and Discovery mechanism, we don't need to worry about how many instances are running and what are their hostnames and ports etc.

Distributed tracing

To get a better understanding of what is going on we need to be able to trace a request as it pass through a number of cooperating microservices and measure the processing time in each microservice that is involved in responding to the request.

The trace events for one request must be collected, grouped together and presented in an understandable way. This is exactly what Zipkin is about!

Distributed tracing ties all individual service calls together, and associates them with a business request through a unique generated ID.

We will be using **Spring Cloud Sleuth** and **Zipkin**:

- **Spring Cloud Sleuth** – library available as a part of **Spring Cloud project**. Lets you track the progress of subsequent microservices by adding the trace IDs for every call and span IDs at the requested points in an application.

What is a Trace ? What is a Span ?

A *trace* represents the whole processing of a request. A *span* represents the processing that takes part in each individual microservice, as a step in the processing of a request. All trace events from processing a request share a common unique *Trace Id*. I> Each individual processing step, i.e. a *span*, is given a unique *Span Id*.

- **Zipkin** – distributed tracing system that helps to gather timing data for every request propagated between independent services. It has simple management console where we can find visualization of the time statistics generated by subsequent services.

Step 1: Getting the Zipkin Server

The quickest way to get started is to fetch the latest release as a self-contained executable jar:

```
curl -sSL https://zipkin.io/quickstart.sh | bash -s
java -jar zipkin.jar
```

You will see a log of a Spring Boot application :) Yes ! **Zipkin Server** is a *Spring Boot* based application. When it's started, you will see:

```
... INFO 3900 --- [ main] o.s.b.w.e.u.UndertowServletWebServer     : Undertow started on port(s) 9411 \
(http) ...
... INFO 3900 --- [ main] z.s.ZipkinServer                        : Started ZipkinServer in 4.196 sec\
onds ...
```

Once the server is running, you can view traces with the **Zipkin UI** at `http://localhost:9411/zipkin/`.

Step 2: Listing the dependencies to add to our microservices

When creating a microservice, to have the distributed tracing, we need to add **Sleuth** and **Zipkin Client** dependencies.

Sleuth

We have a Spring Cloud Starter project for **Sleuth**, the maven dependency to add to our `pom.xml` is:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

Once you add **Sleuth** starter and start the services you can see in logs something like this:

```
2018-07-30 .. INFO [product-service,c01f99,c01f99,true] 18582 --- [nio-9991-exec-5] c.t.l.m.p.ProductC\
ontroller...
```

Sleuth includes the pattern **[appname,traceId,spanId,exportable]** in logs from the **MDC** (Mapped Diagnostic Context) feature of the Logging frameworks.

The pattern is composed of the respective parameters:

- **appname**: the name of the application that logged the `span`.
- **traceId**: the id of the latency graph that contains the `span`.
- **spanId**: the id of a specific operation
- **exportable**: whether the log should be exported to Zipkin or not - Without the Zipkin Client dependency, this `exportable` will be always `false`.

Zipkin Client

We have a Spring Cloud Starter project for **Zipkin**, the maven dependency to add to our `pom.xml` is:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

And we need to add this entry to the global properties in in the Config Server:

config-server/src/main/resources/configurations/application.yml

```
spring:
  sleuth:
    sampler:
      probability: 1
```

This property means that **100% of tracing information will be exported to Zipkin**.

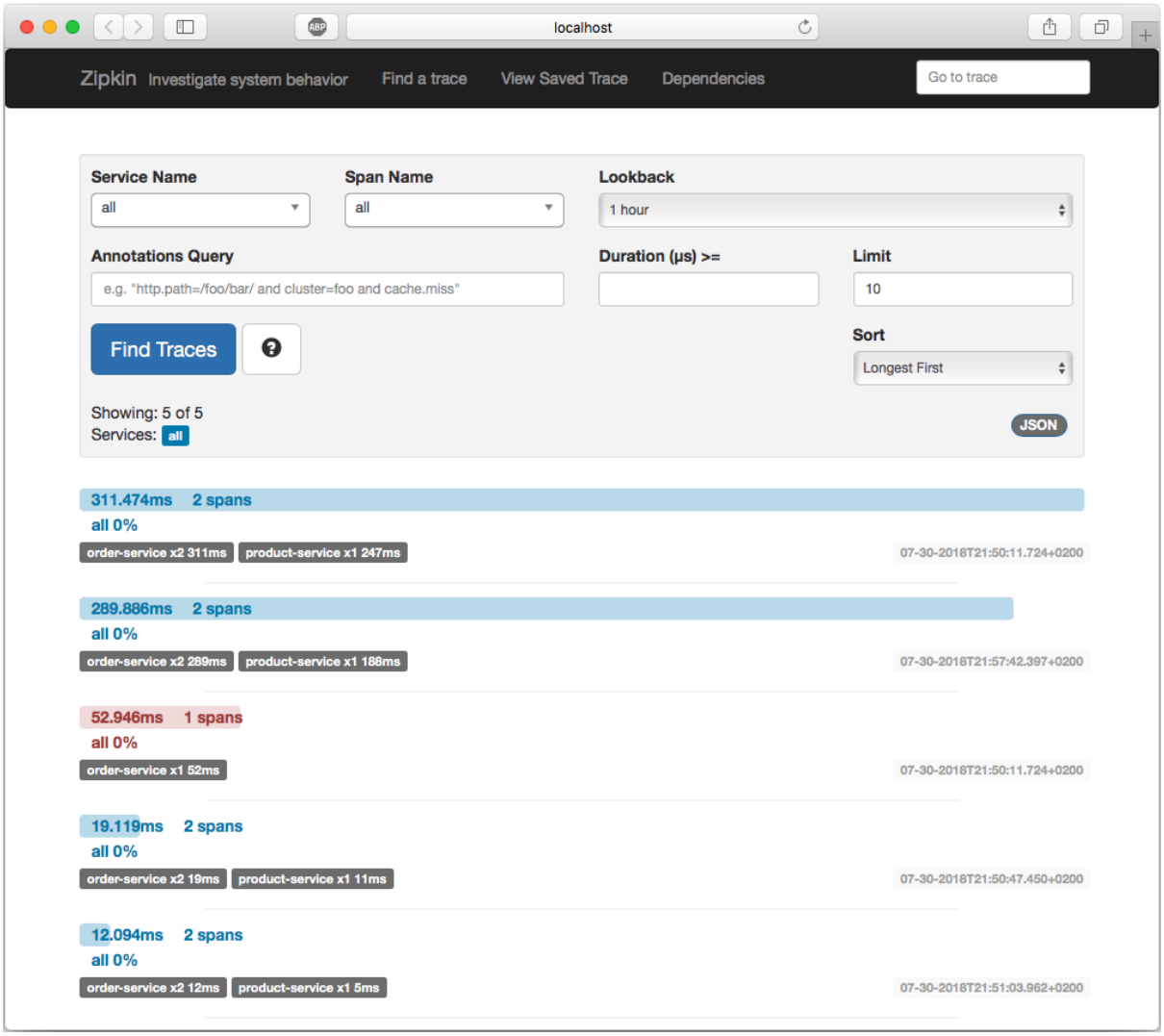
Once the **Zipkin** starter is in the classpath, when you start the service, it will be exporting the `traces` automatically to our **Zipkin Server**.

But how can the **Zipkin Client** will locate our **Zipkin Server** ?

The answer is easy, the **Zipkin Client** by default will send the traces to a default URL: `localhost:9411` (which already the URL of our Zipkin Server). You can change the location of the service by defining a new property `spring.zipkin.baseUrl`.

Once these dependencies are added, when microservices will communicate, on the Zipkin Dashboard, accessible at http://localhost:9411/zipkin/[10], we can see something looking like this:

[10] http://localhost:9411/zipkin/



Zipkin Dashboard

I know that you are searching about these values. Dont worry, I made a little example to generate these traces just to have an example of a populated **Zipkin Dashboard**. We will have real traces later, we will just list the components that we will be using for evey pattern :)

Health check API

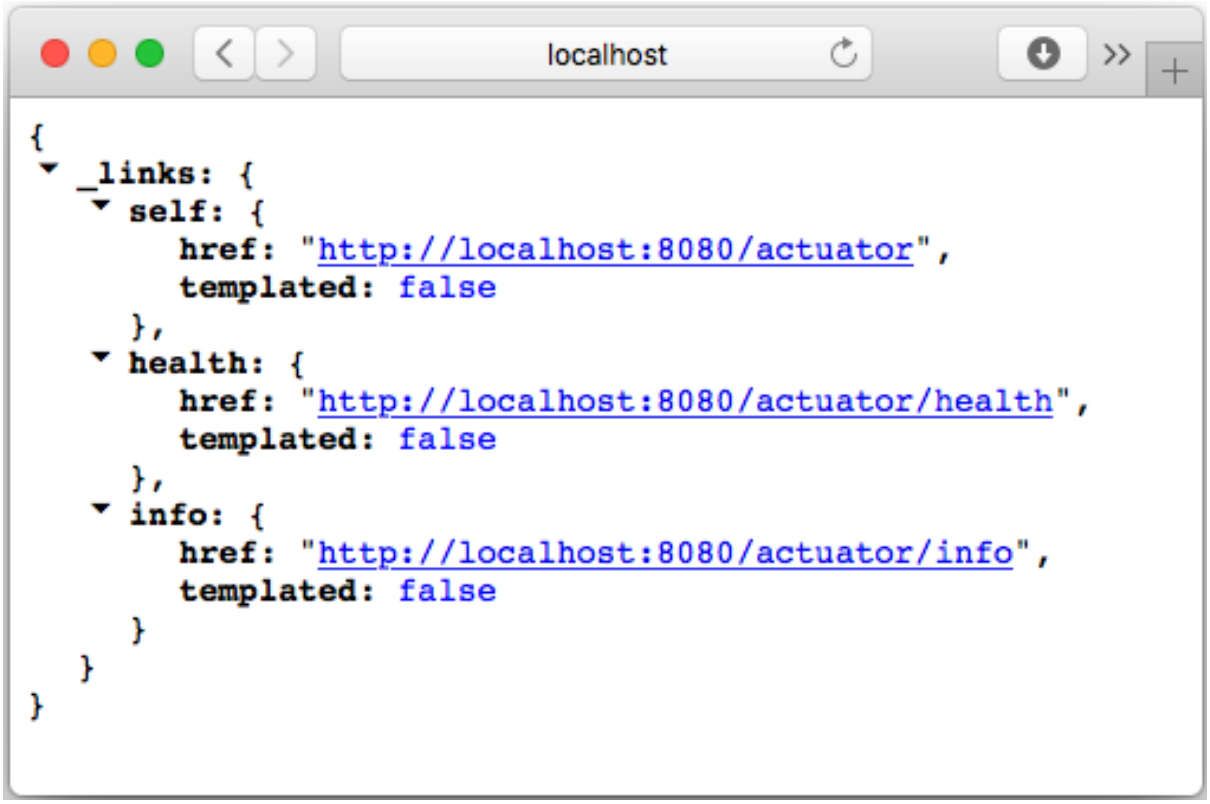
For this pattern, we have a ready great library: the **Spring Boot Actuator**:

Spring Boot Actuator supplies several endpoints in order to monitor and interact with your application. It does so by providing built-in endpoints, but you are also able to build your own endpoints.

To have Actuator in your project, just add the Maven dependency to your pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

That's all !! If you start the application and you go to http://localhost:8080/actuator/[11], you will see an overview of the exposed **Actuator** endpoints:



Overview of the exposed Actuator endpoints

Circuit Breaker

The **Spring Cloud Netflix** stack has a library that can help us implement the **Circuit Braker** pattern easily: it's **Hystrix**.

Step 1: Add the Maven dependency to your project

The Maven dependency to add in the `pom.xml` of each of our microservices:

[11] http://localhost:8080/actuator/

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

Step 2: Enable the circuit beaker

To enable Circuit Breaker, we need to annotate the **Main Class** of our microservice with the @EnableCircuitBreaker:

OrderServiceApplication.java

```
@EnableCircuitBreaker
@SpringBootApplication
public class OrderServiceApplication {

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

Step 3: Apply timeout and fallback method

In my boutique scenarios, we have some cases where the **Order Service** calls the **Product Service** to get the information of a given product ID.

In the **Order Service**, as we mentionned in the *chapter 7*, will communicate with the **Product Service** using REST calls using the Spring's **RestTemplate** component.

One of the recommended practices is to wrap the methods calling the *RestTemplate* component in the same class; which will be the **local client** of the **remote** service.

In the **Order Service**, the **ProductServiceClient** will look like this:

ProductServiceClient.java

```
@Slf4j
@Service
public class ProductServiceClient {

    private final RestTemplate restTemplate;

    public ProductServiceClient(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @HystrixCommand(fallbackMethod = "getDefaultProductById") <1>
    public Optional<ProductDto> getProductById(Long productId) {
        ResponseEntity<ProductDto> productResponse = restTemplate
                .getForEntity("http://product-service/api/product/{id}", <2>
                        ProductDto.class,
                        productId);

        if (productResponse.getStatusCode() == HttpStatus.OK) {
            return Optional.ofNullable(productResponse.getBody());
```

```
        } else {
            log.error("Unable to get product with ID: " + productId
                    + ", StatusCode: " + productResponse.getStatusCode());
            return Optional.empty();
        }
    }

    Optional<ProductDto> getDefaultProductById(Long productId) { <3>
        log.info("Returning default ProductById for product Id: " + productId);

        ProductDto productDto = new ProductDto();
        productDto.setId(productId);
        productDto.setName("UNKNOWN");
        productDto.setDescription("NONE");

        return Optional.ofNullable(productDto);
    }

}
```

1. We have annotated the method from where we are making a REST call with the **@HystrixCommand** so that if it doesn't receive the response within the certain time limit the call gets timed out and invoke the configured fallback method. The fallback method should be defined in the same class and should have the same signature.
2. The Eureka Client will be resolving `product-service` to the location of the an available instance.
3. In the fallback method **getDefaultProductById()** we create an empty ProductDTO with the requested *productId*.

Step 4: Enable the Hystrix Stream in the Actuator Endpoint

To be able to monitor the Circuit Breakers, we need to enable Hystrix Stream in the Actuator Endpoint. This can be done by adding this property to every microservice that enabled **Hystrix**, in its configuration file located in the **Config Server**.

For example; our **order-service.yml** config file will look like this:

config-server/src/main/resources/configurations/order-service.yml

```
server:
    port: 9990

management:
  endpoints:
    web:
      exposure:
        include: "*"
```

This property in reality enable and expose all the available Actuator endpoints, and our great **Hystrix** endpoint will be included with them.

Step 5: Monitoring Circuit Breakers using Hystrix Dashboard

Once we add Hystrix starter to **Order-Service**, we can get the circuits status as a stream of events using an **Actuator** endpoint http://localhost:9990/actuator/hystrix.stream[12] , as our **Order-Service** is running on the port 9990.

Spring Cloud also provides a nice dashboard to monitor the status of **Hystrix commands**.

Let's create a Spring Boot application with **Eureka Discovery** (aka **Eureka Client**), **Config Client** and **Hystrix Dashboard** starters:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

Next we need to annotate the **Main Class** with **@EnableHystrixDashboard**.

HystrixDashboardApplication.java

```
@EnableHystrixDashboard
@SpringBootApplication
public class HystrixDashboardApplication {

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

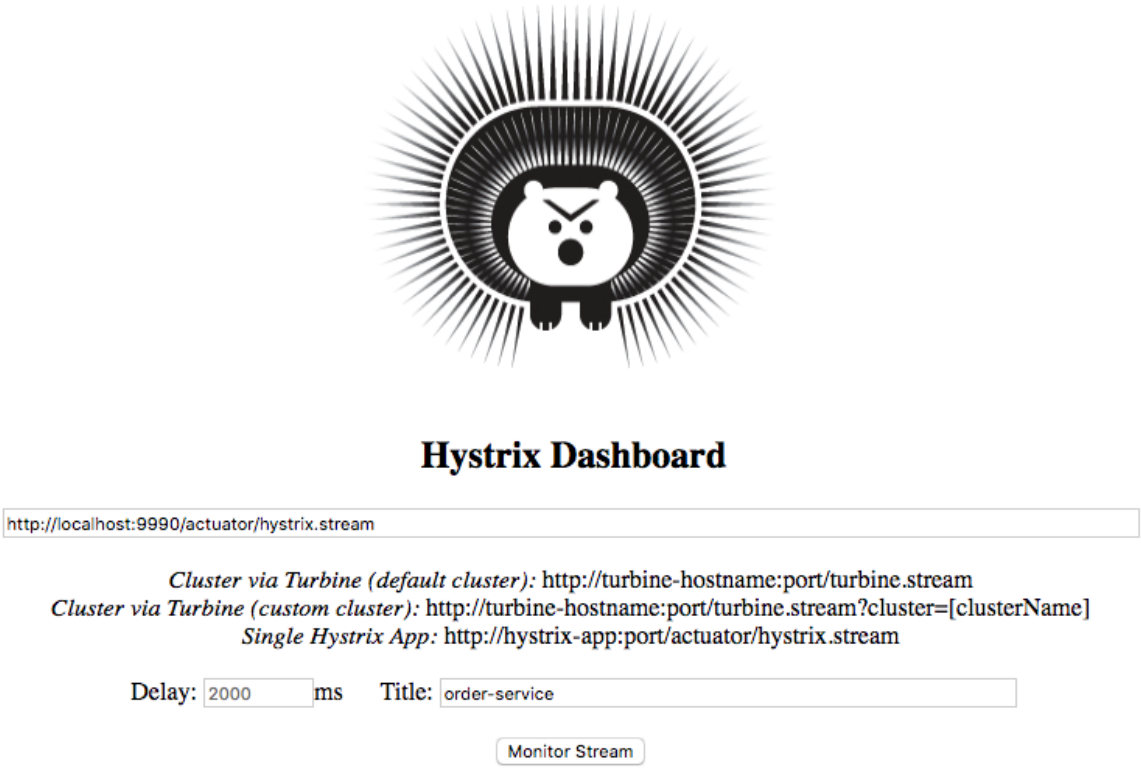
The configuration file of our **Hystrix Dashboard** applications is:

config-server/src/main/resources/configurations/hystrix-dashboard.yml

```
server:
    port: 8988
```

Let's run our **Hystrix Dashboard**, then go to http://localhost:8788/hystrix[13] to view the dashboard:

[12] http://localhost:9990/actuator/hystrix.stream
[13] http://localhost:8988/hystrix



Hystrix Dashboard

Now in **Hystrix Dashboard** home page enter `http://localhost:9990/actuator/hystrix.stream` as stream URL and give Order Service as Title and click on Monitor Stream button.

Now invoke the **order-service** REST endpoint which internally invokes the **product-service** REST endpoint and you cansee the **Circuit** status along with how many calls succeed and how many failures occurred etc.

Hystrix samples for the Order-Service

API Gateway

API Gateway, aka **Edge Service**, is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.

We will use the **Spring Cloud Netflix Zuul** project. The tool provides out-of-the-box routing mechanisms often used in microservices applications as a way of hiding multiple services behind a single facade.

Step 1: Generating the API Gateway project skull

We will create `gateway-service`: a new Spring Boot application using Spring Initializr[14] we need to add the following dependencies:

- Zuul
- Config Client
- Eureka Discovery
- Sleuth
- Actuator

Zuul provides three basic components used for configuration: routes, predicates and filters.

- **Route** is the basic building block of the gateway. It contains destination URI and list of defined predicates and filters.
- **Predicate** is responsible for matching on anything from the incoming HTTP request, such as headers or parameters.
- **Filter** may modify request and response before and after sending it to downstream services.

All these components may be set using configuration properties. So, we need to create and place on the configuration server file `gateway-service.yml` with the routes defined for our microservices.

Step 2: Enable the Zuul Capabilities

Next we need to annotate the **Main Class** with **@EnableZuulProxy** :

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

ApiGatewayApplication.java

```
@EnableZuulProxy
@SpringBootApplication
public class ApiGatewayApplication {

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

Step 3: Defining the route rules

We need to define the routes. In order to make `order-service` available on our gateway under path `/order/**`, we should define a route matching that `path` with the `serviceID`, as it is registered in **Eureka**.

config-server/src/main/resources/configurations/gateway-service.yml

```
server:
  port: 8222
zuul:
  routes:
    order:
      path: /order/**
      serviceId: order-service
    product:
      path: /product/**
      serviceId: product-service
    customer:
      path: /customer/**
      serviceId: customer-service
```

Step 4: Enabling API specification on gateway using Swagger2

When a microservice has enabled **Swagger**, it will expose a **Swagger API** documentation under path `/v2/api-docs`.

The idea here, is to have a **Swagger2 UI** on our **API Gateway**, that aggregates the **Swagger Resources** exposed by our **swagger-enabled** microservices. To do this, we need to implement `SwaggerResourcesProvider` interface inside our `gateway-service`. This Bean is responsible for aggregating the locations of Swagger resources of our microservices. Here's the implementation of `SwaggerResourcesProvider` that takes the required locations from the **ZuulRoutes** :

ProxyApi.java

```
@Configuration
public class ProxyApi {

    private final ZuulProperties zuulProperties;

    public ProxyApi(ZuulProperties zuulProperties) {
        this.zuulProperties = zuulProperties;
    }

    @Primary
    @Bean
    public SwaggerResourcesProvider swaggerResourcesProvider() {
        return () -> {
            List<SwaggerResource> resources = new ArrayList<>();
            zuulProperties.getRoutes()
                    .values()
                    .forEach(route -> resources.add(createSwaggerResource(route)));
            return resources;
        };
    }

    private SwaggerResource createSwaggerResource(ZuulRoute route) {
        SwaggerResource swaggerResource = new SwaggerResource();
        swaggerResource.setName(route.getServiceId());
        swaggerResource.setLocation("/" + route.getId() + "/v2/api-docs");
        swaggerResource.setSwaggerVersion("2.0");
        return swaggerResource;
    }
}
```

Step 5: Run it!

Here's Swagger UI for our sample microservices system available under address http://localhost:8060/swagger-ui.html[15]

Log aggregation and analysis

For the Log aggregation we will use the famous ELK stack. So, what is the ELK Stack? "ELK" is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana.

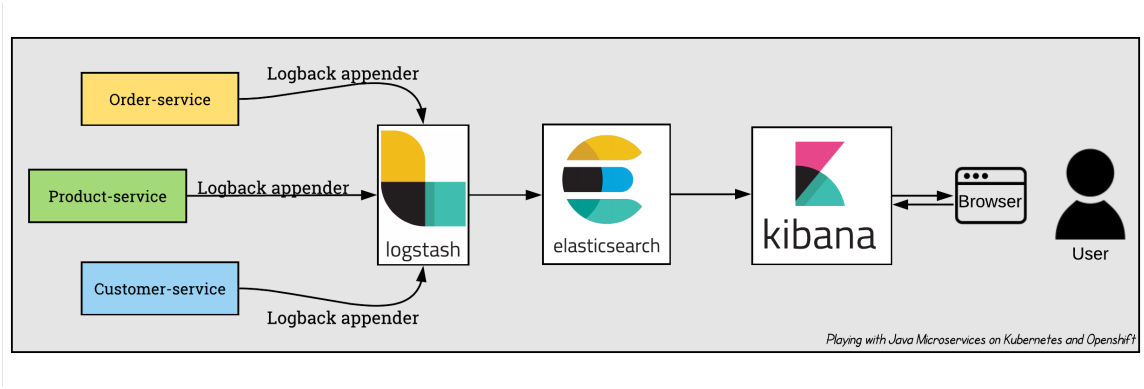
[15] http://localhost:8060/swagger-ui.html

The ELK Stack logos

- Elasticsearch is a search and analytics engine.
- Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch.
- Kibana lets users visualize data with charts and graphs in Elasticsearch.

Together, these tools are most commonly used for centralizing and analyzing logs in distributed systems. The ELK Stack is popular because it fulfills a need in the log analytics space.

Our usecase of ELK with our microservices will be like:



Usecase of the ELK stack in our Microservices Architecture

All our microservices will push their respective logs to **Logstash**, which will use **Elasticsearch** to index them. The indexed logs can be consumed later by **Kibana**.

Our microservices will use **Logback**: one of the most popular logging frameworks in the market. Logback offers a faster implementation than Log4j, provides more options for configuration, and more flexibility in archiving old log files.

Three classes comprise the Logback architecture; Logger, Appender, and Layout.

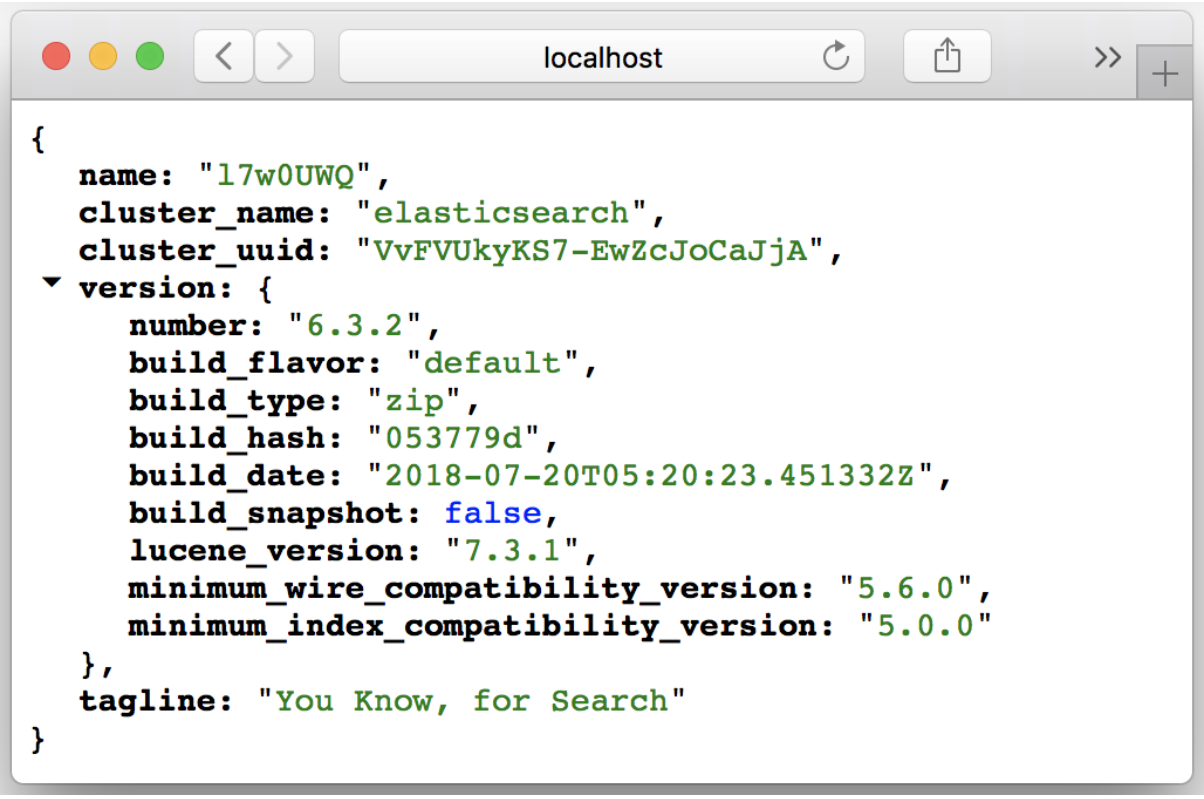
- A **Logger** is a context for log messages. This is the class that applications interact with to create log messages.
- **Appender**s place log messages in their final destinations. A **Logger** can have more than one **Appender**. We generally think of **Appender**s as being attached to text files, but **Logback** is much more potent than that.

- **Layout** prepares messages for outputting. **Logback** supports the creation of custom classes for formatting messages, as well as robust configuration options for the existing ones.

Step 1: Installing Elasticsearch

- Download **Elasticsearch** zip file from https://www.elastic.co/downloads/elasticsearch[16]
  `wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.3.2.zip`
- Extract it (unzip it)
  `unzip elasticsearch-6.3.2.zip`
- Run it !
  `cd elasticsearch-6.3.2 ./bin/elasticsearch`
  When started, you can access Elasticsearch on http://localhost:9200[17].

You will get something looking like this:



Elasticsearch index page

Step 2: Installing Kibana

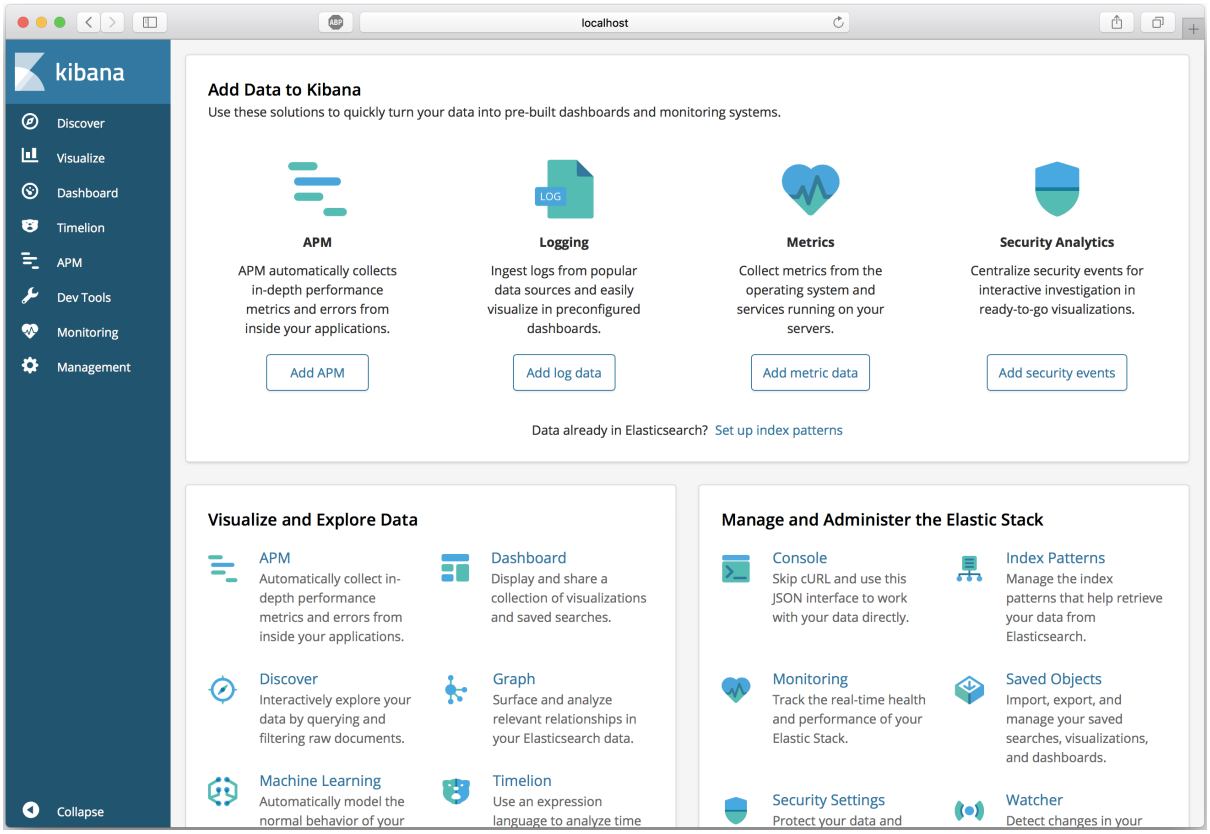
- Download the appropriate **Kibana** distribution for your OS (OS X in ly case) from https://www.elastic.co/downloads/Kibana[18]

[16] https://www.elastic.co/downloads/elasticsearch
[17] http://localhost:9200
[18] https://www.elastic.co/downloads/Kibana

```
wget https://artifacts.elastic.co/downloads/kibana/kibana-6.3.2-darwin-x86_-
64.tar.gz
```

- Extract the archive
  ```
  tar xvzf kibana-6.3.2-darwin-x86_64.tar.gz
  ```
- Run it !
  ```
  cd kibana-6.3.2-darwin-x86_64 ./bin/kibana
  ```
  When started, you can access Kibana on http://localhost:5601[19].

You will get something looking like this:



Kibana home page

Step 3: Installing & Configuring Logstash

Installing Logstash

- Download **Logstash** zip from https://www.elastic.co/downloads/logstash[20]
  ```
  wget https://artifacts.elastic.co/downloads/logstash/logstash-6.3.2.zip
  ```
- Extract it (unzip it)
  ```
  unzip logstash-6.3.2.zip
  ```

[19] http://localhost:5601
[20] https://www.elastic.co/downloads/logstash

Configuring Logstash

Typical Logstash config file consists of three main sections: **input**, **filter** and **output**. Each section contains plugins that do relevant part of the processing.

- **input**: An input plugin enables a specific source of events to be read by Logstash —In our case, it will be a JSON Flow of logs sent by **Logback** in our microservices: we will use the *TCP plugin* with **json** codec and will be listening on a defined port, 5000 for example.
- **filter**: A filter plugin performs intermediary processing on an event. Filters are often applied conditionally depending on the characteristics of the event - In our case, we dont have yet filtering needed.
- **output**: An output plugin sends event data to a particular destination. Outputs are the final stage in the event pipeline - In our case, we will be sending our log events to Elasticsearch (we need to define where is the Elasticsearch **Hosts** and the **Index**).

We need to create a `logstash.conf` file in the root directory of the Logstash installation, which contains:

logstash-6.3.2/logstash.conf

```
input {
    tcp {
        port => 5000
        codec => "json"
    }
}

output {
    elasticsearch {
        hosts => ["127.0.0.1"]
        index => "micro-%{serviceName}"
    }
}
```

Run it !

Once we created the config file, we need to pass it as paramter with the start command:

```
./bin/logstash -f logstash.conf
```

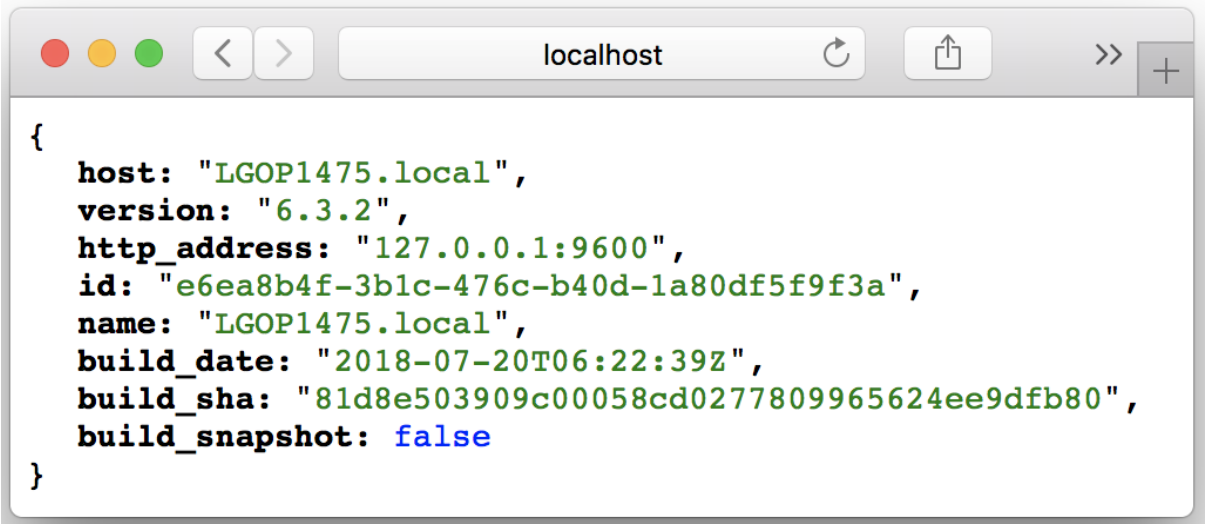
Logstash will start, in the console you can see that it exposes two ports:

- 9600 : Logstash API endpoint
- 5000 : TCP input listener

It will now listen & ship log events to Elasticsearch. When started, you can access **Logstash** on http://localhost:9600[21].

[21] http://localhost:9600

You will get something looking like this:



Logstash index page

Step 4: Enabling the Logback features

Adding Logback libraries to our microservices

We need to add these *Maven* dependencies to the `pom.xml` of every microservice:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>5.1</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
</dependency>
```

- logstash-logback-encoder: Provides logback encoders, layouts, and appenders to log in JSON format.
- logback-core + logback-classic: They can be assimilated to a significantly improved version of **log4j**. Moreover, **logback-classic** natively implements the **SLF4J API** so that you can readily switch back and forth between logback and other logging frameworks such as `log4j` or `java.util.logging`.

Adding Logback configuration file to our microservices

After adding the *Maven* dependencies to our microservices, we need to add a `LogbackConfiguration` file:

LogbackConfiguration.java

```
@Configuration
public class LogbackConfiguration {

    private static final String LOGSTASH_APPENDER_NAME = "LOGSTASH";
    private static final String ASYNC_LOGSTASH_APPENDER_NAME = "ASYNC_LOGSTASH";

    private final Logger LOG = LoggerFactory.getLogger(LoggingConfiguration.class);
    private final LoggerContext CONTEXT = (LoggerContext) LoggerFactory.getILoggerFactory();

    private final String appName;
    private final String logstashHost;
    private final Integer logstashPort;
    private final Integer logstashQueueSize;

    public LogbackConfiguration(
            @Value("${spring.application.name}") String appName,
            @Value("${logstash.host}") String logstashHost,
            @Value("${logstash.port}") Integer logstashPort,
            @Value("${logstash.queue-size}") Integer logstashQueueSize) {

        this.appName = appName; <1>
        this.logstashHost = logstashHost; <1>
        this.logstashPort = logstashPort; <1>
        this.logstashQueueSize = logstashQueueSize; <1>

        addLogstashAppender(CONTEXT);
    }

    private void addLogstashAppender(LoggerContext context) {
        LOG.info("Initializing Logstash logging");

        LogstashTcpSocketAppender logstashAppender = new LogstashTcpSocketAppender(); <2>
        logstashAppender.setName(LOGSTASH_APPENDER_NAME);
        logstashAppender.setContext(context);
        String customFields = "{\"servicename\":\"" + this.appName + "\"}"; <3>

        // More documentation is available at: https://github.com/logstash/logstash-logback-encoder
        LogstashEncoder logstashEncoder = new LogstashEncoder();

        // Set the Logstash appender config
        logstashEncoder.setCustomFields(customFields);
        logstashAppender.addDestinations( <2>
                new InetSocketAddress(this.logstashHost, this.logstashPort) <2>
        );

        ShortenedThrowableConverter throwableConverter = new ShortenedThrowableConverter();
        throwableConverter.setRootCauseFirst(true);
        logstashEncoder.setThrowableConverter(throwableConverter);
        logstashEncoder.setCustomFields(customFields);

        logstashAppender.setEncoder(logstashEncoder);
        logstashAppender.start();

        // Wrap the appender in an Async appender for performance
        AsyncAppender asyncLogstashAppender = new AsyncAppender(); <4>
        asyncLogstashAppender.setContext(context); <4>
```

```
        asyncLogstashAppender.setName(ASYNC_LOGSTASH_APPENDER_NAME); <4>
        asyncLogstashAppender.setQueueSize(this.logstashQueueSize); <4>
        asyncLogstashAppender.addAppender(logstashAppender); <4>
        asyncLogstashAppender.start(); <4>

        context.getLogger("ROOT").addAppender(asyncLogstashAppender); <5>
    }
}
```

1. Injects values from the configuration properties of the application - they can be loaded from local properties or from the Config Server
2. Creates a **LogstashTcpSocketAppender** that will send logs to our **Logstash** server.
3. Sends a custom field in the log event payload, that will include the `servicename` - This field will give us the ability to know from which microservice this event is received.
4. Wraps the appender in an Async appender for performance purposes.
5. Registers the appender's wrapper in the root logger.

Step 5: Adding the Logstash properties to the Config Server

The `logstash.host`, `logstash.port` and `logstash.queue-size` values injected in the previous step need to be defined in our configuration. These values are used by all our microservices, so we need to define them in the common `application.yml` file in the **Config Server**:

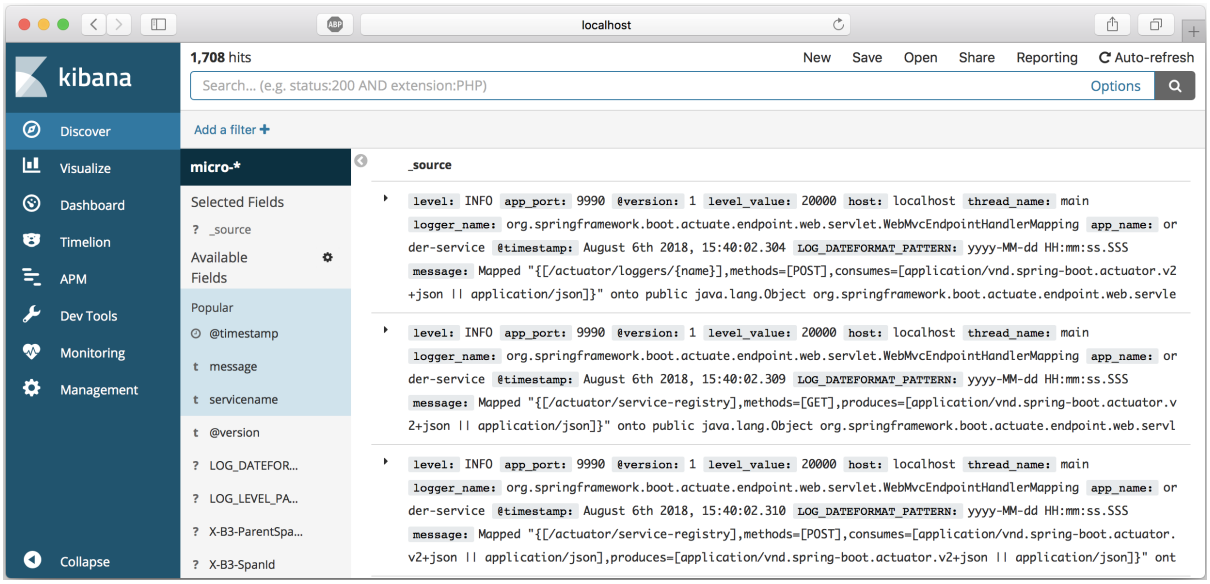
config-server/src/main/resources/configurations/application.yml

```
eureka:
  instance:
    hostname: localhost
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/

name:
  value: nebrass

spring:
  cloud:
    config:
      override-system-properties: false
  sleuth:
    sampler:
      probability: 1

logstash:
  host: localhost
  port: 5000
  queue-size: 512
```

Step 6: Attending the Kibana party

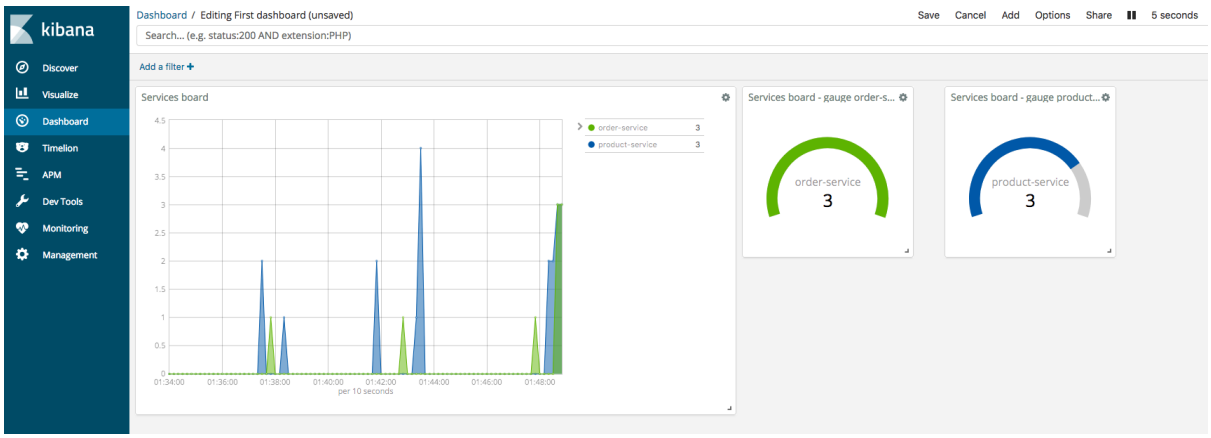
Now, if you go to Kibana at http://localhost:5601[22], on the section **Discover** you will get:

[22] http://localhost:5601



Kibana Discover Page

You can play around with Kibana, and generate cool Dashboards with many analysis functions; Here is an example of a dashboard:



Kibana Dashboard Example

You can learn more about Kibana here: Kibana User Guide[23].

Conclusion

In this chapter, we implemented many patterns that we presented in the previous chapter. We already implemented:

- Externalized configuration : Config Server
- Service Discovery and Registration : Eureka Server
- Distributed tracing : Zipkin
- Health check API : Actuator

[23] https://www.elastic.co/guide/en/kibana/6.3/index.html

- Circuit Breaker : Hystrix
- API Gateway : Zuul
- Log aggregation and analysis : ELK Stack

These components are wrapping our microservices: we need them to insure operating our microservices.

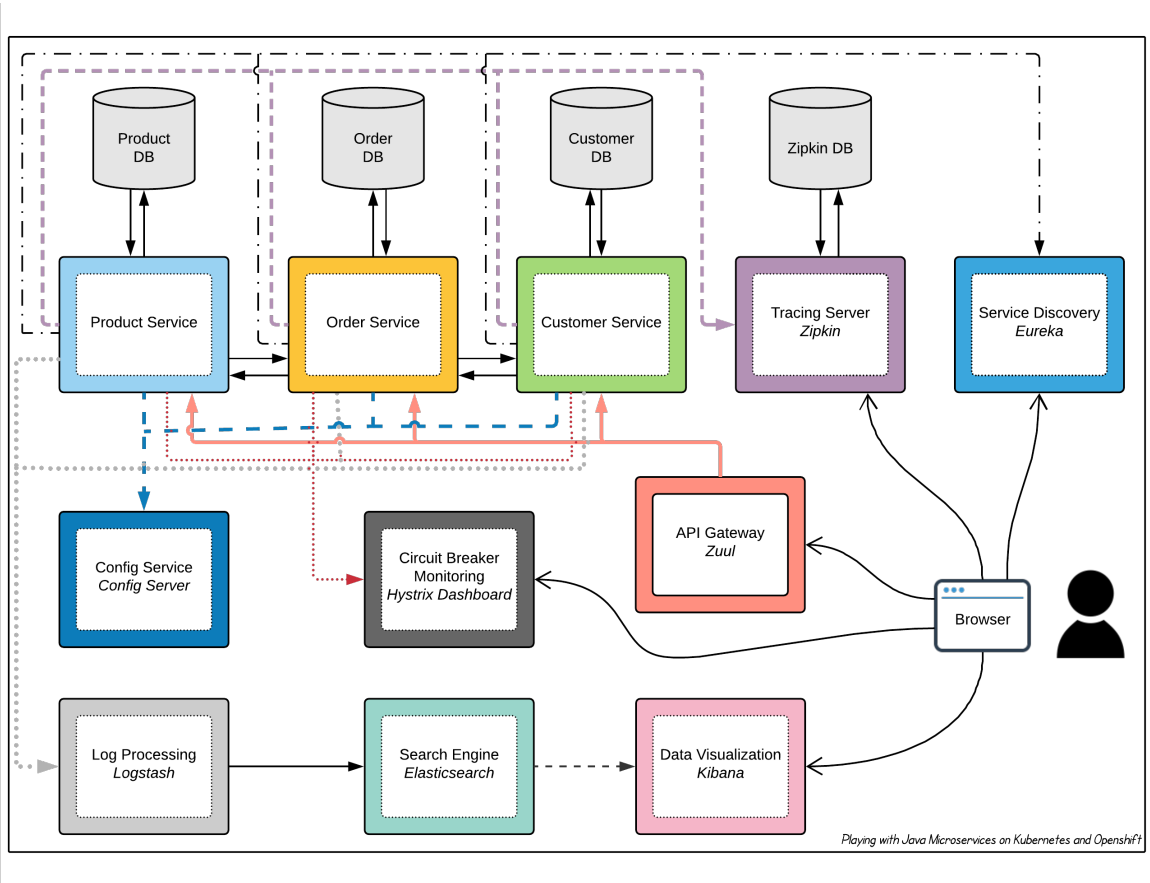
Chapter 10: Building the standalone microservices

Introduction

The Domain is splitted, the patterns are spotted, the libraries are picked up.. every thing is ready to start implementing our standalone microservices.

Global Architecture Big Picture

Here is our Architecture Landscape.



Global Architecture Big Picture

In the previous chapter, we covered:

- The Service Discovery with Eureka

- The API Manager with Zuul
- The central Config Server with Spring Cloud Config Server
- The log aggregation and analyzing with Elasticsearch + Logstash + Kibana
- The Tracing Server with Zipkin
- The Circuit Breaker with Hystrix / Monitoring with Hystrix Dashboard

We need now to implement our standalone microservices.

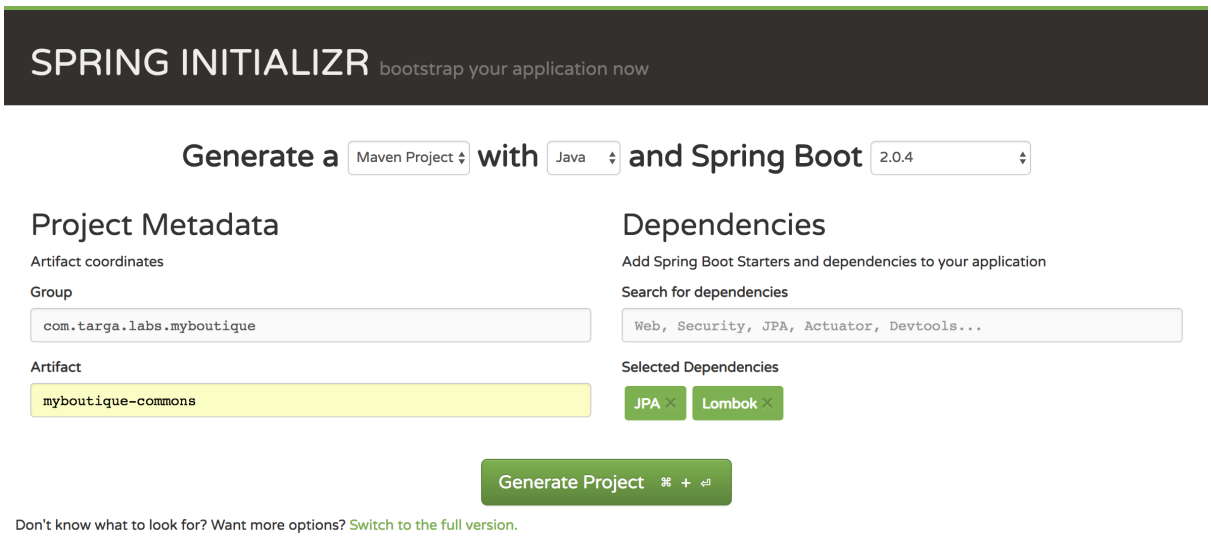
Implementing the µServices

Before starting

Before starting, we need to create the `myboutique-commons` project, which is a JAR library wrapping the common objects
used by our microservices.

Step 1: Generating the Commons project skull

We will create `myboutique-commons`: a new Spring Boot application using Spring Initializr[24]



Generating the Myboutique Commons project

We need to add the following dependencies:

- JPA
- Lombok

Step 2: Moving Code from our monolith

Then, we need to move the code from the `com.targa.labs.myboutique.commons` package of the monolith, to our `myboutique-commons` project.

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

Step 3: Moving Code from our monolith

After moving the code, you will get some errors in the `dto` package. That's caused by a missing dependency. We can find and add it using the IDE. Or you can add it manually in your `pom.xml`:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
</dependency>
```

Step 4: Building the project

We need to build this project at least once, so it can be available in our Maven local repository, to be used by our microservices:

```
mvn clean install
```

The build will fail :) don't be sad, this is not a problem. It's failing because of the test class, so you need to delete the `MyboutiqueCommonsApplicationTests` class and to delete also this Maven dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

I know many people will be scared because of these removals. But this project contains only DTOs and simple objects, and it doesn't contain no business logic code. That's why I think it's safe to ommit Testing here.

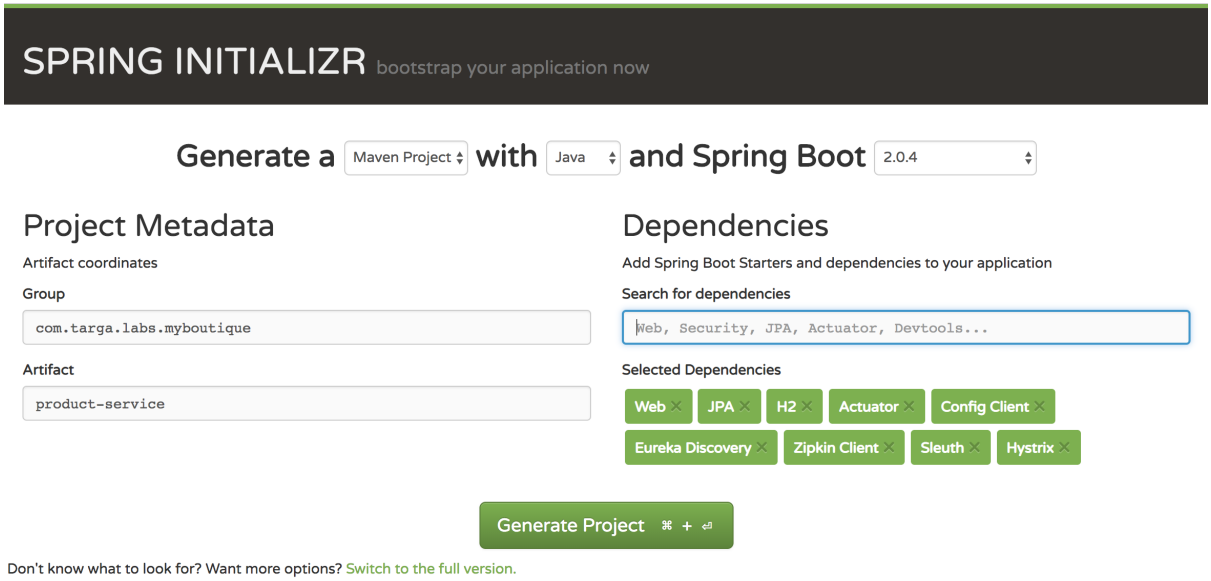
Now let's do another build using `mvn clean install` to install our JAR in our Maven local repository.

The Product Service

Step 1: Generating the Product Service project skull

We will create `product-service`: a new Spring Boot application using Spring Initializr[25]

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



Generating the Product Service project

We need to add the following dependencies:

- Web
- JPA
- H2
- Actuator
- Lombok
- Config Client
- Eureka Discovery
- Zipkin Client
- Sleuth
- Hystrix

Step 2: Swagger 2

Next we need to add the Swagger dependencies:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
```

Now we will activate the **Swagger2** capabilities, by annotating the **Main** Class with the `@EnableSwagger2` annotation.

Step 3: Application Configuration

We need a `bootstrap.yml` in our `product-service/src/main/resources/` folder:

product-service/src/main/resources/bootstrap.yml

```
spring:
  cloud:
    config:
      uri: http://localhost:8888
  application:
    name: product-service
```

And in the `Config Server`, we need to create a dedicated properties file of our `product-service`. This properties file will contain the port on which the `product-service` will be running on.

The content of the `product-service.yml` file:

config-server/src/main/resources/configurations/product-service.yml

```
server:
    port: 9990
```

Step 4: Logback

Next we need the Logback dependencies:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>5.2</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
</dependency>
```

Now we need to create a **LoggingConfiguration** Class that applies the Logback settings:

LoggingConfiguration.java

```
@Configuration
public class LoggingConfiguration {

    private static final String LOGSTASH_APPENDER_NAME = "LOGSTASH";
    private static final String ASYNC_LOGSTASH_APPENDER_NAME = "ASYNC_LOGSTASH";

    private final Logger LOG = LoggerFactory.getLogger(LoggingConfiguration.class);
    private final LoggerContext CONTEXT = (LoggerContext) LoggerFactory.getILoggerFactory();

    private final String appName;
    private final String logstashHost;
    private final Integer logstashPort;
    private final Integer logstashQueueSize;

    public LoggingConfiguration(
            @Value("${spring.application.name}") String appName,
            @Value("${logstash.host}") String logstashHost,
            @Value("${logstash.port}") Integer logstashPort,
            @Value("${logstash.queue-size}") Integer logstashQueueSize) {

        this.appName = appName;
        this.logstashHost = logstashHost;
        this.logstashPort = logstashPort;
        this.logstashQueueSize = logstashQueueSize;

        addLogstashAppender(CONTEXT);

    }

    private void addLogstashAppender(LoggerContext context) {
        LOG.info("Initializing Logstash logging");

        LogstashTcpSocketAppender logstashAppender = new LogstashTcpSocketAppender();
        logstashAppender.setName(LOGSTASH_APPENDER_NAME);
        logstashAppender.setContext(context);
        String customFields = "{\"servicename\":\"" + this.appName + "\"}";

        // More documentation is available at: https://github.com/logstash/logstash-logback-encoder
        LogstashEncoder logstashEncoder = new LogstashEncoder();
        // Set the Logstash appender config
        logstashEncoder.setCustomFields(customFields);
        // Set the Logstash appender config
        logstashAppender.addDestinations(
                new InetSocketAddress(this.logstashHost, this.logstashPort)
        );

        ShortenedThrowableConverter throwableConverter = new ShortenedThrowableConverter();
        throwableConverter.setRootCauseFirst(true);
        logstashEncoder.setThrowableConverter(throwableConverter);
        logstashEncoder.setCustomFields(customFields);

        logstashAppender.setEncoder(logstashEncoder);
        logstashAppender.start();

        // Wrap the appender in an Async appender for performance
        AsyncAppender asyncLogstashAppender = new AsyncAppender();
```

```
        asyncLogstashAppender.setContext(context);
        asyncLogstashAppender.setName(ASYNC_LOGSTASH_APPENDER_NAME);
        asyncLogstashAppender.setQueueSize(this.logstashQueueSize);
        asyncLogstashAppender.addAppender(logstashAppender);
        asyncLogstashAppender.start();

        context.getLogger("ROOT").addAppender(asyncLogstashAppender);
    }

}
```

Step 5: Activating the Circuit Beaker capability

To activate the **Circuit Beaker** capabilitiy, we have to annotate the **Main** Class with the `@EnableCircuitBreaker` annotation.

Step 6: Moving Code from our monolith

Now, we will move the content of the package `com.targa.labs.myboutique.product` in the **monolith**. This package matches the **Product Bounded Context**.

While moving the code from a project to another, most of IDEs, NetBeans here in our case, suggest doing some **Refactoring** of the code. This actions meaning doing something automatically, for example, renaming packages.

After moving and refactoring the code, there will be some errors due of the missing `myboutique-commons` dependency. So we need to add this dependency to our `pom.xml`:

```
<dependency>
    <groupId>com.targa.labs.myboutique</groupId>
    <artifactId>myboutique-commons</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

You can now build the `product-service`:

```
mvn clean install
```

Try to run the `product-service`:

```
mvn spring-boot:run
```

> Requirements
>
> You need to have to run the **Config Server** and the **Eureka Server** before running the `product-service`.

The Order Service

Step 1: Generating the Order Service project skull

We will create `order-service`: a new Spring Boot application using Spring Initializr[26]

We need to add the following dependencies:

- Web
- JPA
- H2
- Actuator
- Lombok
- Config Client
- Eureka Discovery
- Zipkin Client
- Sleuth
- Hystrix

Step 2: Swagger 2

Next we need to add the Swagger dependencies. We will activate the **Swagger2** capabilities, by annotating the **Main** Class with the `@EnableSwagger2` annotation.

Step 3: Application Configuration

We need a `bootstrap.yml` in our `order-service/src/main/resources/` folder:

order-service/src/main/resources/bootstrap.yml

```
spring:
  cloud:
    config:
      uri: http://localhost:8888
  application:
    name: order-service
```

And in the `Config Server`, we need to create a dedicated properties file of our `order-service`. This properties file will contain the port on which the `order-service` will be running on.

The content of the `order-service.yml` file:

config-server/src/main/resources/configurations/order-service.yml

```
server:
    port: 9991
```

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

Step 4: Logback

Next we need the Logback dependencies and the **LoggingConfiguration** Class that applies the Logback settings.

Step 5: Activating the Circuit Beaker capability

To activate the **Circuit Beaker** capabilitiy, we have to annotate the **Main** Class with the `@EnableCircuitBreaker` annotation.

Step 6: Moving Code from our monolith

Now, we will move the content of the package `com.targa.labs.myboutique.order` in the **monolith**. This package matches the **Order Bounded Context**.

While moving the code from a project to another, most of IDEs, NetBeans here in our case, suggest doing some **Refactoring** of the code. This actions meaning doing something automatically, for example, renaming packages.

After moving and refactoring the code, there will be some errors due of the missing `myboutique-commons` dependency. So we need to add this dependency to our `pom.xml`:

```
<dependency>
    <groupId>com.targa.labs.myboutique</groupId>
    <artifactId>myboutique-commons</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

You can now build the `order-service`:

```
mvn clean install
```

Try to run the `order-service`:

```
mvn spring-boot:run
```

The Customer Service

Step 1: Generating the Customer Service project skull

We will create `customer-service`: a new Spring Boot application using Spring Initializr[27]

We need to add the following dependencies:

- Web
- JPA
- H2

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

- Actuator
- Lombok
- Config Client
- Eureka Discovery
- Zipkin Client
- Sleuth
- Hystrix

Step 2: Swagger 2

Next we need to add the Swagger dependencies. We will activate the **Swagger2** capabilities, by annotating the **Main** Class with the `@EnableSwagger2` annotation.

Step 3: Application Configuration

We need a `bootstrap.yml` in our `customer-service/src/main/resources/` folder:

customer-service/src/main/resources/bootstrap.yml

```
spring:
  cloud:
    config:
      uri: http://localhost:8888
  application:
    name: customer-service
```

And in the `Config Server`, we need to create a dedicated properties file of our `customer-service`. This properties file will contain the port on which the `customer-service` will be running on.

The content of the `customer-service.yml` file:

config-server/src/main/resources/configurations/customer-service.yml

```
server:
    port: 9992
```

Step 4: Logback

Next we need the Logback dependencies and the **LoggingConfiguration** Class that applies the Logback settings.

Step 5: Activating the Circuit Beaker capability

To activate the **Circuit Beaker** capabilitiy, we have to annotate the **Main** Class with the `@EnableCircuitBreaker` annotation.

Step 6: Moving Code from our monolith

Now, we will move the content of the package `com.targa.labs.myboutique.customer` in the **monolith**. This package matches the **Customer Bounded Context**.

While moving the code from a project to another, most of IDEs, NetBeans here in our case, suggest doing some **Refactoring** of the code. This actions meaning doing something automatically, for example, renaming packages.

After moving and refactoring the code, there will be some errors due of the missing `myboutique-commons` dependency. So we need to add this dependency to our `pom.xml`:

```
<dependency>
    <groupId>com.targa.labs.myboutique</groupId>
    <artifactId>myboutique-commons</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

In the `CartService` class, we are calling the `create` method of the `OrderService`. We already know that the `OrderService` is an object that doesn't belong to our **Customer Bounded Context**:

CartService.java

```
@RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class CartService {

    private final CartRepository cartRepository;
    private final CustomerRepository customerRepository;
    private final OrderServiceClient orderService;

    ...

    public CartDto create(Long customerId) {
        if (this.getActiveCart(customerId) == null) {
            Customer customer = this.customerRepository.findById(customerId)
                    .orElseThrow(() -> new IllegalStateException("The Customer does not exist!"));

            Cart cart = new Cart(
                    customer,
                    CartStatus.NEW
            );

            OrderDto order = this.orderService.create(mapToDto(cart));
            cart.setOrderId(order.getId());

            return mapToDto(this.cartRepository.save(cart));
        } else {
            throw new IllegalStateException("There is already an active cart");
        }
    }

    ...
}
```

So we need to replace the direct call with a HTTP call from the `customer-service` to the `order-service`.

We will be using **Feign** instead of **RestTemplate**.

Feign is a **Spring Cloud Netflix** library for providing a higher level of abstraction over REST-based service calls. **Spring Cloud Feign** works on a declarative principle.
When using Feign, we write declarative REST service interfaces at the client, and use those interfaces to program the client. The developer need not worry about the implementation of this interface. This will be dynamically provisioned by Spring at runtime. With this declarative approach, developers need not get into the details of the HTTP level APIs provided by `RestTemplate`.

To be able to use **Feign**, we need to add its **Maven** dependency to our pom.xml:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

We will create a new class `OrderServiceClient` that we will be using instead of the `OrderService` class:

OrderServiceClient.java

```
@FeignClient(name = "order-service") <1>
public interface OrderServiceClient {

    @RequestMapping(value = "/api/orders", method = POST) <2>
    OrderDto create(CartDto cartDto); <2>
}
```

1. Ribbon will look for an entry in Eureka Registry and translate it to the proper hostname or ip and port.
2. The method that will be requesting the REST webservice

So the `CartService` class will look like :

CartService.java

```
@RequiredArgsConstructor
@Slf4j
@Service
@Transactional
public class CartService {

    private final CartRepository cartRepository;
    private final CustomerRepository customerRepository;
    private final OrderServiceClient orderService;

    ...
```

```
    public CartDto create(Long customerId) {
        if (this.getActiveCart(customerId) == null) {
            Customer customer = this.customerRepository.findById(customerId)
                    .orElseThrow(() -> new IllegalStateException("The Customer does not exist!"));

            Cart cart = new Cart(
                    customer,
                    CartStatus.NEW
            );

            OrderDto order = this.orderService.create(mapToDto(cart));
            cart.setOrderId(order.getId());

            return mapToDto(this.cartRepository.save(cart));
        } else {
            throw new IllegalStateException("There is already an active cart");
        }
    }

    ...
}
```

You can now build the `customer-service`:

```
mvn clean install
```

Try to run the `customer-service`:

```
mvn spring-boot:run
```

Conclusion

In the these two chapters, we developped many components:

- Eureka Server
- Config Server
- API Gateway
- Hystrix Dashboard
- Product Service
- Order Service
- Customer Service
- MyBoutique commons JAR

Every component in this list needs to be delivered, deployed and monitored separately. Our microservices will be built and deployed as WARs or JARs to runtime environments. In a CI/CD sense, we will have dedicated pipeline by component. We will deal with this in the incoming chapters.

Part Three: Containers & Cloud Era

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.

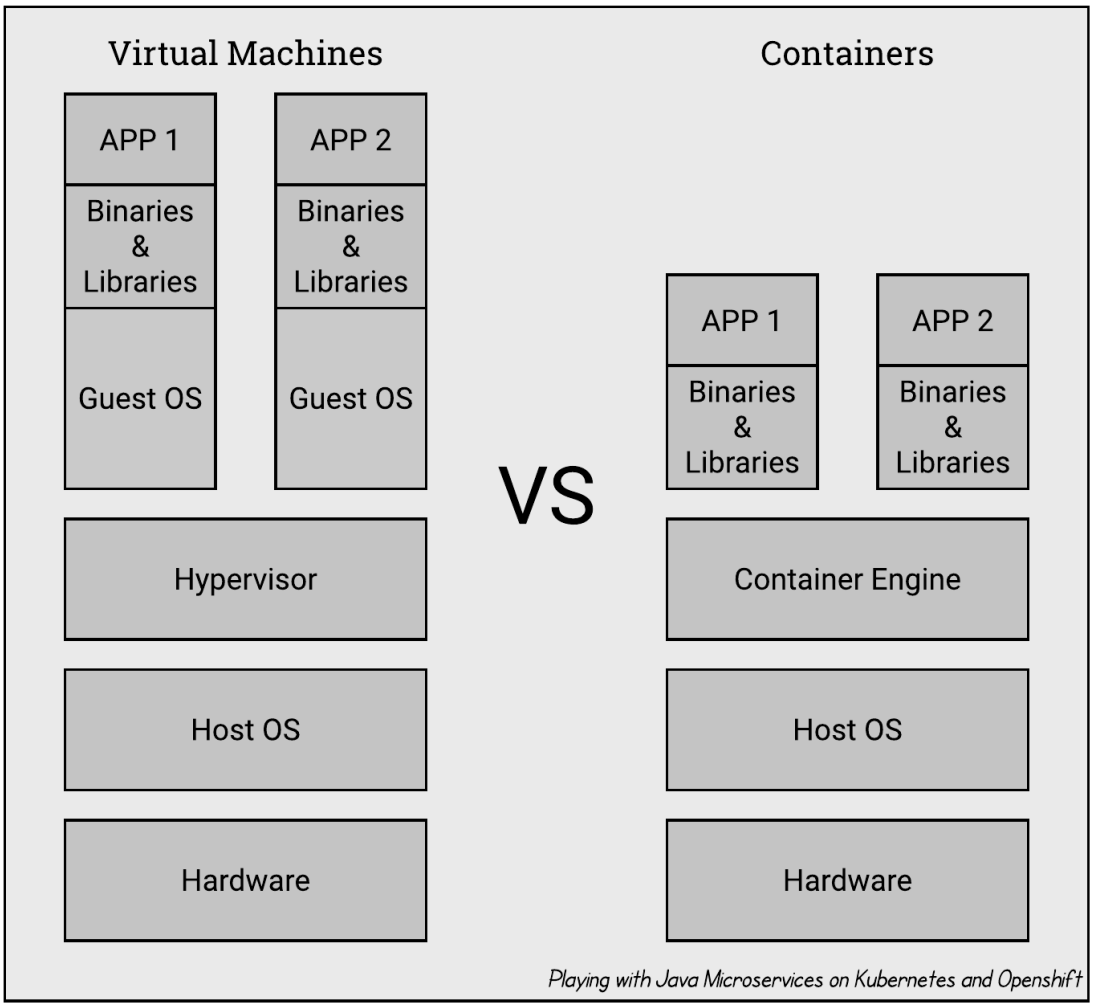
The major risk that encounters software application, is updates of the runtime that can break the application. For example, an OS update might include many updates, including libraries that can affect the running application with incompatible updates.

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/[28];

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`:

```
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.

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

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**.

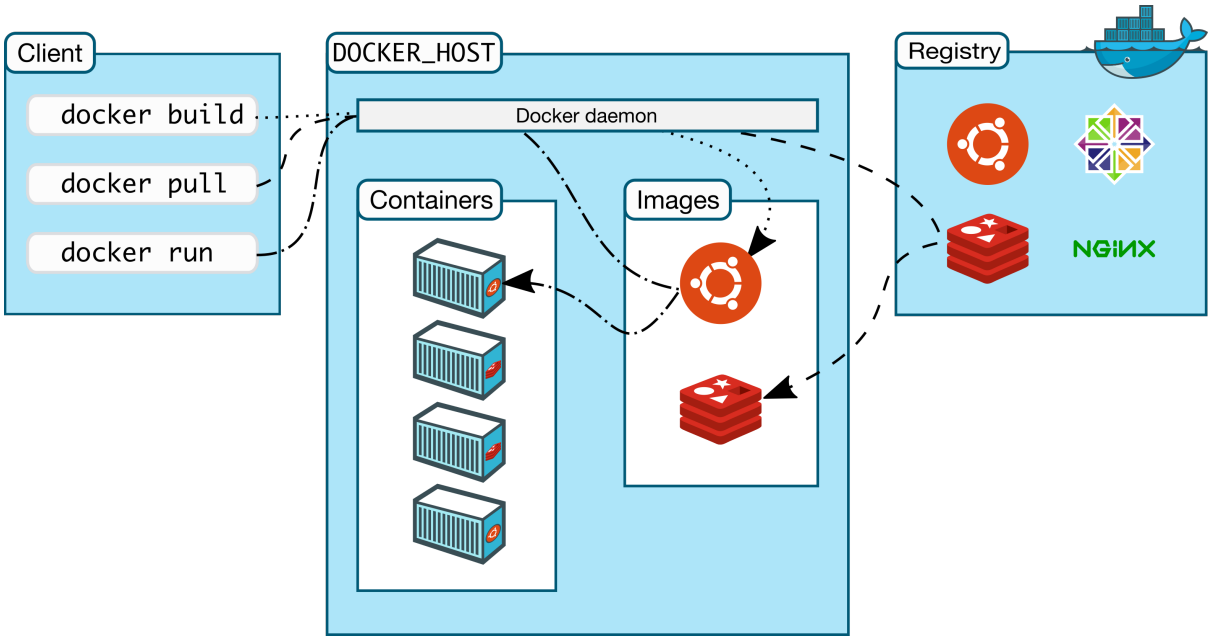
To list all the Docker containers:

```
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[29] 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!";
    }
}
```

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:

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

```
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.

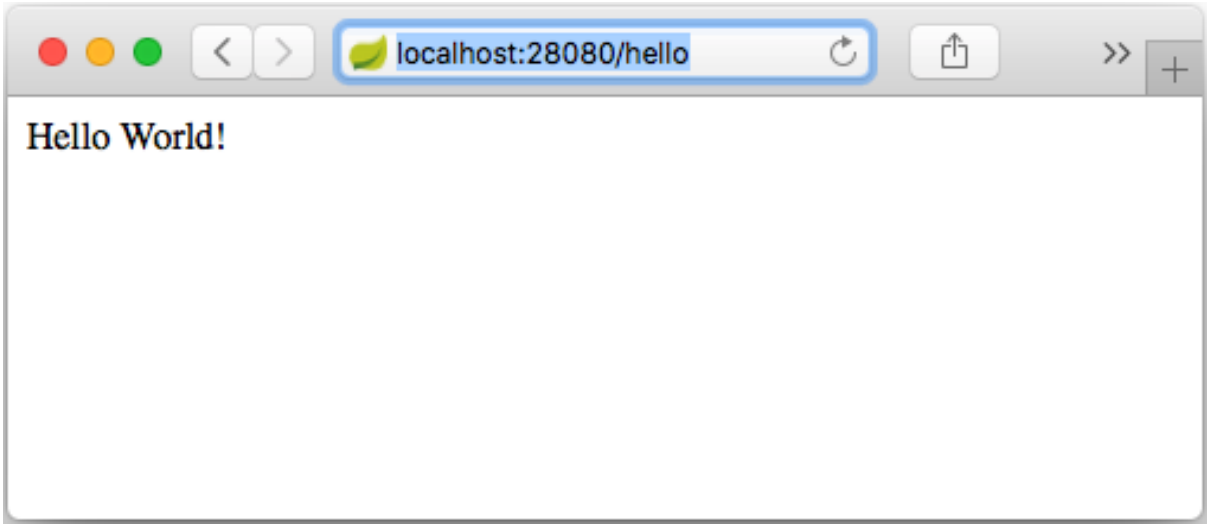
Run the app

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 -Djava.securit…"   47 seconds ago Up 46 seconds   0.0.0.0:28080-\
>8080/tcp   sharp_engelbart
```

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/[30]. Make note of your username.

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

```
docker login
```

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:

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

```
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
298c3bb2664f: 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
391a6a6b3651: 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.

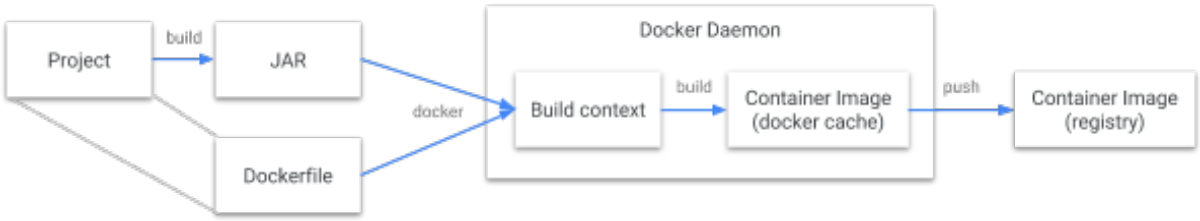
You can also specify authentication details in Maven `settings.xml` or `pom.xml` files.

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[31]

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>
```

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:

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

```
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   *:2808\
0->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 eurek\
a.datacenter is not set, defaulting to default
ERROR 1 --- [nfoReplicator-0] c.n.d.s.t.d.RedirectingEurekaHttpClient  : Request execution error

com.sun.jersey.api.client.ClientHandlerException: java.net.ConnectException: Connection refused (Conne\
ction 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/e\
ureka: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-SNAP\
SHOT
```

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_ZIP-
  KIN_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_ZIP-
  KIN_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_ZIP-
  KIN_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

To deploy our (so huge, so big) application), we will be using Docker. We will deploy our code in a container so we can enjoy the great feature provided by Docker.

Docker has become the standard to develop and run containerized applications.

This is great ! Using Docker is quitely simple, especially in development stages. Deploying containers in the same server (docker-machine) is simple, but when start thinking to deploy many containers to many servers, things become complicated (managing the servers, managing the container state, ...).

Here come the orchestration system, which provides many features like orchestrating computing, networking and storage infrastructure on behalf of user workloads:

- Scheduling: matching containers to machines based on many factors like resources needs, affinity requirements...
- Replications
- Handling failures
- Etc...

For our tutorial, we will choose Kubernetes, the star of container orchestration.

What is Kubernetes ?

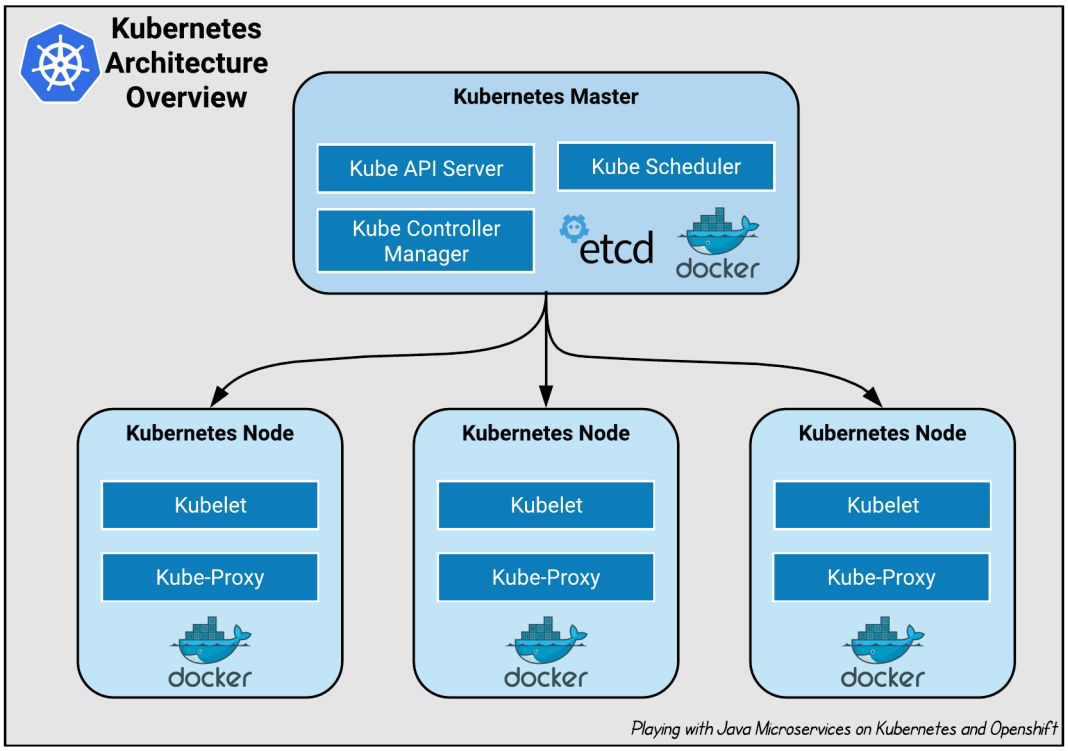
Kubernetes (Aka K8s) was a project spun out of Google as a open source next-gen container scheduler designed with the lessons learned from developing and managing Borg and Omega.

Kubernetes is designed to have loosely coupled components centered around deploying, maintaining and scaling applications. K8s abstracts the underlying infrastructure of the nodes and provides a uniform layer for the deployed applications.

Kubernetes Architecture

In the big plan, a Kubernetes cluster is composed of two items:

- Master Nodes: The main control plane for Kubernetes. It contains an API Server, a Scheduler, a Controller Manager (K8s cluster manager) and a datastore to save the cluster state called **Etcd**.
- Worker Nodes: A single host, physical or virual machine, capable of running **POD**. They are managed by the Master nodes.



Kubernetes Architecture Overview

Let's have a look inside a Master node:

- (Kube) API-Server: allows the communication, thru REST APIs, between the Master node and all its clients such as Worker Nodes, kube-cli, ...
- (Kube) Scheduler: a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity to assign a Node to a newly created **POD**.
- (Kube) Controller Manager: a daemon that embeds the core control loops shipped with Kubernetes. A control loop is a permenent listener that regulates the state of the system. In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the API-Server and makes changes attempting to move the current state towards the desired state.
- Etcd: a strong, consistent and highly available key-value store used for persisting the cluster state.

Then, what about a Worker node?

- Kubelet: an agent that runs on each node in the cluster. It makes sure that containers are running in a **POD**.
- Kube-Proxy: enables the Kubernetes service abstraction by maintaining network rules on the host and perform networking actions.

> **Tip**
>
> The Container Runtime that we will user is Docker. Kubernetes is compatible with many others like Cri-o, Rkt, ...

Kubernetes Core Concepts

The K8s ecosystem covers many concepts and components. We will try to introduce them briefly.

Kubectl

The `kubectl` is a command line interface for running commands against Kubernetes clusters.

Cluster

A collection of hosts that aggregate their resources (CPU, Ram, Disk, ...) into a usable pool.

Namespace

A logical partitioning capability that enable one Kubernetes cluster to be used by multiple users, teams of users, or a single user with multiple applications without concern for undesired interaction. Each user, team of users, or application may exist within its Namespace, isolated from every other user of the cluster and operating as if it were the sole user of the cluster.

List all Namespace: `kubectl get namespace` or `kubectl get ns`

Label

Key-value paris that are used to identify and select related sets of objects. Labels have a strict syntax and defined character set.

Annotation

Key-value pairs that contain non-identifying information or metadata. Annotations do not have the the syntax limitations as labels and can contain structured or unstructured data.

Selector

Selectors use labels to filter or select objects. Both equality-based (=, ==, !=) or simple key-value matching selectors are supported.

Use case of Annotations, Labels and Selectors.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Pod

It is the basic unit of work for Kubernetes. Represent a collection of containers that share resources, such as IP addresses and storage.

Pod Example.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

List all Pods: `kubectl get pod` or `kubectl get po`

ReplicationController

A framework for defining pods that are meant to be horizontally scaled. A replication controller includes a pod definition that is to be replicated, and the pods created from it can be scheduled to different nodes.

ReplicationController Example.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
```

```
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

List all ReplicationControllers: `kubectl get replicationcontroller` or `kubectl get rc`

ReplicaSet

An upgraded version of ReplicationController that supports set-based selectors.

ReplicaSet Example.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        env:
        - name: GET_HOSTS_FROM
          value: dns
        ports:
        - containerPort: 80
```

List all ReplicaSets: `kubectl get replicaset` or `kubectl get rs`

Deployment

Includes a Pod template and a replicas field. Kubernetes will make sure the actual state (amount of replicas, Pod template) always matches the desired state. When you update a Deployment it will perform a “rolling update”.

Deployment Example.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

List all Deployments: `kubectl get deployment`

StatefulSet

A controller tthat aims to manage Pods that must persist or maintain state. Pod identity including hostname, network, and storage will be persisted.

StatefulSet Example.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
```

```
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

List all StatefulSets: `kubectl get statefulset`

DaemonSet

Ensures that an instance of a specific pod is running on all (or a selection of) nodes in a cluster.

DaemonSet Example.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: gcr.io/google-containers/fluentd-elasticsearch:1.20
      terminationGracePeriodSeconds: 30
```

List all DaemonSets: `kubectl get daemonset` or `kubectl get ds`

Service

Define a single IP/port combination that provides access to a group of pods. It uses label selectors to map groups of pods and ports to a cluster-unique virtual IP.

Service Example.

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

List all Services: `kubectl get service` or `kubectl get svc`

Ingress

An ingress controller is the primary method of exposing a cluster service (usually http) to the outside world. These are load balancers or routers that usually offer SSL termination, name-based virtual hosting etc…

Ingress Example.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        backend:
          serviceName: test
          servicePort: 80
```

List all Ingress: `kubectl get ingress`

Volume

Storage that is tied to the Pod Lifecycle, consumable by one or more containers within the pod.

PersistentVolume

A PersistentVolume (PV) represents a storage resource. PVs are commonly linked to a backing storage resource, NFS, GCEPersistentDisk, RBD etc. and are provisioned ahead of time. Their lifecycle is handled independently from a pod.

List all PersistentVolumes: `kubectl get persistentvolume` or `kubectl get pv`

PersistentVolumeClaim

A PersistentVolumeClaim (PVC) is a request for storage that satisfies a set of requirements. Commonly used with dynamically provisioned storage.

List all PersistentVolumeClaims: `kubectl get persistentvolumeclaim` or `kubectl get pvc`

StorageClass

Storage classes are an abstraction on top of an external storage resource. These will include a provisioner, provisioner configuration parameters as well as a PV reclaimPolicy.

List all StorageClasses: `kubectl get storageclass` or `kubectl get sc`

Job

The job controller ensures one or more pods are executed and successfully terminates. It will do this until it satisfies the completion and/or parallelism condition.

List all Jobs: `kubectl get job`

CronJob

An extension of the Job Controller, it provides a method of executing jobs on a cron-like schedule.

List all CronJobs: `kubectl get cronjob`

ConfigMap

Externalized data stored within Kubernetes that can be referenced as a commandline argument, environment variable or injected as a file into a volume mount. Ideal for implementing the External Configuration Store pattern.

List all ConfigMaps: `kubectl get configmap` or `kubectl get cm`

Secret

Functionally identical to ConfigMaps, but stored encoded as base64, and encrypted at rest (if configured).

List all Secrets: `kubectl get secret`

Run Kubernetes locally

For our tutorial we will not build a real Kubernetes Cluster. We will use **Minikube**.

Minikube is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a VM on your laptop for users looking to try out Kubernetes or develop with it day-to-day.

For Minikube installation : https://github.com/kubernetes/minikube

After the installation, to start Minikube:

```
minikube start
```

The `minikube start` command creates a **kubectl context** called **minikube**. This context contains the configuration to communicate with your minikube cluster.

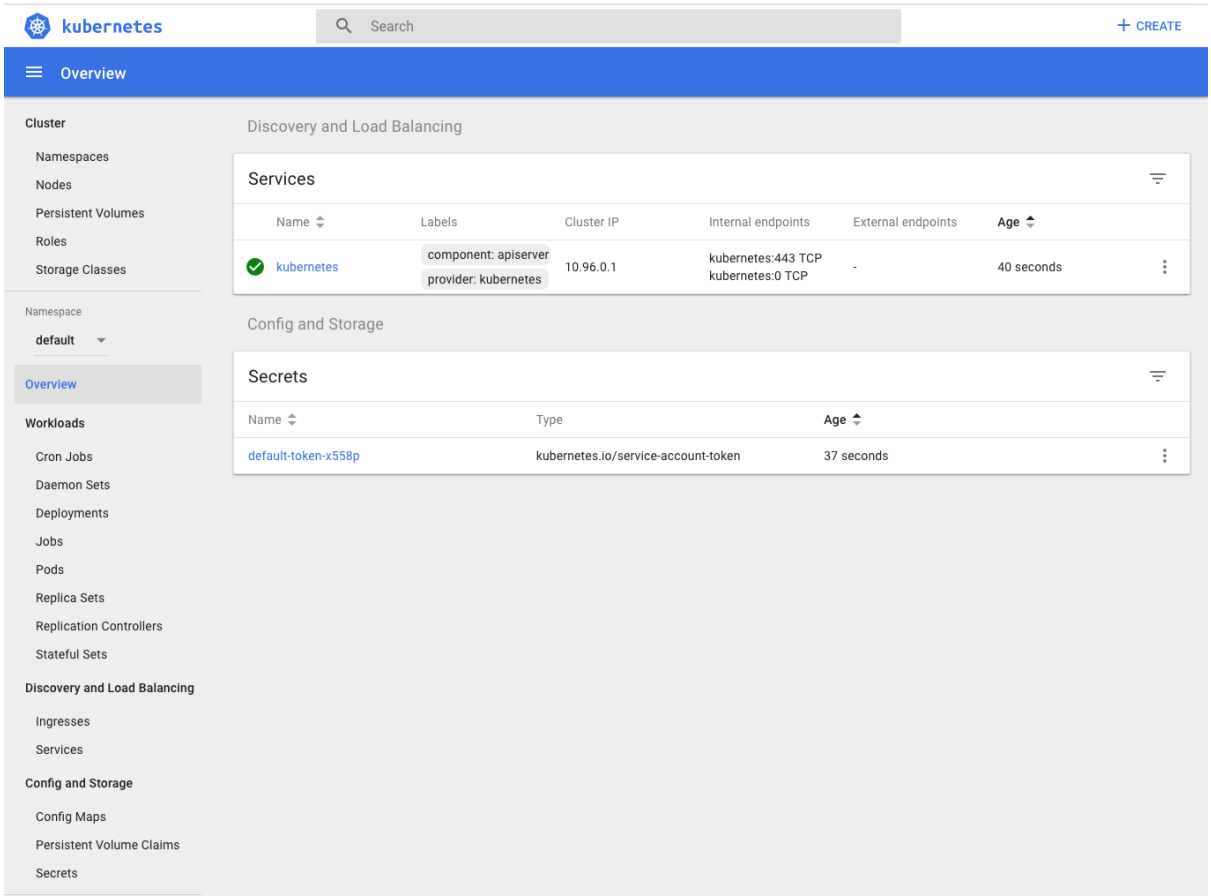
Minikube sets this context to default automatically, but if you need to switch back to it in the future, run:

```
kubectl config use-context minikube
```

To access the Kubernetes Dashboard:

```
minikube console
```

The Dashboard will be opened in your default browser:



Kubernetes Dashboard (Web UI)

The `minikube stop` command can be used to stop your cluster. This command shuts down the minikube virtual machine, but preserves all cluster state and data. Starting the cluster again will restore it to it's previous state.

The `minikube delete` command can be used to delete your cluster. This command shuts down and deletes the minikube virtual machine. No data or state is preserved.

Chapter 13: The Kubernetes style

Now we want to move from H2 Databse to PostgreSQL. So we have to configure the application to use it by mentioning some properties like JDBC driver, url, username, password... in the `application.properties` file and to add the PostgreSQL JDBC Driver in the `pom.xml`.

Discovering the Kubernetes style

First of all, we start by adding this dependency to our `pom.xml`:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Next, we do some modifications to the `application.properties`:

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:postgresql://${POSTGRES_SERVICE}:5432/${POSTGRES_DB_NAME}
spring.datasource.username=${POSTGRES_DB_USER}
spring.datasource.password=${POSTGRES_DB_PASSWORD}
```

We have used environment properties placeholders:

- *POSTGRES_SERVICE* : Host of PostgreSQL DB Server
- *POSTGRES_DB_NAME* : PostgreSQL DB Name
- *POSTGRES_DB_USER* : PostgreSQL Username
- *POSTGRES_DB_PASSWORD* : PostgreSQL Password

We will extract these values from a Kubernetes ConfigMap and Secret objects.

Create the ConfigMap

We need to create the ConfigMap:

```
kubectl create configmap postgres-config \
    --from-literal=postgres.service.name=postgresql \
    --from-literal=postgres.db.name=boutique
```

We can check the created ConfigMap:

```
kubectl get cm postgres-config -o json
```

The output will look like this:


```
{
    "apiVersion": "v1",
    "data": {
        "postgres.db.name": "boutique",
        "postgres.service.name": "postgresql"
    },
    "kind": "ConfigMap",
    "metadata": {
        "creationTimestamp": "2018-03-25T16:42:39Z",
        "name": "postgres-config",
        "namespace": "default",
        "resourceVersion": "195",
        "selfLink": "/api/v1/namespaces/default/configmaps/postgres-config",
        "uid": "87d7481c-304b-11e8-889d-080027a8a37c"
    }
}
```


Create the Secret

Next we create the Secret:

```
kubectl create secret generic db-security \
    --from-literal=db.user.name=nebrass \
    --from-literal=db.user.password=password
```

We can check the created Secret:

```
kubectl get secret db-security -o json
```

The output will look like this:


```
{
    "apiVersion": "v1",
    "data": {
        "db.user.name": "bmVicmFzcw==",
        "db.user.password": "cGFzc3dvcmQ="
    },
    "kind": "Secret",
    "metadata": {
        "creationTimestamp": "2018-03-25T16:56:36Z",
        "name": "db-security",
        "namespace": "default",
        "resourceVersion": "714",
        "selfLink": "/api/v1/namespaces/default/secrets/db-security",
        "uid": "7ac96df3-304d-11e8-889d-080027a8a37c"
    },
    "type": "Opaque"
}
```

- The credentials are encoded as base64. This is to protect the secret from being exposed accidentally to someone looking or from being stored in a terminal log.
- From kubernetes's point of view the contents of this Secret is unstructured: it can contain arbitrary key-value pairs.

Deploy PostgreSQL to Kubernetes

As the configuration is centralized and stored in the Kubernetes Cluster, we can share them between the Spring Boot Application and the PostgreSQL Service that we will create now.

I already prepared the PostgreSQL resource file, in the `src/main/assets/`. This YAML file contains a Deployment and a Service resources.

We loaded the properties from our ConfigMap and Secret.

The content of `postgres.yml`:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: postgresql
  namespace: default
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: postgresql
    spec:
      volumes:
      - name: data
        emptyDir: {}
      containers:
      - name: postgres
        image: postgres:9.6.5
        env:
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: db-security
              key: db.user.name
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-security
              key: db.user.password
        - name: POSTGRES_DB
          valueFrom:
            configMapKeyRef:
              name: postgres-config
              key: postgres.db.name
        ports:
        - containerPort: 5432
        volumeMounts:
        - name: data
```

```
          mountPath: /var/lib/postgresql/
---
apiVersion: v1
kind: Service
metadata:
  name: postgresql
  namespace: default
spec:
  selector:
    app: postgresql
  ports:
  - port: 5432
```

- We will use the postgres:9.6.5 image
- The `env` block is used to load data in the container environment.
- Create an environment variable with a value loaded from a `key` called `db.user.name` in the `secret` called `db-security`.
- Create an environment variable with a value loaded from a `key` called `db.user.password` in the `secret` called `db-security`.
- Create an environment variable with a value loaded from a `key` called `postgres.db.name` in the `configMap` called `postgres-config`.

To apply this resource file to Kubernetes, we can do:

```
kubectl create -f src/main/assets/postgres.yml
```

The output will be:

```
deployment "postgresql" created
service "postgresql" created
```

We can check the created Deployment:

```
kubectl get deployment postgresql -o json
```

The output will look like this:


```
{
    "apiVersion": "extensions/v1beta1",
    "kind": "Deployment",
    "metadata": {
        "annotations": {
            "deployment.kubernetes.io/revision": "1"
        },
        "creationTimestamp": "2018-03-25T17:04:03Z",
        "generation": 1,
        "labels": {
            "app": "postgresql"
        },
        "name": "postgresql",
        "namespace": "default",
        "resourceVersion": "1025",
```

```
        "selfLink": "/apis/extensions/v1beta1/namespaces/default/deployments/postgresql",
        "uid": "84fa2f1a-304e-11e8-889d-080027a8a37c"
    },
    "spec": {
        ...
    },
    "status": {
        "availableReplicas": 1,
        "conditions": [
            {
                "lastTransitionTime": "2018-03-25T17:04:03Z",
                "lastUpdateTime": "2018-03-25T17:04:03Z",
                "message": "Deployment has minimum availability.",
                "reason": "MinimumReplicasAvailable",
                "status": "True",
                "type": "Available"
            }
        ],
        "observedGeneration": 1,
        "readyReplicas": 1,
        "replicas": 1,
        "updatedReplicas": 1
    }
}
```

We can check the created Service:

```
kubectl get service postgresql -o json
```

The output will look like this:

```
{
    "apiVersion": "v1",
    "kind": "Service",
    "metadata": {
        "creationTimestamp": "2018-03-25T17:04:03Z",
        "name": "postgresql",
        "namespace": "default",
        "resourceVersion": "1000",
        "selfLink": "/api/v1/namespaces/default/services/postgresql",
        "uid": "84fd14fa-304e-11e8-889d-080027a8a37c"
    },
    "spec": {
        "clusterIP": "10.104.62.217",
        "ports": [
            {
                "port": 5432,
                "protocol": "TCP",
                "targetPort": 5432
            }
        ],
        "selector": {
            "app": "postgresql"
        },
        "sessionAffinity": "None",
        "type": "ClusterIP"
```

```
    },
    "status": {
        "loadBalancer": {}
    }
}
```

In this output, there is something interesting: the port and the target port:

- The port this service will be available on
- The container port the service will forward to

We already mentionned the `port` in the `spring.datasource.url` property.

- What do you think if we use the powerful features of Kubernetes to resolve this port dynamically?
- Ok but how? :)

After creating these resources, the *effective* properties will like this :

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:postgresql://postgresql:5432/boutique
spring.datasource.username=nebrass
spring.datasource.password=password
```

The Datasource URL is pointing to a host called **postgresql**. The resolution of the hostname to IP is done by Kubernetes.

If we check the `postgresql` service:

```
kubectl get svc postgresql

NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
postgresql   ClusterIP   10.111.244.143   <none>        5432/TCP   1h
```

There is an other cool feature in Kubernetes, we can fetch data related to the service itself. We can for example get the port associated to this service.

For example:

- `${postgresql.service.host}` will be resolved to `10.111.244.143`.
- `${postgresql.service.port}` will be resolved to `5432`.

We can do it better) we can merge the environment variables in the great holders that will be resolved by Kubernetes. They will become:

- `${postgresql.service.host}` can be written `${${POSTGRES_SERVICE}.service.host}`
- `${postgresql.service.port}` can be written `${${POSTGRES_SERVICE}.service.port}`

In this way, the internal placeholder will be resolved by the environment variable provided by the ConfigMap, and the external placeholder will be resolved by Kubernetes.

The resulting `application.properties` will look like:

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:postgresql://${${POSTGRES_SERVICE}.service.host}:${${POSTGRES_SERVICE}.serv\
ice.port}/${POSTGRES_DB_NAME}
spring.datasource.username=${POSTGRES_DB_USER}
spring.datasource.password=${POSTGRES_DB_PASSWORD}
```

Now, we are ConfigMaps addicts :) we will host our `application.properties` in a ConfigMap in Kubernetes. To do it, just do:

```
kubectl create configmap app-config
    --from-file=src/main/resources/application.properties
```

Now that the `application.properties`, how can our Spring Boot application use them?

The answer is so easy: the **Spring Cloud Kubernetes** plugin.

What is Spring Cloud Kubernetes

The Spring Cloud Kubernetes plug-in implements the integration between Kubernetes and Spring Boot. It provides access to the configuration data of a ConfigMap using the Kubernetes API.

It make so easy to integrate Kubernetes ConfigMap directly with the Spring Boot externalized configuration mechanism, so that Kubernetes ConfigMaps behave as an alternative property source for Spring Boot configuration.

To enable the great features of the plugin:

1. Add the Maven Dependency:

   Add this dependency to the `pom.xml`:

   ```
   <dependency>
       <groupId>io.fabric8</groupId>
       <artifactId>spring-cloud-starter-kubernetes</artifactId>
       <version>0.1.6</version>
   </dependency>
   ```

2. Create the Bootstrap file:

   Create a new file `bootstrap.properties` under `src/main/resources`:

   ```
   spring.application.name=${project.artifactId}
   spring.cloud.kubernetes.config.name=app-config
   ```

   - The `${project.artifactId}` will be parsed and populated by `maven-resources-plugin`, that you will find in the `pom.xml` of the sample project hosted in the Github repository of this tutorial.

   - The name of the ConfigMap where we stored our great `application.properties`.

That's it! The Spring Cloud Kubernetes is correctly integrated to our application. When we deploy our application to Kubernetes, it will use the `application.properties` stored in the ConfigMap `app-config`.

You say deploy? Ok but how to do it?

Deploy it to Kubernetes

The deployment?! A dedicated full story, that can have many chapters. But we will try to keep it short and simple.

By definition, Kubernetes is a container orchestration solution. So deploying an application to Kubernetes means :

- Containerizing the application: creating an image embedding the application.
- Preparing the deployment resources (Deployment, ReplicaSet, etc...).
- Deploying the container to Kubernetes.

These steps can take some time to be done, even if we try to automate this process, it will take us long time to implement it, and it will take more time to cover all the cases and variants of the apps.

As these tasks are so heavy, we need some tool that do all of this for easy.

Here comes the super powerfull tool: **Fabric8-Maven-Plugin**.

Fabric8 Logo

Fabric8-Maven-Plugin is a one-stop-shop for building and deploying Java applications for Docker, Kubernetes and OpenShift. It brings your Java applications on to Kubernetes and OpenShift. It provides a tight integration into maven and benefits from the build configuration already provided. It focuses on three tasks:

- Building Docker images
- Creating OpenShift and Kubernetes resources
- Deploy application on Kubernetes and OpenShift

The plugin will do all the heavy tasks ! Yes ! He will ! :)

It can be configured very flexibly and supports multiple configuration models for creating:

- **Zero Configuration** for a quick ramp-up where opinionated defaults will be pre-selected.
- **Inline Configuration** within the plugin configuration in an XML syntax.
- **External Configuration** templates of the real deployment descriptors which are enriched by the plugin.
- **Docker Compose Configuration** provide Docker Compose file and bring up docker compose deployments on a Kubernetes/OpenShift cluster.

To enable *fabric8-maven-plugin* on your project just add this to the plugins sections of your `pom.xml`:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.41</version>
</plugin>
```

Now in order to use *fabric8-maven-plugin* to build or deploy, make sure you have anKubernetes cluster up and running.

The *fabric8-maven-plugin* supports a rich set of goals for providing a smooth Java developer experience. You can categorize these goals as follows:

- Build goals are used to create and manage the Kubernetes build artifacts like Docker images.
 - `fabric8:build` : Build Docker images
 - `fabric8:resource` : Create Kubernetes resource descriptors
 - `fabric8:push` : Push Docker images to a registry
 - `fabric8:apply` : Apply resources to a running cluster

- Development goals are used in deploying resource descriptors to the development cluster.
 - `fabric8:run` : Run a complete development workflow cycle `fabric8:resource` ⟶ `fabric8:build` ⟶ `fabric8:apply` in the foreground.
 - `fabric8:deploy` : Deploy resources descriptors to a cluster after creating them and building the app. Same as `fabric8:run except that it runs in the background.` - fabric8:undeploy `: Undeploy and remove resources descriptors from a cluster.` - fabric8:watch‘ : Watch for doing rebuilds and restarts

If you want to integrate the goals in the maven lifecycle phases, you can do it easily:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.41</version>

  <!-- This block will connect fabric8:resource and fabric8:build to lifecycle phases -->
  <executions>
    <execution>
       <id>fmp</id>
       <goals>
         <goal>resource</goal>
         <goal>build</goal>
       </goals>
    </execution>
  </executions>
</plugin>
```

Note

For lazyness purposes :p I will be referencing the *fabric8-maven-plugin* as **f8mp**.

Now when we do `mvn clean install` for example, the plugin will build the docker images and will generate the Kubernetes resource descriptors in the `${basedir}/target/classes/META-INF/fabric8/kubernetes` directory.

Let's check the generated resource descriptors.

> **Warning**
>
> Wait ! Wait! We said that we will pass the ConfigMaps to the Spring Boot application. Where is that?!

Yep! Before generating our resources descriptors, we have to tell this to **f8mp**.

f8mp has an easy way to do this: the plugin can handle some **Resource Fragments**. It's a piece of YAML code located in the `src/main/fabric8` directory. Each resource get is own file, which contains some skeleton of a resource description. The plugin will pick up the resource, enriches it and then combines all the data. Within these descriptor files you are can freely use any Kubernetes feature.

In our case, we will deliver in the **Resource Fragment** the configuration of the environment variables to the Pod, where our Spring Boot Application will be executed. We will use a **fragment** of a **Deployment**, which will look like this:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ${project.artifactId}
  namespace: default
spec:
  template:
    spec:
      containers:
        - name: ${project.artifactId}
          env:
          - name: POSTGRES_SERVICE
            valueFrom:
              configMapKeyRef:
                name: postgres-config
                key: postgres.service.name
          - name: POSTGRES_DB_NAME
            valueFrom:
              configMapKeyRef:
                name: postgres-config
                key: postgres.db.name
          - name: POSTGRES_DB_USER
            valueFrom:
              secretKeyRef:
                name: db-security
                key: db.user.name
          - name: POSTGRES_DB_PASSWORD
            valueFrom:
              secretKeyRef:
                name: db-security
                key: db.user.password
```

- The name of our Deployment and the container.
- The environment variables that we are creating and populating from the ConfigMap and Secret.

Now, when the **f8mp** will try to generate the resources descriptors, it will find this resource fragment, combine it with the other data. The resulting output will be coherent with the fregment that we already provided.

Let's try it. Just run `mvn clean install`:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building MyBoutique 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
...
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.41:resource (fmp) @ myboutique -
[INFO] F8: Running in Kubernetes mode
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using Docker image fabric8/java-jboss-openjdk8-jdk:1.3 as base / builder
[INFO] F8: using resource templates from /Users/n.lamouchi/MyBoutique/src/main/fabric8
[INFO] F8: fmp-service: Adding a default service 'myboutique' with ports [8080]
[INFO] F8: spring-boot-health-check: Adding readiness probe on port 8080, path='/health', scheme='HTTP\
', with initial delay 10 seconds
[INFO] F8: spring-boot-health-check: Adding liveness probe on port 8080, path='/health', scheme='HTTP'\
, with initial delay 180 seconds
[INFO] F8: fmp-revision-history: Adding revision history limit to 2
[INFO] F8: f8-icon: Adding icon for deployment
[INFO] F8: f8-icon: Adding icon for service
[INFO] F8: validating /Users/n.lamouchi/MyBoutique/target/classes/META-INF/fabric8/openshift/myboutiqu\
e-svc.yml resource
[INFO] F8: validating /Users/n.lamouchi/MyBoutique/target/classes/META-INF/fabric8/openshift/myboutiqu\
e-deploymentconfig.yml resource
[INFO] F8: validating /Users/n.lamouchi/MyBoutique/target/classes/META-INF/fabric8/openshift/myboutiqu\
e-route.yml resource
[INFO] F8: validating /Users/n.lamouchi/MyBoutique/target/classes/META-INF/fabric8/kubernetes/myboutiq\
ue-deployment.yml resource
[INFO] F8: validating /Users/n.lamouchi/MyBoutique/target/classes/META-INF/fabric8/kubernetes/myboutiq\
ue-svc.yml resource
[INFO]
...
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.41:build (fmp) @ myboutique -
[INFO] F8: Building Docker image in Kubernetes mode
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using Docker image fabric8/java-jboss-openjdk8-jdk:1.3 as base / builder
[INFO] Copying files to /Users/n.lamouchi/MyBoutique/target/docker/nebrass/myboutique/snapshot-180327-\
174802-0575/build/maven
[INFO] Building tar: /Users/n.lamouchi/MyBoutique/target/docker/nebrass/myboutique/snapshot-180327-174\
802-0575/tmp/docker-build.tar
[INFO] F8: [nebrass/myboutique:snapshot-180327-174802-0575] "spring-boot": Created docker-build.tar in\
 283 milliseconds
[INFO] F8: [nebrass/myboutique:snapshot-180327-174802-0575] "spring-boot": Built image sha256:61171
[INFO] F8: [nebrass/myboutique:snapshot-180327-174802-0575] "spring-boot": Tag with latest
[INFO]
...
```

```
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 30.505 s
[INFO] Finished at: 2018-03-27T17:48:29+02:00
[INFO] Final Memory: 70M/721M
[INFO] ------------------------------------------------------------------------
```

- Generating the resources descriptors based on the detected configuration: Spring Boot application that is using the port 8080 with existing Actuator endpoints.
- Building the Docker image in Kubernetes mode (locally and not like the Openshift mode, which uses the Openshift S2I Mechanism for the build).

After building our project, we got these files in the `${basedir}/target/classes/META-INF/fabric8/kubernetes` directory:

- my-school-deployment.yml
- my-school-svc.yml

Let's check the Deployment:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    fabric8.io/git-commit: 0120b762d7e26994e8b01d7e85f8941e5d095130
    fabric8.io/git-branch: master
    fabric8.io/scm-tag: HEAD
    ...
  labels:
    app: myboutique
    provider: fabric8
    version: 0.0.1-SNAPSHOT
    group: com.onepoint.labs
  name: myboutique
  namespace: default
spec:
  replicas: 1
  revisionHistoryLimit: 2
  selector:
    matchLabels:
      app: myboutique
      provider: fabric8
      group: com.onepoint.labs
  template:
    metadata:
      annotations:
        fabric8.io/git-commit: 0120b762d7e26994e8b01d7e85f8941e5d095130
        fabric8.io/git-branch: master
        fabric8.io/scm-tag: HEAD
        ...
      labels:
        app: myboutique
        provider: fabric8
        version: 0.0.1-SNAPSHOT
```

```
      group: com.onepoint.labs
  spec:
    containers:
    - env:
      - name: POSTGRES_SERVICE
        valueFrom:
          configMapKeyRef:
            key: postgres.service.name
            name: postgres-config
      - name: POSTGRES_DB_NAME
        valueFrom:
          configMapKeyRef:
            key: postgres.db.name
            name: postgres-config
      - name: POSTGRES_DB_USER
        valueFrom:
          secretKeyRef:
            key: db.user.name
            name: db-security
      - name: POSTGRES_DB_PASSWORD
        valueFrom:
          secretKeyRef:
            key: db.user.password
            name: db-security
      - name: KUBERNETES_NAMESPACE
        valueFrom:
          fieldRef:
            fieldPath: metadata.namespace
      image: nebrass/myboutique:snapshot-180327-003059-0437
      imagePullPolicy: IfNotPresent
      livenessProbe:
        httpGet:
          path: /health
          port: 8080
          scheme: HTTP
        initialDelaySeconds: 180
      name: myboutique
      ports:
      - containerPort: 8080
        name: http
        protocol: TCP
      - containerPort: 9779
        name: prometheus
        protocol: TCP
      - containerPort: 8778
        name: jolokia
        protocol: TCP
      readinessProbe:
        httpGet:
          path: /health
          port: 8080
          scheme: HTTP
        initialDelaySeconds: 10
      securityContext:
        privileged: false
```

- Generated annotations that holds many usefull data, like the `git-commit` id or the `git-branch`

- Labels section holds the Maven Project `groupId`, `artifactId` and `version` information. Add to that, a label `provider=fabric8` to tell you that this data is generated by **f8mp**

- The Docker Image, generated and built by **f8mp**. The suffix *snapshot-180327-003059-0437* is the default format to assign a version tag.

- A liveness probe checks if the container in which it is configured is still up.

- A readiness probe determines if a container is ready to service requests.

> **Tip**
>
> The liveness and readiness probes are generated because the **f8mp** has detected that the Spring-Boot-Actuator library in the classpath.

At this point, we can deploy our application just using the command `mvn fabric8:apply`, the output will look like:

```
[INFO] --- fabric8-maven-plugin:3.5.41:apply (default-cli) @ myboutique ---
[INFO] F8: Using Kubernetes at https://192.168.99.100:8443/ in namespace default with manifest
/Users/n.lamouchi/Downloads/MyBoutiqueReactive/target/classes/META-INF/fabric8/kubernetes.yml
[INFO] Using namespace: default
[INFO] Updating a Service from kubernetes.yml
[INFO] Updated Service: target/fabric8/applyJson/default/service-myboutique.json
[INFO] Using namespace: default
[INFO] Creating a Deployment from kubernetes.yml namespace default name myboutique
[INFO] Created Deployment: target/fabric8/applyJson/default/deployment-myboutique.json
[INFO] F8: HINT: Use the command `kubectl get pods -w` to watch your pods start up
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 16.003 s
[INFO] Finished at: 2018-03-28T00:03:56+02:00
[INFO] Final Memory: 78M/756M
[INFO] ------------------------------------------------------------------------
```

We can check all the resources that exists on our cluster

```
kubectl get all
```

This command will list all the resources in the `default` namespace. The output will be something:

```
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/myboutique   1         1         1            1           6m
deploy/postgresql   1         1         1            1           7m

    NAME                    DESIRED   CURRENT   READY     AGE
rs/myboutique-5dd7cbff98    1         1         1         6m
rs/postgresql-5f57747985    1         1         1         7m

NAME                              READY     STATUS    RESTARTS   AGE
po/myboutique-5dd7cbff98-w2wtl    1/1       Running   0          6m
po/postgresql-5f57747985-8n9h6    1/1       Running   0          7m

NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
svc/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP    23m
svc/myboutique   ClusterIP   10.106.72.231   <none>        8080/TCP   20m
svc/postgresql   ClusterIP   10.111.62.173   <none>        5432/TCP   21m
```

Tip

We can list all these resources on the K8s Dashboard.

Wow! Yes, these resources have been created during the steps that we did before :) Good job !

It works ! Hakuna Matata !

It's done) we deployed the application and all its required resources; but how can we access the deployed application?

The application will be accessible thru the Kubernetes Service object called `myboutique`.

Let's check what is the service `myboutique`, type `kubectl get svc myboutique`, the output will be:

```
NAME        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
myboutique  ClusterIP   10.106.72.231   <none>        8080/TCP   1d
```

The type of our `service` is `ClusterIP`. What is a `ClusterIP` ?

`ClusterIP` is the default ServiceType. It exposes the service on a cluster-internal IP so it will be only reachable from within the cluster.

So we cannot use this ServiceType because we need our service to be reachable from outisde the cluster. So is there any other type of service?

Yes! There are three other types of services, other than `ClusterIP`:

- `NodePort`: Exposes the service on each Node's IP at a static port (the `NodePort`). A `ClusterIP` service, to which the `NodePort` service will route, is automatically created. You'll be able to contact the `NodePort` service, from outside the cluster, by requesting `<NodeIP>:<NodePort>`.

- `LoadBalancer`: Exposes the service externally using a cloud provider's load balancer. `NodePort` and `ClusterIP` services, to which the external load balancer will route, are automatically created.

- `ExternalName`: Maps the service to the contents of the `externalName` field (e.g. `foo.bar.example.com`), by returning a `CNAME` record with its value.

> **Tip**
>
> In our case, we will be using the `LoadBalancer` service, which redirects traffic across all the nodes. Clients connect to the `LoadBalancer` service through the load balancer's IP.

Ok :) The `LoadBalancer` will be our `ServiceType`. But how can we tell this to the **f8mp** ?

We have two solutions:

- The **Resource Fragments** as we did before.
- The **Inline Configuration**, which is XML based configuration of the **f8mp** plugin.

Let's use this time the **Inline Configuration** to tell the **f8mp** that we want a `LoadBalancer` service:

In the `configuration` section of the **f8mp** plugin, we will declare an `enricher`.

An `enricher` is a component used to create and customize Kubernetes and Openshift resource objects. **f8mp** comes with a set of `enrichers` which are enabled by default. One of these enrichers, is the `fmp-service` which is used to customize the Services.

The **f8mp** with the configured `enricher` will look like:

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>3.5.41</version>
    <configuration>
        <enricher>
            <config>
                <fmp-service>
                    <type>LoadBalancer</type>
                </fmp-service>
            </config>
        </enricher>
    </configuration>
    ...
</plugin>
```

Let's build and a redeploy our project using `mvn clean install fabric8:apply` and see what is the type of the deployed `service` using `kubectl get svc myboutique`:

```
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
myboutique   LoadBalancer   10.106.72.231   <pending>     8080:31246/TCP   2d
```

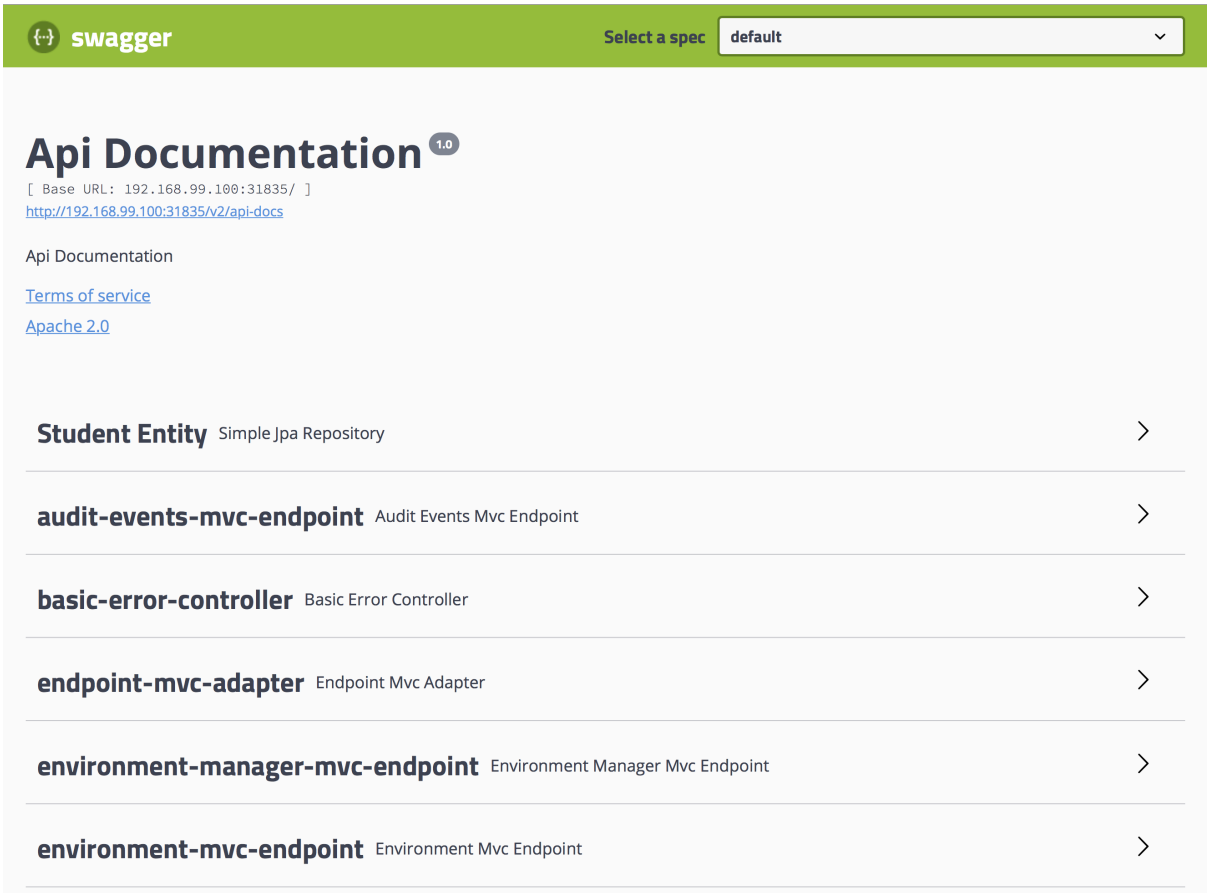
> **Warning**
>
> The `<pending>` shown in the `EXTERNAL-IP` column is due that we are using **minikube**.

Cool ! How can we access the application now? How can we get the URL of the deployed application?

Euuuuh! The answer is shorter than the question :D to get the URL of the deployed app on `minikube` just type:

```
open $(minikube service myboutique --url)
```

This command will open the URL of the Spring Boot Application in your default browser :)



Landing Page of the deployed application on Kubernetes

> **Tip**
>
> The command `minikube service myboutique --url` will give us the path of the service `myboutique`, which is pointing on our Spring Boot Application.

How can I access the Swagger UI of my deployed App?

```
open $(minikube service myboutique --url)/swagger-ui.html
```

This command will open the URL of the Spring Boot Application in your default browser :)

Revisiting our Cloud Patterns after meeting Kubernetes

This is the best part that I love in the book. Since I started writing this book, I was waiting to start typing words in this part :D

In this part, we will get rid of many components that we wrote :D and replace them by Kubernetes components :D Ohh ! This makes me happy :D !!

> When in Rome, do as the Romans do
>
> – Unknown

The **Spring Cloud** ecosystem is incubating a project called **Spring Cloud Kubernetes**: a set of librairies offering many tools to interact with Kubernetes clusters.

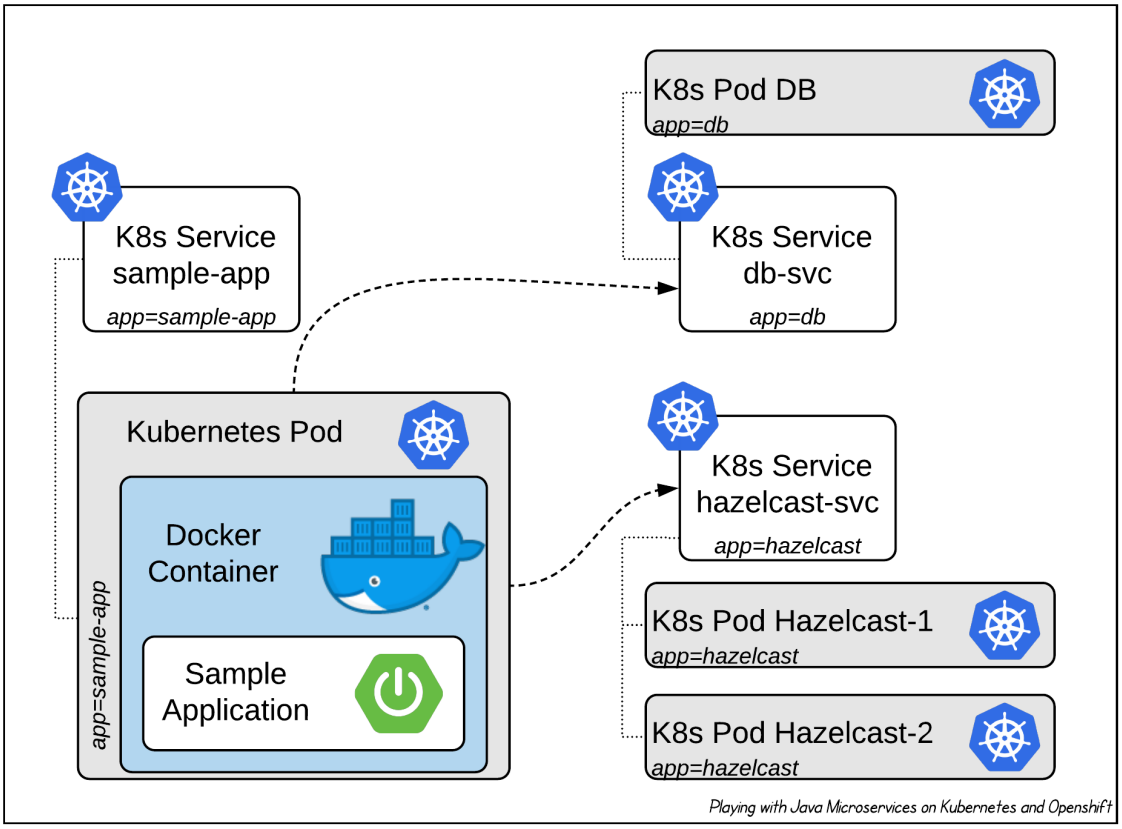
Service Discovery and Registration

Kubernetes offers great Service Discovery features. We can use these features to get rid of Eureka :D Yes ! We can make Kubernetes insure discovering our Microservices.

Cool !! This is great ! But we already have talked about many Kubernetes components, which one can we use to do Service Discovery?

The answer is: **Services** ! Yeah ! Let me tell you the story:

We already saw in the first part of this chapter, how to run a Spring Boot application in Kubernetes.
Our **application** is exectued in a **Docker Container** running inside a **Kubernetes Pod**. Our containerized application will need to talk to other applications in different other pods like *Hazelcast* cache engine or a DB. As illustrated in the next diagram:



Application running in a Container inside a Pod - Communication with Services

The DB instance is running in a Pod, that can crash or restard for any reasons.. so if we had eureka, we can keep go after the new instance.. Houhou ! The Kubernetes Service is doing the same thing ! The `db-svc` object is pointing on the DB Pods - when a new DB pod is created, it get automatically registered in the cluster data as a pod that belongs to those tracked by the `db-svc`.

The `db-svc` service is tracking the DB Pods using the selector `app=sample-app`. When deploying our microservices as containers, our pods will be tracked easily by exposing them as `Services`. In our microservices, we will keep calling distant microservices with their **Service Name**, and then the embedded DNS server is Kubernetes will resolve the **Service Name** in the same namespace of our pods. So we don't need Eureka or any library or configuration.

Kubernetes has a special pod called `kube-dns`, this pod is living in the `kube-system` namespace. This pod is the component responsible for resolving service names in our Kubernetes Cluster.

Service Discovery using **Environment Variables**

There is an other option for Service Discovery using **Environment Variables**: It consists to injecting a set of environment variables for each active Service when a Pod get created. This option is not recommended - the DNS-based Service Discovery is preferable.

Engaging the Kubernetes-enabled Discovery library

Step 1: Remove the Eureka Client dependency

We already mentionned that we will not be using **Eureka**, so we don't need anymore the **Eureka Client** dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

We need also to remove all **Eureka** properties from `application.properties|yaml` files.

Step 2: Add the Kubernetes-enabled Discovery library

The **Spring Cloud Kubernetes** library contains a library called **Spring Cloud Kubernetes Discovery**: this library allows us to query Kubernetes `Services` by name.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-kubernetes-discovery</artifactId>
</dependency>
```

Step 3: Defining the Kubernetes Service name:

Some **Spring Cloud** components will use the `DiscoveryClient` in order to obtain info about the a service instance. For this to work, we need to be sure that the `Service` name has the same value as the `spring.application.name` property.

To do this, we need to add this configuration tag to the **f8mp**:

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>3.5.41</version>
    <configuration>
        <enricher>
            <config>
                <fmp-service>
                    <name>${project.name}</name>
                    ...
                </fmp-service>
            </config>
        </enricher>
    </configuration>
    ...
</plugin>
```

We set `${project.name}` value to `config.fmp-service.name` property of the plugin `enricher`. This `${text}` tag will be parsed and replaced by **Maven**. This way, we will be sure that the service name will be the **Maven project name**.

Next we need to define the `spring.application.name` with the **Maven Project Name**; to do this we will need to add this configuration to the Maven Build tag:

```
<build>
   <resources>
       <resource>
           <directory>src/main/resources</directory>
           <filtering>true</filtering>
           <includes>
               <include>application.properties</include>
           </includes>
       </resource>
   </resources>
   <plugins>
...
</build>
```

And in the `application.properties` file; we will define the

```
spring.application.name=@project.name@
```

- Ok, but what are doing??
- Keep calm! :D Here we asked Maven to parse files in the `src/main/resources` while building the application; Here, the Maven parser will recognize the pattern `@project.name@` and it will replace it with the Maven `project.name` defined in the `pom.xml`.

Now we are sure that the `spring.application.name` and the Kubernetes `Service` name have the same value, which is the Maven `project.name`.

Load Balancing

We previously did Load Balancing in calls between microservices using **Zuul**, **Eureka** & **Ribbon**, but when we are in the Kubernetes world, there will be a sense of keeping them ?! :)

For example, in the previous schema, *Hazelcast* cache is running as two instances, in two different pods.

Server Side Load Balancing

Kubernetes already do this by including load balancing capabilities in the **Service** concept. Again? Yeah ! Again ! `Service` provides a single point of contact where calls will be load balanced and redirected to an appropriate pod between the available instances. Just we make a REST call to the `Service`, and Kubernetes will reroute the request to the relevant `Pod`.

When we want to deal with *Hazelcast*, just mention `hazelcast-svc` as hostname, and Kubernetes will ensure loadbalancing the request between the two available *Hazelcast* pods.

Client Side Load Balancing

We already used **Spring Cloud Netflix Ribbon**. In the `spring-cloud-kubernetes` project, we have a dedicated version of **Ribbon** for Kubernetes called `spring-cloud-kubernetes-ribbon`.

Within the `spring-cloud-kubernetes-ribbon` project, there is a Kubernetes client that populates a **Ribbon** `ServerList` containing information about a `Service` endpoints. This `ServerList` will be used by a load-balanced client to access the convenient service endpoint.

The discovery feature is used by the **Spring Cloud Kubernetes Ribbon** project to fetch the list of the endpoints defined for an application to be load balanced.

Externalized Configuration

We already used `ConfigMap` objects to store some properties of our application, and we used `Secret` objects to store passwords and credentials. We can inject these values in containers and use them as environment variables.

We can replace `Config Server` easily by `ConfigMaps` and `Secrets`.

This project provides integration with `ConfigMap` to make config maps accessible by Spring Boot.

The `ConfigMap PropertySource` when enabled will lookup Kubernetes for a `ConfigMap` named after the application (see spring.application.name). If the map is found it will read its data and do the following:

- apply individual configuration properties.
- apply as yaml the content of any property named `application.yaml`
- apply as properties file the content of any property named `application.properties`

Replacing the Config Server by ConfigMaps

Step 1: Removing the Spring Cloud Config footprints

From the `pom.xml` file, we need to remove Spring Cloud Config dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

Next, in the `bootstrap.yml` file we need to remove Spring Cloud Config properties:

```
spring:
  cloud:
    config:
      uri: http://localhost:8888
```

Step 2: Adding the Maven Dependencies

We need to add two dependencies to make our Spring Boot application consume the `ConfigMaps` instead of the `Config Server`:

Spring Cloud Kubernetes Maven dependencies for ConfigMaps

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-kubernetes-core</artifactId>
    <version>0.3.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-kubernetes-config</artifactId>
    <version>0.3.0.RELEASE</version>
</dependency>
```

Step 3: Creating ConfigMaps based on the Application properties files

We need to create a `ConfigMap` for every `application.properties|yaml` file. To do this, we will go to `config-server/src/main/resources/configurations/` folder containing the properties files of our microservices.

For every file, we need to create a dedicated `ConfigMap`, which has the same name as the `properties` file, for example: we will create a ConfigMap called `order-service` based on the `order-service.yml` file:

```
kubectl create configmap order-service --from-file=application.properties=order-service.yml
```

This command created a `ConfigMap` called `order-service` based on the `order-service.yml`. The `spring-cloud-kubernetes-config` library will look for a `ConfigMap` that has the same name as the `spring.application.name` property in the context.

The `ConfigMap`'s Key will be `application.properties`. Obviously, you got it ! Yeah ! That key will enable the Spring Boot application to handle the `ConfigMap` value like a real `application.properties` file.

Step 4: Authorizing the ServiceAccount access to ConfigMaps

We need to authorize the `ServiceAccount`, the default one as we didn't explicitly define one, to access the `ConfigMaps` in the same `Namespace`, which is the `default` one. So the **Service Account** is `default:default`.

```
kubectl create clusterrolebinding configmap-access --clusterrole view --serviceaccount=default:default
```

BINGO!! Let's optimize the integration.

Step 5: Boosting Spring Cloud Kubernetes Config

There is a great features in **Spring Cloud Config Server** integration that enables the applications to detect changes on external property sources and update their internal status to reflect the new configuration.
The same feature exists in **Spring Cloud Kubernetes** is able to trigger an application reload when a related `ConfigMap` or `Secret` changes.

This feature is disabled by default and can be enabled using the configuration property in the `application.properties` file:

```
spring.cloud.kubernetes.reload.enabled=true
```

The following levels of reload are supported (property `spring.cloud.kubernetes.reload.strategy`):

- `refresh` *(default)*: only configuration beans annotated with `@ConfigurationProperties` or `@RefreshScope` are reloaded. This reload level leverages the refresh feature of Spring Cloud Context.
- `restart_context`: the whole Spring *ApplicationContext* is gracefully restarted. Beans are recreated with the new configuration.
- `shutdown`: the Spring *ApplicationContext* is shut down to activate a restart of the container. When using this level, make sure that the lifecycle of all non-daemon threads is bound to the ApplicationContext and that a replication controller or replica set is configured to restart the pod.

The reload feature supports two operating modes:

- **event** *(default)*: watches for changes in config maps or secrets using the Kubernetes API (web socket). Any event will produce a re-check on the configuration and a reload in case of changes. The `view` role on the service account is required in order to listen for config map changes. A higher level role (eg. `edit`) is required for secrets (secrets are not monitored by default).
- **polling**: re-creates the configuration periodically from config maps and secrets to see if it has changed. The polling period can be configured using the property `spring.cloud.kubernetes.reload.period` and defaults to *15 seconds*. It requires the same role as the monitored property source. This means, for example, that using polling on file mounted secret sources does not require particular privileges.

Properties:

Name	Type	Default	Description
spring.cloud.kubernetes.reload.enabled	Boolean	false	Enables monitoring of property sources and configuration reload
spring.cloud.kubernetes.reload.monitoring-config-maps	Boolean	true	Allow monitoring changes in config maps
spring.cloud.kubernetes.reload.monitoring-secrets	Boolean	false	Allow monitoring changes in secrets
spring.cloud.kubernetes.reload.strategy	Enum	refresh	The strategy to use when firing a reload (*refresh*, *restart_context*, *shutdown*)
spring.cloud.kubernetes.reload.mode	Enum	event	Specifies how to listen for changes in property sources (*event*, *polling*)
spring.cloud.kubernetes.reload.period	Long	15000	The period in milliseconds for verifying changes when using the *polling* strategy

Spring Cloud Kubernetes Reload properties

Notes:

- Properties under *spring.cloud.kubernetes.reload.* should not be used in config maps or secrets: changing such properties at runtime may lead to unexpected results;
- Deleting a property or the whole config map does not restore the original state of the beans when using the *refresh* level.

Log aggregation

We already used ELK (Elasticsearch, Logstash, Kibana). EFK is the same stack where we replace Logstash by Fluentd.
So why replace Logstash with Fluentd?

First of all, as many other projects like Kubernetes, Prometheus, etc.. Also, I find it very easy to configure, there is a lot of plugins and its memory footprint is very low.

Fluentd is an open source data collector, which lets you unify the data collection and consumption for a better use and understanding of data. Fluentd is an active project in the Cloud Native Computing Foundation, the Kubernetes project neighbourhood. Fluentd offers a great Kubernetes integration, which is far away better than Logstash. This means that installing and using Fluentd on Kubernetes is easier than doing the same using Logstash.

In this tutorial, we will be using **FluentBit**, a lightweight shipper that can be installed as agents on edge hosts or devices in a distributed architecture. It acts as a collector and forwarder and was designed with performance in priority.

Step 1: Prepare the Minikube

We need to have a custom configuration for Minikube, different from the default one which is relatively insufficient for a large number of Pod being executed simultaneously.

Default Minikube configuration
Minikube has a default configuration that is used if no custom one is defined.

- The _DefaultMemory_ is 2048m
- The _DefaultCPUS_ is 2 cpus
- The _DefaultDiskSize_ is 20g

When needed, we override these values to fit the needs.

We need to delete the old Kubernetes Cluster:

```
minikube delete --force
```

We need 8 giga of memory:

```
minikube config set memory 8192
```

We need 3 cpus:

```
minikube config set cpus 3
```

Step 2: Install EFK using Helm

To install the EFK Stack on our **Kubernetes** cluster, we will be using **Helm**.

What is Helm ?

Helm is a package manager for **Kubernetes** that allows developers and operators to more easily package, configure, and deploy I> applications and services onto **Kubernetes** clusters.

Helm is now an official **Kubernetes** project and is part of the **Cloud Native Computing Foundation**, a non-profit that supports open source projects in and around the **Kubernetes** ecosystem.

Helm can:

- Install software.
- Automatically install software dependencies.
- Upgrade software.
- Configure software deployments.
- Fetch software packages from repositories.

Helm provides this functionality through the following components:

- A command line tool, `helm`, which provides the user interface to all Helm functionality.
- A companion server component, `tiller`, that runs on your Kubernetes cluster, listens for commands from `helm`, and handles the I> configuration and deployment of software releases on the cluster.
- The Helm packaging format, called charts.
- An official curated charts repository with prepackaged charts for popular open-source software projects.

What is Helm Chart

Helm packages are called charts, and they consist of a few YAML configuration files and some templates that are rendered into Kubernetes manifest files.

The `helm` command can install a chart from a local directory, or from a `.tar.gz` packaged version of this directory structure.
These packaged charts can also be automatically downloaded and installed from chart repositories or repos.

Step 2.1: Prepare Helm

First of all, we need to install **Helm Tiller** component. **Tiller** is the in-cluster component of Helm. It interacts directly with the Kubernetes API server to install, upgrade, query, and remove Kubernetes resources. It also stores the objects that represent releases.

To deploy the `tiller` pod in the cluster:

```
helm init
```

The `tiller` pod will be created in the `kube-system` namespace.

Verify that the `tiller` pod is running by the command:

```
kubectl get pods -n kube-system -w
```

```
NAME                                    READY   STATUS    RESTARTS   AGE
...
tiller-deploy-845cffcd48-mh8vl          1/1     Running   0          1m
```

Step 2.2: Add the Chart repository

We will be using the great **EFK Charts** created by **Alen Komljen**[32], great devops and blogger.

We need to add his repository in the Helm repositories:

```
helm repo add akomljen-charts \
    https://raw.githubusercontent.com/komljen/helm-charts/master/charts/
```

Step 3: Installing the Elasticsearch Operator

We will start by defining an **Operator** : A Kubernetes Operator is basically a custom API object registered as **Custom Resource Definition** (aka CRD) which enables us to create a custom business logic for operating with particular service. An Operator represents human operational knowledge in software to reliably manage an application.

To install the **Elasticsearch Operator**:

```
helm install --name es-operator \
    --namespace logging \
    akomljen-charts/elasticsearch-operator
```

After deploying the **Elasticsearch Operator** we can check that new CustomResourceDefinition (aka CRD) is created:

```
kubectl get crd
```

```
NAME                                          CREATED AT
elasticsearchclusters.enterprises.upmc.com    2018-11-02T20:37:02Z
```

To check the details of this CRD:

[32] https://akomljen.com/get-kubernetes-logs-with-efk-stack-in-5-minutes/

```
kubectl describe crd elasticsearchclusters.enterprises.upmc.com

Name:         elasticsearchclusters.enterprises.upmc.com
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  apiextensions.k8s.io/v1beta1
Kind:         CustomResourceDefinition
Metadata:
...
Spec:
...
  Group:   enterprises.upmc.com
  Names:
    Kind:       ElasticsearchCluster
    List Kind:  ElasticsearchClusterList
    Plural:     elasticsearchclusters
    Singular:   elasticsearchcluster
  Scope:        Namespaced
  Version:      v1
...
```

As you can see, we have a new kind of resource called `ElasticsearchCluster`. This CRD will be used when creating the **Elasticsearch Cluster**.

Step 4: Installing the EFK Stack using Helm

After deploying the CustomResourceDefinition required by the cluster, just enter this command to install the EFK Stack:

```
helm install --name efk \
    --namespace logging \
    akomljen-charts/efk
```

This action will take some minutes, the time that Docker images get pulled from repos and deployed to Kubernetes.

Just type this command to verify that all services are created and running:

```
kubectl get pods -n logging -w

NAME                                                READY   STATUS    RESTARTS   AGE
efk-kibana-fc4bbccb7-57kzp                          1/1     Running   0          5m
es-client-efk-cluster-857fd4c567-w4qw8              1/1     Running   0          5m
es-data-efk-cluster-default-0                       1/1     Running   0          5m
es-master-efk-cluster-default-0                     1/1     Running   0          5m
es-operator-elasticsearch-operator-fbbd9556c-vdq4r  1/1     Running   0          37m
fluent-bit-mfq2m                                    1/1     Running   0          5m
```

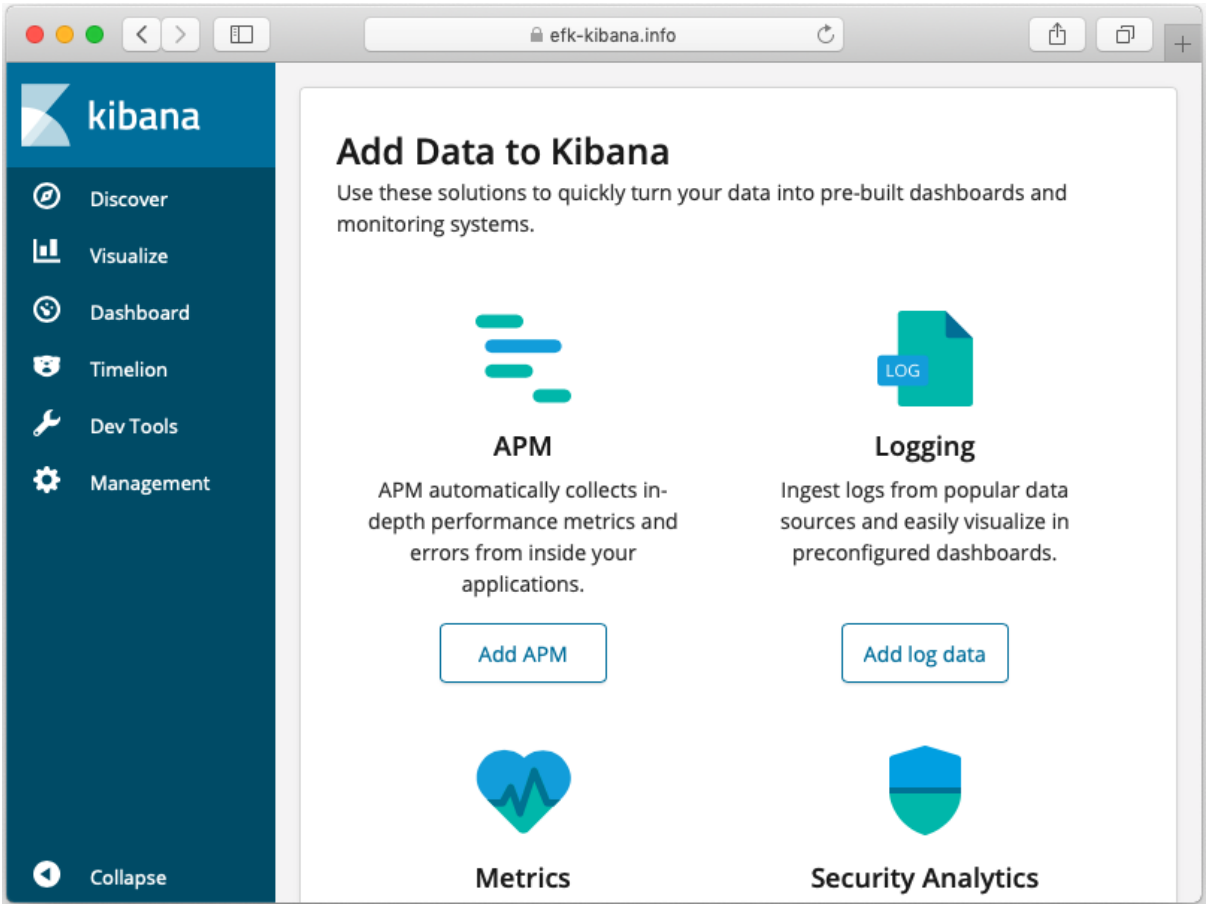
After a few minutes, all services should be up and running. We will create an **Ingress** to access the **Kibana** service:

```
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: kibana-public
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: efk-kibana.info
      http:
        paths:
          - path: /
            backend:
              serviceName: efk-kibana
              servicePort: 5601
```

To access the `host` defined `efk-kibana.info`, just do:

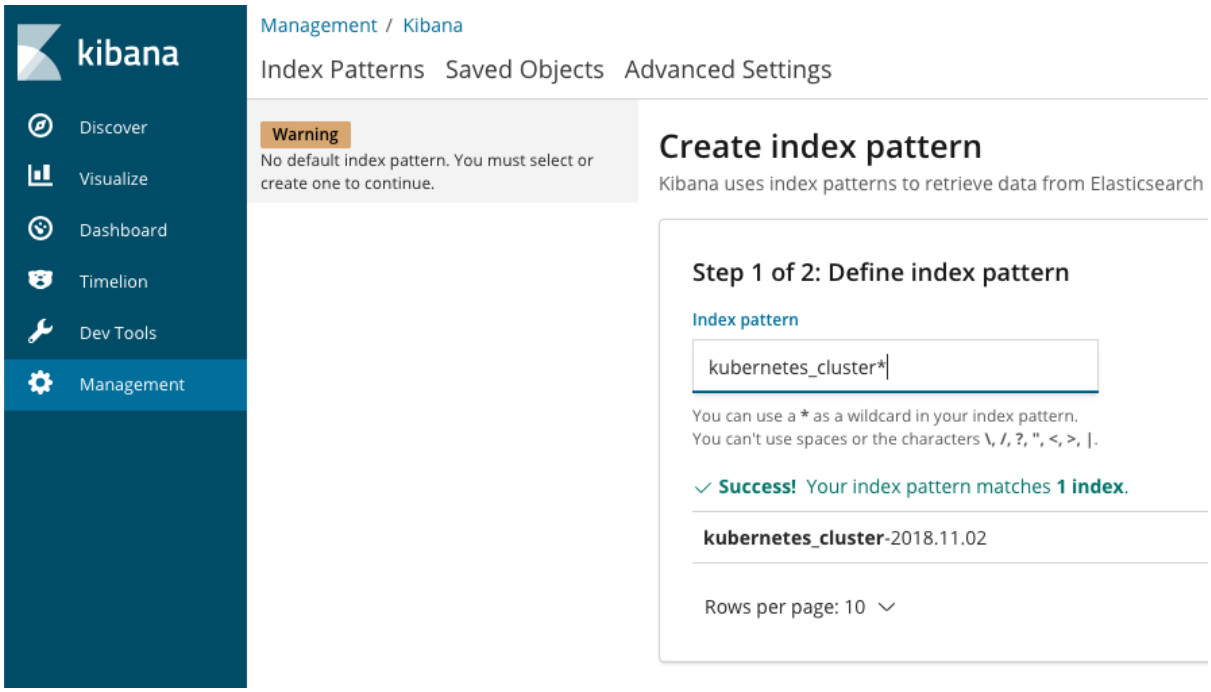
```
echo "$(minikube ip) efk-kibana.info" | sudo tee -a /etc/hosts
```

This command will make the **Kibana** service reacheable on `efk-kibana.info`:



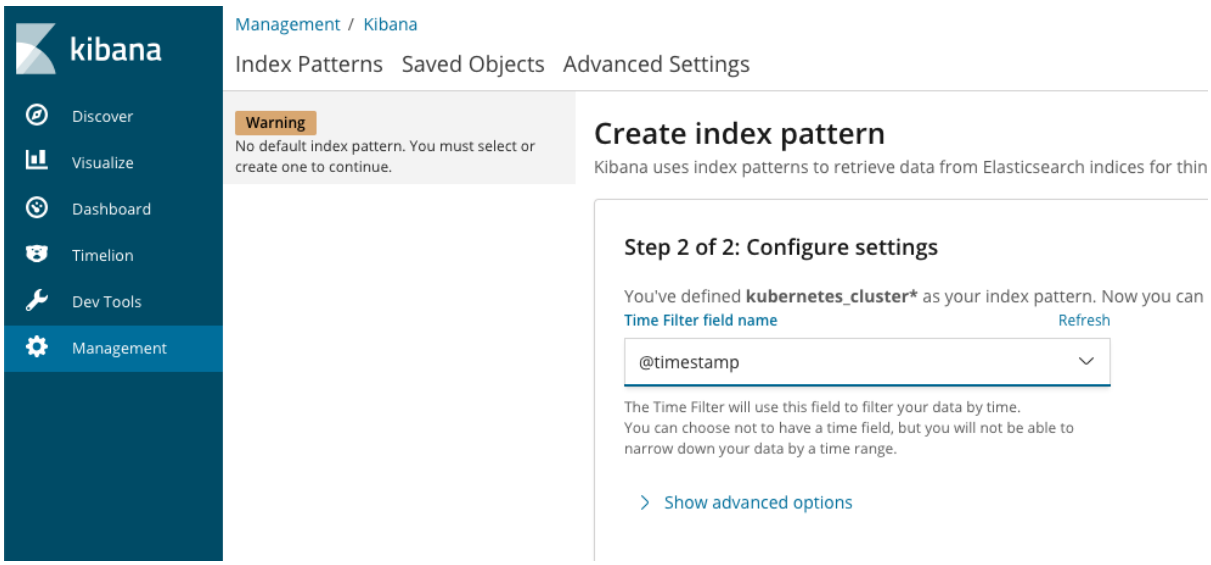
Kibana console reacheable on **`efk-kibana.info`**

Then, go to `Dashboard` menu item, configure the index to `kubernetes_cluster*`:



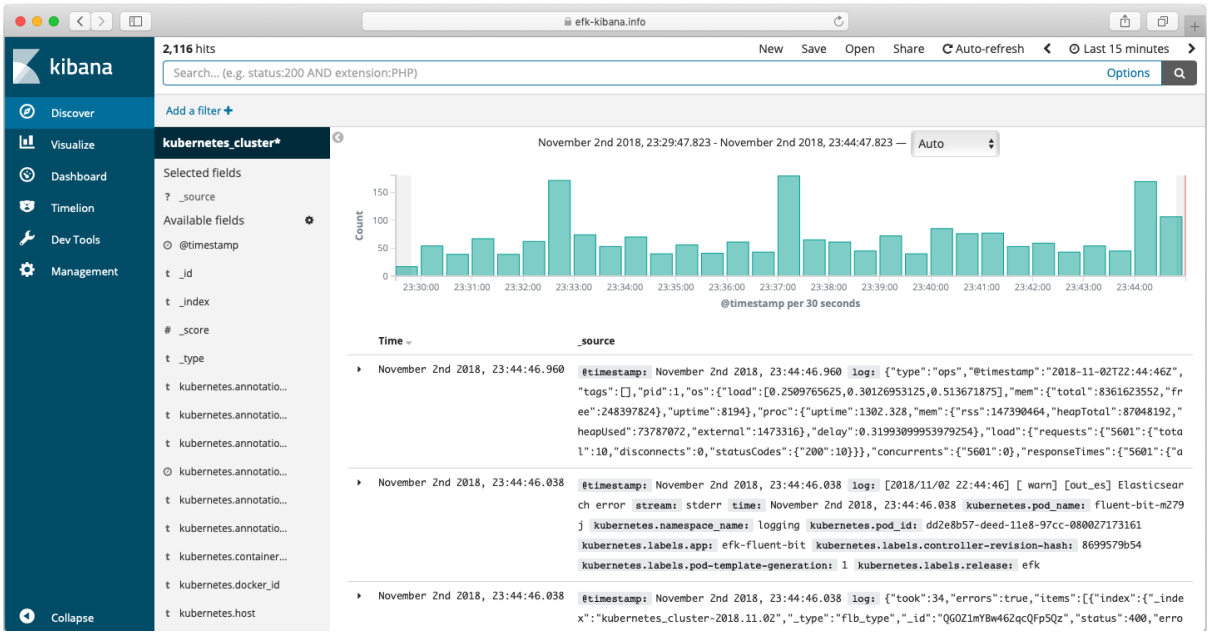
Kibana Index - Step 1

Next, choose a `@timestamp` and Kibana is ready:



Kibana Index - Step 2

You should see all logs from all namespaces in your Kubernetes cluster. Rich Kubernetes metadata will be there also:



Kibana - Discover

Now, all Pods logs are now centralized in Kibana.

Step 5: Remove the broadcasting appenders

We no more need the log appenders that broadcast logs to *Logstash*. The reason is simple: our actual log aggregation mecanism is sending log from the docker container, using Fluent Bit, to Elasticsearch. So, we will keep the console appenders only.

We need to delete the `LoggingConfiguration.java` class as it defines the broadcasting log appenders.

We need to add an Logstash appender that writes code to the standard output (console). To do so, add this `logback.xml` to the `src/main/resources/` folder of each microservice:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <include resource="org/springframework/boot/logging/logback/base.xml" />
    <logger name="org.hibernate.validator" level="info" />
    <logger name="org.springframework.cloud.kubernetes" level="debug" />
    <root level="info">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

This configuration file will let the application write its log in a coherent way with the EFK style.

Health check API

Our microservices are using **Spring Boot Actuator** library to expose operational information (health, metrics, etc..) about the running application.

Spring Boot Actuator exposes some helpful endpoints, one of them is the `/health` endpoint, which returns the value `UP` when it is **OK** or **DOWN** when something is wrong.

Kubernetes already has the same mindset of dealing with operational information endpoints. Kubernetes has two types of probes ReadinessProbe and LivenessProbe :

- Readiness probes are used to know when a Container is ready to start accepting traffic. A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.
- Liveness probes are used to know when to restart a Container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

Liveness and readiness probes are an excellent way to make sure that your application stays healthy in your Kubernetes cluster.

What is changing when Actuator lands on the Kubernetes territory ?

The solution is to pair the Liveness and Readiness probes with the Actuator `health` endpoints. This will garantee a high level of consistency between our Spring Boot/Cloud applications and the Kubernetes cluster.

The idea, is to register the Spring Boot Actuator endpoints as Liveness and Readiness probes in Kubernetes.

This task is done automatically by our great `f8mp` plugin: when generating the `deployment.yaml` file, the generator adds Kubernetes liveness and readiness probes pointing to properties scanned in the application. Whenthe generator detects the `spring-boot-starter-actuator` dependency, the `f8mp` enricher will add Kubernetes readiness and liveness probes for Spring Boot.

If we check the Kubernetes Resources Descriptors generated by the `f8mp` plugin, we can see that there are the readiness and liveness probes:

target/classes/META-INF/fabric8/kubernetes/deployment.yml

```
apiVersion: extensions/v1beta1
kind: Deployment
...
spec:
...
  template:
...
    spec:
      containers:
      - env:
        ...
        livenessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 180
        ...
        readinessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 10
...
```

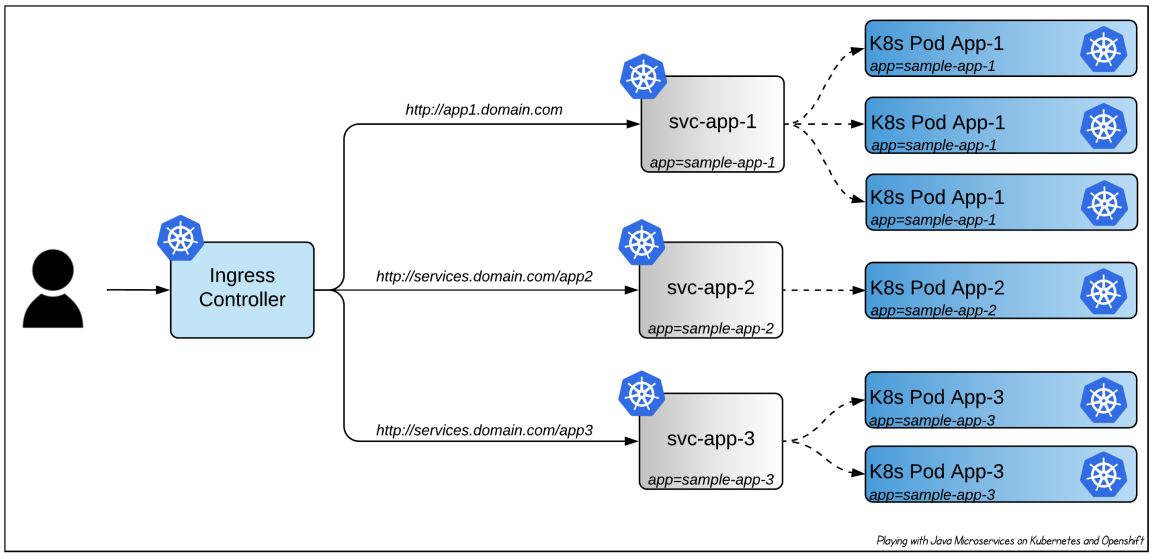
Great ! Let's move to the next pattern !

API Gateway

An API gateway is a programming façade that sits in front of APIs and acts as a single point of entry for a defined group of microservices.

For implementing the API Gateway use used **Spring Cloud Netflix Zuul**. But which Kubernetes feature can replace it ?

Kubernetes Ingress manages external access to the services in a cluster, typically HTTP. Ingress can provide load balancing, SSL termination and name-based virtual hosting.



Exposing services using an Ingress

An Ingress is a collection of rules that allow inbound connections to reach the cluster services. It can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, offer name based virtual hosting, and more. Users request ingress by POSTing the Ingress resource to the API server. An Ingress controller is responsible for fulfilling the Ingress, usually with a loadbalancer, though it may also configure your edge router or additional frontends to help handle the traffic in an HA manner.

Let's bring our Api Gateway to Kubernetes :D

Step 1: Delete the old ApiGateway microservice

As we did with Eureka and the Config Server, we will also delete totally the ApiGateway microservice :D Kubernetes already provides the Ingress resource that we can use to implement the API Gateway pattern.

Step 2: Create the ApiGateway Ingress

Our Ingress will point on our 3 microservices:
- ProductService
- OrderService
- CustomerService

We need a domain name for the ingress; let's suppose that we have already the `myboutique.io` domain name.

Our Ingress descriptor will look like:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: api-gateway
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  backend:
    serviceName: default-http-backend
    servicePort: 80
  rules:
  - host: myboutique.io
    http:
      paths:
      - path: /product
        backend:
          serviceName: product-service
          servicePort: 8080
      - path: /order
        backend:
          serviceName: order-service
          servicePort: 8080
      - path: /customer
        backend:
          serviceName: customer-service
          servicePort: 8080
```

Just save this content to `api-gateway-ingress.yml` and to create the resource just do:

```
kubectl create -f api-gateway-ingress.yml
```

Et hop ! The Ingress is created successfully !

Distributed Tracing

In this part, we will need to move the standalone Zipkin instance that we had before, to the Kubernetes Cluster and to update our configuration to make microservice point on the Zipkin hosted on Kubernetes instead of the older `localhost:9411`.

Step 1: Deploy Zipkin to Kubernetes

There is an official Docker image for Zipkin: it's `openzipkin/zipkin` that we will deploy to Kubernetes:

```
kubectl create deployment zipkin --image=openzipkin/zipkin
```

Then, after creating the `Deployment`, we need to expose it using a `Service` that we will be used as a Zipkin `hostname`:

```
kubectl expose deployment zipkin --type=LoadBalancer --port 9411
```

Now, all requests forwarded to the hostname called `zipkin` will be loadbalanced to the available Zipkin `Pods`.

Step 2: Forward Sleth traces to Zipkin

We need to mention that :

```
spring:
  application:
    name: order-service
  sleuth:
    sampler:
      probability: 1
  zipkin:
    baseUrl: http://zipkin/
...
```

The `spring.zipkin.baseUrl` is pointing on the `zipkin` Kubernetes Service.

That's it !! Everything will be working like a charm ! :)

Chapter 14: Getting started with OpenShift

Introduction

OpenShift is a container application platform that brings Docker and Kubernetes to the enterprise.

OpenShift Container Platform logo

OpenShift provides container-based software deployment and management for apps on-premise, in a public cloud, or hosted. OpenShift adds operations-centric tools on top of Kubernetes to enable rapid application development, easy deployment and scaling, and long-term lifecycle maintenance.

This is the literature definitions that I see that didn't clearly define what OpenShift is ?

I define OpenShift as a **PaaS** made by RedHat, built around a core of application containers (mainly based on Docker), with orchestration provided by Kubernetes, running on Red Hat Enterprise Linux. OpenShift is a layered system designed to expose underlying Docker-formatted container image and Kubernetes concepts as accurately as possible, with a focus on easy composition of applications by a developer.

What is a PaaS ?

Platform as a Service (PaaS) or **Application Platform as a Service (aPaaS)** is a category of *cloud computing services* that provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.

You can get OpenShift deployed on your public, private or even hybrid cloud, on bare-metal or you can even get hosted or managed editions by RedHat.

OpenShift can be deployed on a public, private or hybrid cloud. OpenShift helps you deploying your applications using Docker containers. OpenShift aims to simplify the development and deployment of **Cloud Native Applications**.

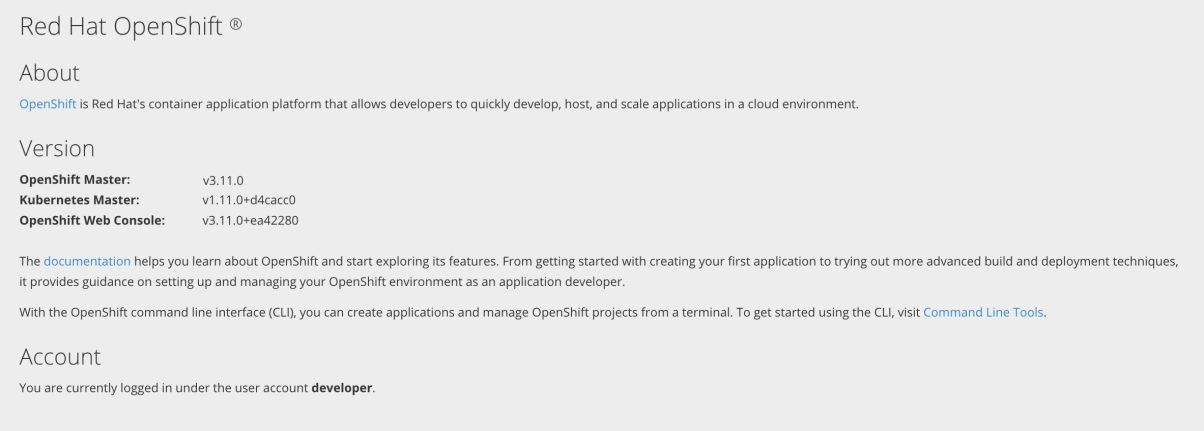
What is a Cloud Native Application ?

Cloud-native is an approach to building and running applications that exploits the advantages of the cloud computing delivery model. **Cloud-native** is about how applications are created and deployed, *not where*. While today public cloud impacts the thinking about infrastructure investment for virtually every industry, a cloud-like delivery model isn't exclusive to public environments. It's appropriate for both public and private clouds. Most important is the ability to offer nearly limitless computing power, on-demand, along with modern data and application services for developers. When companies build and operate applications in a **cloud-native** fashion, they bring new ideas to market faster and respond sooner to customer demands.

Organizations require a platform for building and operating cloud-native applications and services that automates and integrates the concepts of:
- DevOps as a Development Process
- Microservices as Application Architecture
- Containers as Deployment and Packaging format
- Cloud as Application Infrastructure

What is really OpenShift ?

Openshift is mainly based on Kubernetes. Moreover, Openshift Container Platform versions are indicating the Kubernetes version included in the product. For example the OCP v3.9 is based on Kubernetes v1.9, the OCP v3.11 is based on Kuberentes v1.11 and so on.. This detail shows how much OpenShift relies on Kuberentes. Personnally, I like to present OpenShift as a customized version of Kubernetes, delivered with some extra components.



OCP version based on Kuberentes version

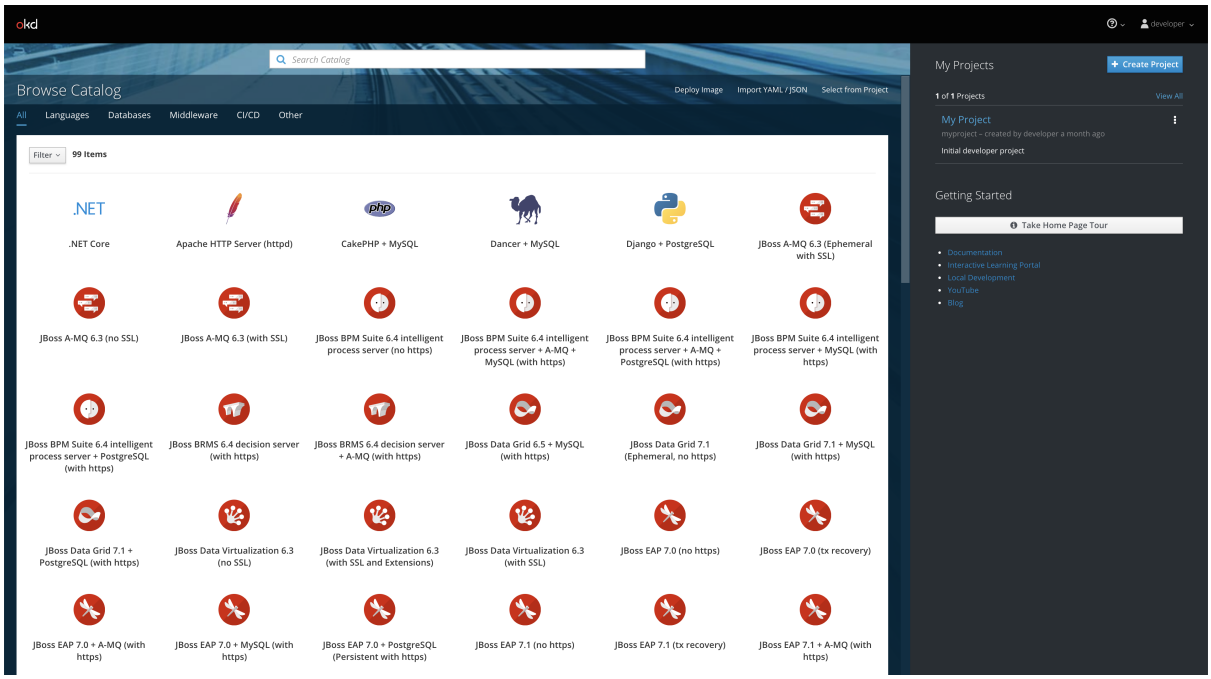
OpenShift Container Platform is a paid product that you can install on your infrastructure that has Red Hat's paid support included in the subscription that you are paying, and that needs to be renewed or even upgraded when your cluster grows.

There is a free open source version of OpenShift called **OKD**. It includes most of the features of its commercial product, but you cannot have Red Hat's support.

OpenShift comes with its own containers-images registry, that is used to store locally, the Docker images that we will be using accross our cluster. This registry will hold all the Red Hat certified

images that you will have included in the paid subscription. In the **OKD** installation we will have also this registry, but without Red Hat certified Docker images.

OpenShift comes also with its own command-line interface, called *openshift-cli*, and its own web dashboard (and not the Kubernetes Dashboard UI)



OCP Dashboard

Chapter 15: The Openshift style

Part three: Conclusions

Chapter 16: Conclusions