Reference Architectures 2017
Spring Boot Microservices on Red Hat OpenShift Container Platform 3

Babak Mozaffari

Reference Architectures 2017 Spring Boot Microservices on Red Hat OpenShift Container Platform 3

Babak Mozaffari
refarch-feedback@redhat.com

Legal Notice

Abstract

This reference architecture demonstrates the design, development and deployment of Spring Boot Microservices on Red Hat® OpenShift Container Platform 3.

Table of Contents

COMMENTS AND FEEDBACK

In the spirit of open source, we invite anyone to provide feedback and comments on any reference architecture. Although we review our papers internally, sometimes issues or typographical errors are encountered. Feedback allows us to not only improve the quality of the papers we produce, but allows the reader to provide their thoughts on potential improvements and topic expansion to the papers. Feedback on the papers can be provided by emailing refarch-feedback@redhat.com. Please refer to the title within the email.

CHAPTER 1. EXECUTIVE SUMMARY

This reference architecture demonstrates the design, development, and deployment of **Spring Boot** microservices on **Red Hat® OpenShift Container Platform 3**. The reference application is built with a number of open source components, commonly found in most Spring Boot microservice deployments. Minor adjustments to the software stack are applied as appropriate for the OpenShift environment.

Red Hat OpenShift Application Runtimes (RHOAR) is an ongoing effort by Red Hat to provide official OpenShift images where certain 3rd party software, including Spring Boot, have been tested and verified on top of supported components including JBoss Web Server, OpenJDK and the base image itself.

The reference architecture serves as a potential blueprint for certain greenfield and brownfield projects. This includes scenarios where teams or environments have a strong preference to use the software stack most common in Spring Boot microservices, despite the availability of other options when taking advantage of OpenShift as the deployment platform. This architecture can also help guide the migration and deployment of existing Spring Boot microservices on OpenShift Container Platform.

CHAPTER 2. SOFTWARE STACK

2.1. FRAMEWORK

Numerous frameworks are available for building microservices, and each provides various advantage and disadvantages. This reference architecture focuses on a microservice architecture built on top of the Spring Boot framework. The Spring Boot framework can use various versions of **`Tomcat`**, **`Jetty`** and **`Undertow`** as its embedded servlet containers. This paper focuses on the use of Spring Boot with an embedded Tomcat server, running on an OpenShift base image from Red Hat®, with a supported JVM and environment.

2.2. CLIENT LIBRARY

2.2.1. Overview

While invoking a microservice is typically a simple matter of sending a JSON or XML payload over HTTP, various considerations have led to the prevalence of specialized client libraries, particularly in a Spring Boot environment. These libraries provide integration with not only Spring Boot, but also many other tools and libraries often required in a microservice architecture.

2.2.2. Ribbon

`Ribbon` is an Inter-Process Communication (remote procedure calls) library with built-in client-side load balancers. The primary usage model involves REST calls with various serialization scheme support.

This reference architecture uses Ribbon, without relying on it for much intelligence. The main reason for including and using Ribbon is its prevalence in Spring Boot microservice applications, and relatedly, its support for and integration with various tools and libraries commonly used in such applications.

2.2.3. gRPC

The more modern **`gRPC`** is a replacement for Ribbon that's been developed by ***`Google`*** and adopted by a large number of projects.

While Ribbon uses simple text-based JSON or XML payloads over HTTP, gRPC relies on *Protocol Buffers* for faster and more compact serialization. The payload is sent over **HTTP/2** in binary form. The result is better performance and security, at the expense of compatibility and tooling support in the existing market.

2.3. SERVICE REGISTRY

2.3.1. Overview

Microservice architecture often implies dynamic scaling of individual services, in a private, hybrid or public cloud where the number and address of hosts cannot always be predicted or statically configured in advance. The solution is the use of a service registry as a starting point for discovering the deployed instances of each service. This will often be paired by a client library or load balancer layer that seamlessly fails over upon discovering that an instance no longer exists, and caches service registry lookups. Taking things one step further, integration between a client library and the

service registry can make this lookup and invoke process into a single step, and transparent to developers.

In modern cloud environments, such capability is often provided by the platform, and service replication and scaling is a core feature. This reference architecture is built on top of OpenShift, therefore benefiting from the **Kubernetes** Service abstraction.

2.3.2. Eureka

Eureka is a ***REST*** (REpresentational State Transfer) based service that is primarily used in the ***AWS*** cloud for locating services for the purpose of load balancing and failover of middle-tier servers.

Tight integration between Ribbon and Eureka allows declarative use of Eureka when the caller is using the Ribbon library.

2.3.3. Consul

Consul is a tool for discovering and configuring services in your infrastructure. It is provided both as part of the ***HashiCorp*** enterprise suite of software, as well as an open source component that is used in the Spring Cloud.

Integration with Ribbon within a Spring Cloud environment allows transparent and declarative lookups of services registered with Consul.

2.3.4. ZooKeeper

Apache ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

Once again, the support of ZooKeeper within Spring Cloud environments and integration with Ribbon allows declarative lookups of service instances before invocation.

2.3.5. OpenShift

In OpenShift, a Kubernetes service serves as an internal load balancer. It identifies a set of replicated pods in order to proxy the connections it receives to them. Additional backing pods can be added to, or removed from a service, while the service itself remains consistently available, enabling anything that depends on the service to refer to it through a consistent address.

Contrary to a third-party service registry, the platform in charge of service replication can provide a current and accurate report of service replicas at any moment. The service abstraction is also a critical platform component that is as reliable as the underlying platform itself. This means that the client does not need to keep a cache and account for the failure of the service registry itself. Ribbon can be declaratively configured to use OpenShift instead of a service registry, without any code changes.

2.4. LOAD BALANCER

2.4.1. Overview

For client calls to stateless services, high availability (HA) translates to a need to look up the service from a service registry, and load balance among available instances. The client libraries previously mentioned include the ability to combine these two steps, but OpenShift makes both actions

redundant by including load balancing capability in the service abstraction. OpenShift provides a single address where calls will be load balanced and redirected to an appropriate instance.

2.4.2. Ribbon

Ribbon allows load balancing among a static list of instances that are declared, or however many instances of the service that are discovered from a registry lookup.

2.4.3. gRPC

gRPC also provides load balancing capability within the same library layer.

2.4.4. OpenShift Service

OpenShift provides load balancing through its concept of service abstraction. The cluster IP address exposed by a service is an internal load balancer between any running replica pods that provide the service. Within the OpenShift cluster, the service name resolves to this cluster IP address and can be used to reach the load balancer. For calls from outside and when going through the router is not desirable, an external IP address can be configured for the service.

2.5. CIRCUIT BREAKER

2.5.1. Overview

The highly distributed nature of microservices implies a higher risk of failure of a remote call, as the number of such remote calls increases. The circuit breaker pattern can help avoid a cascade of such failures by isolating problematic services and avoiding damaging timeouts.

2.5.2. Hystrix

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.

Hystrix implements both the circuit breaker and bulkhead patterns.

2.6. EXTERNALIZED CONFIGURATION

2.6.1. Overview

Externalized configuration management solutions can provide an elegant alternative to the typical combination of configuration files, command line arguments, and environment variables that are used to make applications more portable and less rigid in response to outside changes. This capability is largely dependent on the underlying platform and is provided by *ConfigMaps* in OpenShift.

2.6.2. Spring Cloud Config

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

2.6.3. OpenShift ConfigMaps

ConfigMaps can be used to store fine-grained information like individual properties, or coarse-grained information like entire configuration files or JSON blobs. They provide mechanisms to inject containers with configuration data while keeping containers agnostic of OpenShift Container Platform.

2.7. DISTRIBUTED TRACING

2.7.1. Overview

For all its advantages, a microservice architecture is very difficult to analyze and troubleshoot. Each business request spawns multiple calls to, and between, individual services at various layers. Distributed tracing ties all individual service calls together, and associates them with a business request through a unique generated ID.

2.7.2. Sleuth/Zipkin

Spring Cloud Sleuth generates trace IDs for every call and span IDs at the requested points in an application. This information can be integrated with a logging framework to help troubleshoot the application by following the log files, or broadcast to a Zipkin server and stored for analytics and reports.

2.7.3. Jaeger

Jaeger, inspired by **Dapper** and **OpenZipkin**, is an open source distributed tracing system that fully conforms to the **Cloud Native Computing Foundation (CNCF)** *OpenTracing* standard. It can be used for monitoring microservice-based architectures and provides distributed context propagation and transaction monitoring, as well as service dependency analysis and performance / latency optimization.

2.8. PROXY/ROUTING

2.8.1. Overview

Adding a proxy in front of every service call enables the application of various filters before and after calls, as well as a number of common patterns in a microservice architecture, such as A/B testing. Static and dynamic routing rules can help select the desires version of a service.

2.8.2. Zuul

Zuul is an edge service that provides dynamic routing, monitoring, resiliency, security, and more. Zuul supports multiple routing models, ranging from declarative URL patterns mapped to a destination, to groovy scripts that can reside outside the application archive and dynamically determine the route.

2.8.3. Istio

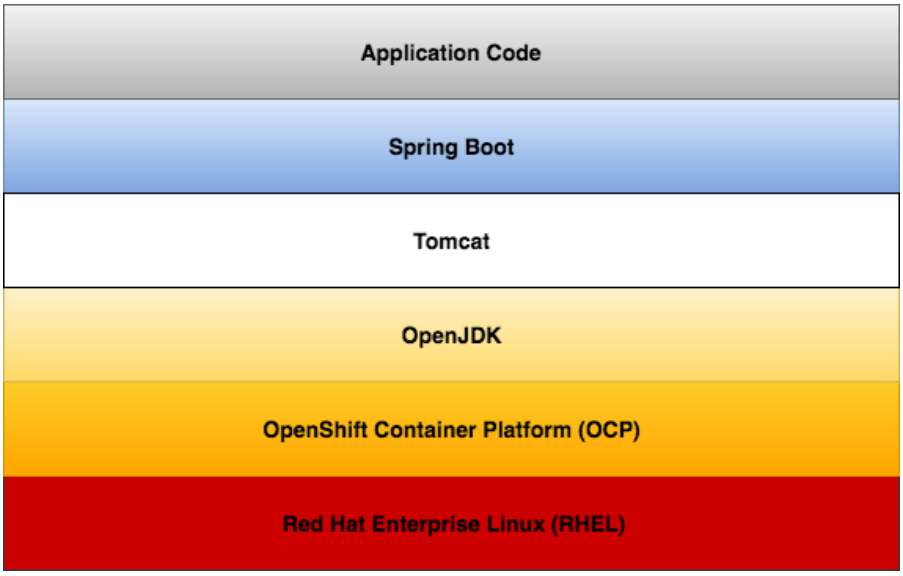
Istio is an open platform-independent service mesh that provides traffic management, policy enforcement, and telemetry collection. Istio is designed to manage communications between microservices and applications. Istio is still in pre-release stages.

Red Hat is a participant in the Istio project.

CHAPTER 3. REFERENCE ARCHITECTURE ENVIRONMENT

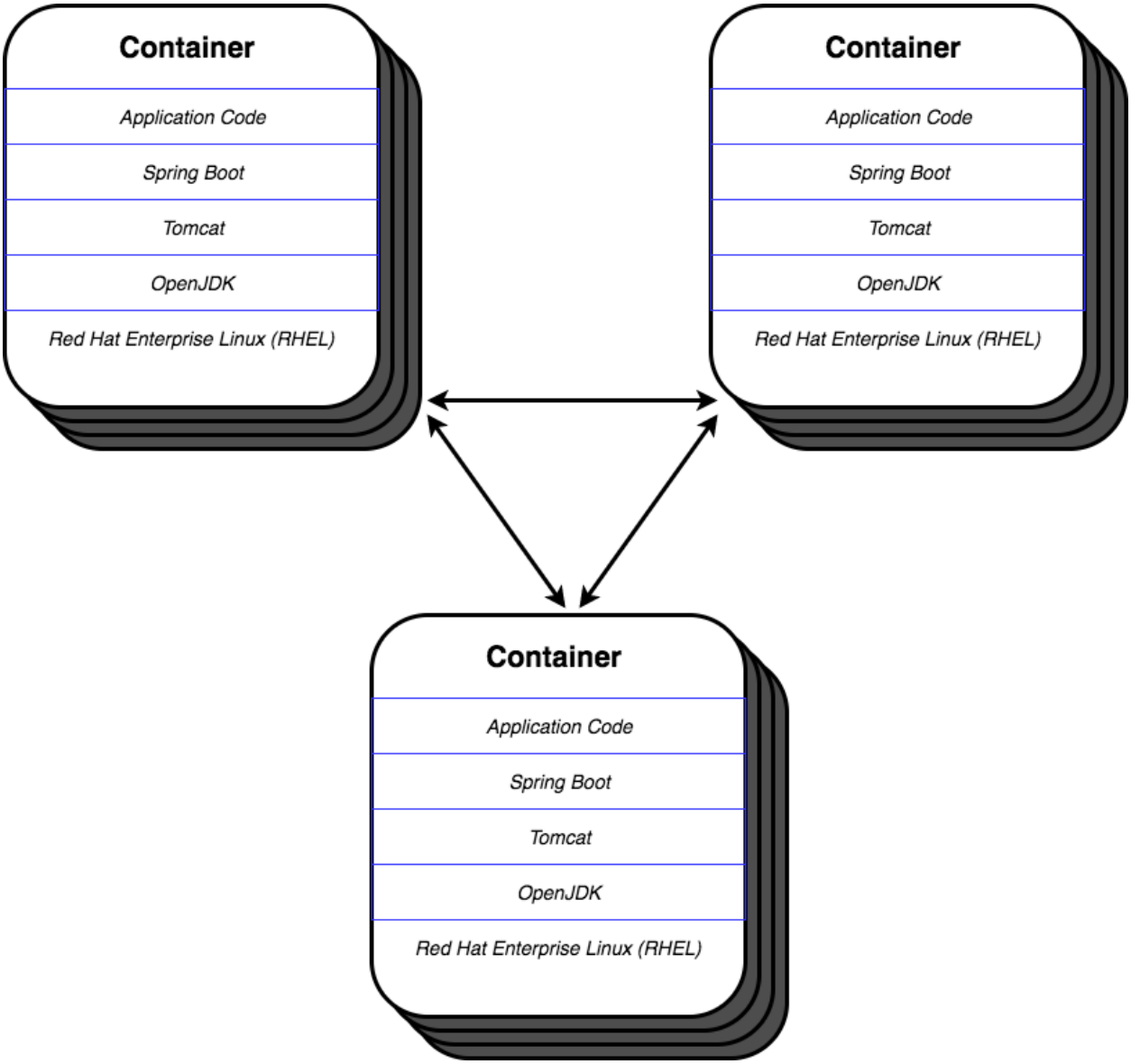
This reference architecture demonstrates an airline ticket search system built in the microservice architectural style. Each individual microservice is implemented as a *REST* service on top of Spring Boot with an embedded Tomcat server, deployed on an OpenShift image with a supported **OpenJDK**. The software stack of a typical microservice is as follows:

Figure 3.1. Microservice Software Stack



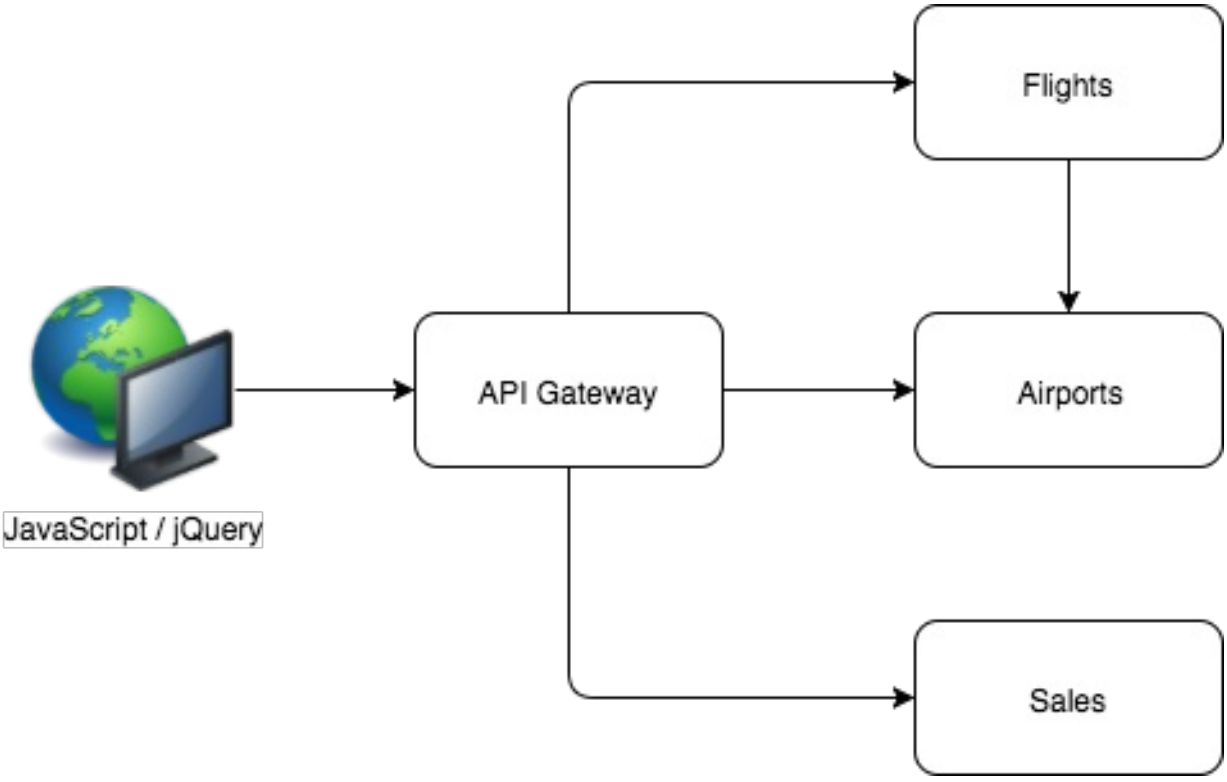
Each microservice instance runs in a container instance, with one container per OpenShift pod and one pod per service replica. At its core, an application built in the microservice architectural style consists of a number of replicated containers calling each other:

Figure 3.2. Container Software Stack



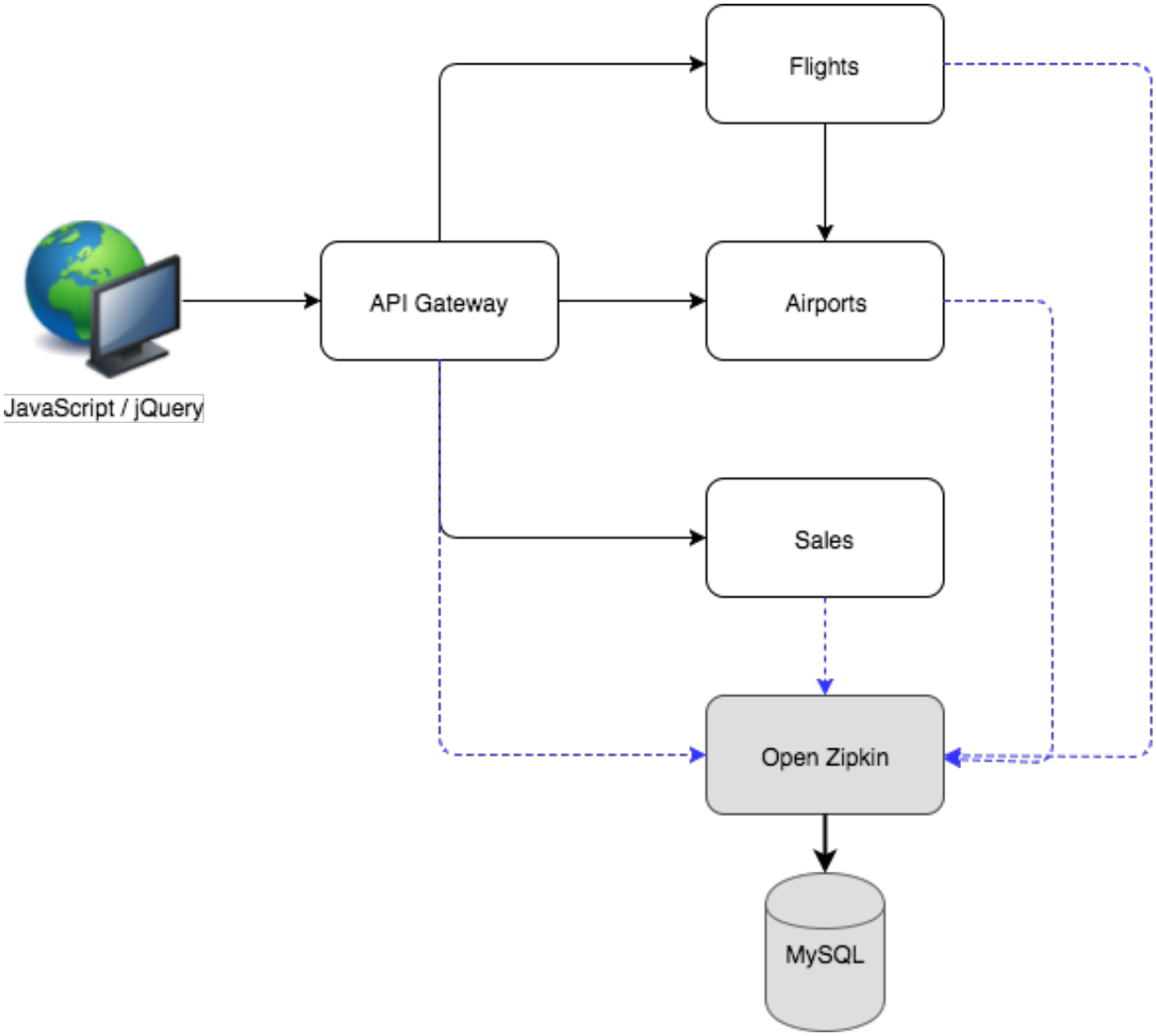
The core functionality of the application is provided by microservices, each fulfilling a single responsibility. One service acts as the API gateway, calling individual microservices and aggregating the response so it can be consumed easier.

Figure 3.3. Functional Diagram



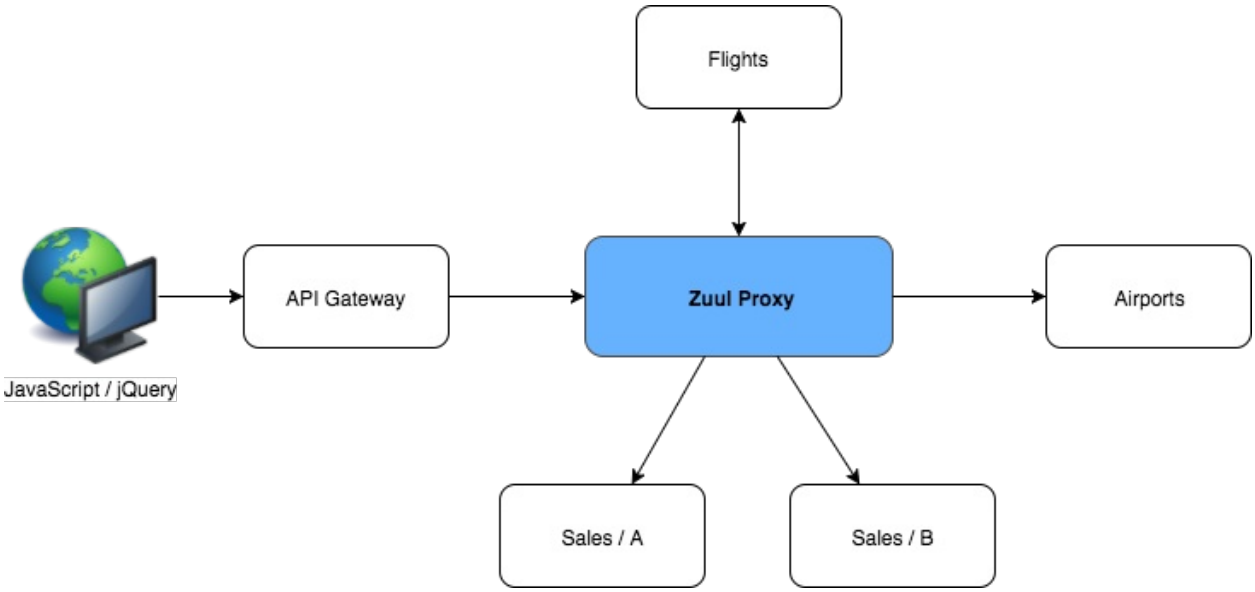
The architecture makes extended use of **`Spring Sleuth`** and **`OpenZipkin`** for distributed tracing. OpenZipkin runs as a separate service with a **`MySQL`** database used to persist its data, and it is called from every service in the application.

Figure 3.4. Zipkin Calls



Finally, the reference architecture uses **`Zuul`** as an edge service to provide static and dynamic routing. The result is that all service calls are actually directed to Zuul and it proxies the request as appropriate. This capability is leveraged to demonstrate **A/B testing** by providing an alternate version of the *Sales* service and making a runtime decision to use it for a group of customers.

Figure 3.5. Zuul Proxy



CHAPTER 4. CREATING THE ENVIRONMENT

4.1. OVERVIEW

This reference architecture can be deployed in either a production or a trial environment. In both cases, it is assumed that *ocp-master1* refers to one (or the only) OpenShift master host and that the environment includes two other OpenShift schedulable hosts with the host names of *ocp-node1* and *ocp-node2*. Production environments would have at least 3 master hosts to provide **High Availability (HA)** resource management, and presumably a higher number of working nodes.

It is further assumed that OpenShift Container Platform has been properly installed, and that a *Linux* user with *sudo* privileges has access to the host machines. This user can then set up an OpenShift user through its *identity providers*.

4.2. PROJECT DOWNLOAD

Download the source code and related artifacts for this reference architecture application from its public github repository:

```
$ git clone https://github.com/RHsyseng/spring-boot-msa-ocp.git
LambdaAir
```

Change directory to the root of this project. It is assumed that from this point on, all instructions are executed from inside the *LambdaAir* directory.

```
$ cd LambdaAir
```

4.3. SHARED STORAGE

This reference architecture environment uses Network File System (NFS) to make storage available to all OpenShift nodes.

Attach 2GB of storage and create a volume group for it, and two logical volumes of 1GB for each required persistent volume:

```
$ sudo pvcreate /dev/vdc
$ sudo vgcreate spring-boot-ocp /dev/vdc
$ sudo lvcreate -L 1G -n zipkin spring-boot-ocp
$ sudo lvcreate -L 1G -n zuul spring-boot-ocp
```

Create a corresponding mount directory for each logical volume and mount them.

```
$ sudo mkfs.ext4 /dev/spring-boot-ocp/zipkin
$ sudo mkdir -p /mnt/zipkin/mysql
$ sudo mount /dev/spring-boot-ocp/zipkin /mnt/zipkin/mysql

$ sudo mkfs.ext4 /dev/spring-boot-ocp/zuul
$ sudo mkdir -p /mnt/zuul/volume
$ sudo mount /dev/spring-boot-ocp/zuul /mnt/zuul/volume
```

Share these mounts with all nodes by configuring the */etc/exports* file on the NFS server, and make sure to restart the NFS service before proceeding.

4.4. OPENSHIFT CONFIGURATION

Create an OpenShift user, optionally with the same name, to use for creating the project and deploying the application. Assuming the use of HTPasswd as the authentication provider:

```
$ sudo htpasswd -c /etc/origin/master/htpasswd ocpAdmin
New password: PASSWORD
Re-type new password: PASSWORD
Adding password for user ocpAdmin
```

Grant OpenShift admin and cluster admin roles to this user, so it can create persistent volumes:

```
$ sudo oadm policy add-cluster-role-to-user admin ocpAdmin
$ sudo oadm policy add-cluster-role-to-user cluster-admin ocpAdmin
```

At this point, the new OpenShift user can be used to sign in to the cluster through the master server:

```
$ oc login -u ocpAdmin -p PASSWORD --server=https://ocp-
master1.xxx.example.com:8443

Login successful.
```

Create a new project to deploy this reference architecture application:

```
$ oc new-project lambdaair --display-name="Lambda Air" --
description="Spring Boot Microservices on Red Hat OpenShift Container
Platform 3"
Now using project "lambdaair" on server "https://ocp-
master1.xxx.example.com:8443".
```

4.5. ZIPKIN DEPLOYMENT

Zipkin uses MySQL database for storage, which in turn requires an OpenShift persistent volume to be created. Edit *Zipkin/zipkin-mysql-pv.json* and provide a valid NFS server and path, before proceeding. Once the file has been corrected, use it to create a persistent volume:

```
$ oc create -f Zipkin/zipkin-mysql-pv.json
persistentvolume "zipkin-mysql-data" created
```

Validate that the persistent volume is available:

```
$ oc get pv
NAME                CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
CLAIM                          REASON    AGE
zipkin-mysql-data   1Gi        RWO           Recycle
Available                                          1m
```

Once available, use the provided zipkin template to deploy both MySQL and Zipkin services:

```
$ oc new-app -f Zipkin/zipkin-mysql.yml
--> Deploying template "lambdaair/" for "Zipkin/zipkin-mysql.yml" to
project lambdaair
```

```
     ---------
     MySQL database service, with persistent storage. For more
information about using this template, including OpenShift
considerations, see https://github.com/sclorg/mysql-
container/blob/master/5.7/README.md.

     NOTE: Scaling to more than one replica is not supported. You must
have persistent volumes available in your cluster to use this template.

     The following service(s) have been created in your project:
zipkin-mysql.

            Username: zipkin
            Password: TwnDiEpoMqOGiJNb
       Database Name: zipkin
      Connection URL: mysql://zipkin-mysql:3306/

     For more information about using this template, including
OpenShift considerations, see https://github.com/sclorg/mysql-
container/blob/master/5.7/README.md.


     * With parameters:
        * Memory Limit=512Mi
        * Namespace=openshift
        * Database Service Name=zipkin-mysql
        * MySQL Connection Username=zipkin
        * MySQL Connection Password=TwnDiEpoMqOGiJNb # generated
        * MySQL root user Password=YJmmYOO3BVyX77wL # generated
        * MySQL Database Name=zipkin
        * Volume Capacity=1Gi
        * Version of MySQL Image=5.7

--> Creating resources ...
    secret "zipkin-mysql" created
    service "zipkin" created
    service "zipkin-mysql" created
    persistentvolumeclaim "zipkin-mysql" created
    configmap "zipkin-mysql-cnf" created
    configmap "zipkin-mysql-initdb" created
    deploymentconfig "zipkin" created
    deploymentconfig "zipkin-mysql" created
    route "zipkin" created
--> Success
    Run 'oc status' to view your app.
```

Note

The output above includes randomly generated passwords for the database that will be different each time. It is advisable to note down the passwords for your deployed database, in case it is later needed for troubleshooting.

You can use *oc status* to get a report, but for further details and to view the progress of the deployment, *watch* the pods as they get created and deployed:

```
$ watch oc get pods

Every 2.0s: oc get pods
Fri Jul 21 02:04:15 2017

NAME                     READY     STATUS              RESTARTS   AGE
zipkin-1-deploy          1/1   Running             0          2m
zipkin-1-sclgl           0/1   Running             0          2m
zipkin-mysql-1-deploy    1/1   Running             0          2m
zipkin-mysql-1-tv2v1     0/1   ContainerCreating   0          1m
```

It may take a few minutes for the deployment process to complete, at which point there should be two pods in the *Running* state:

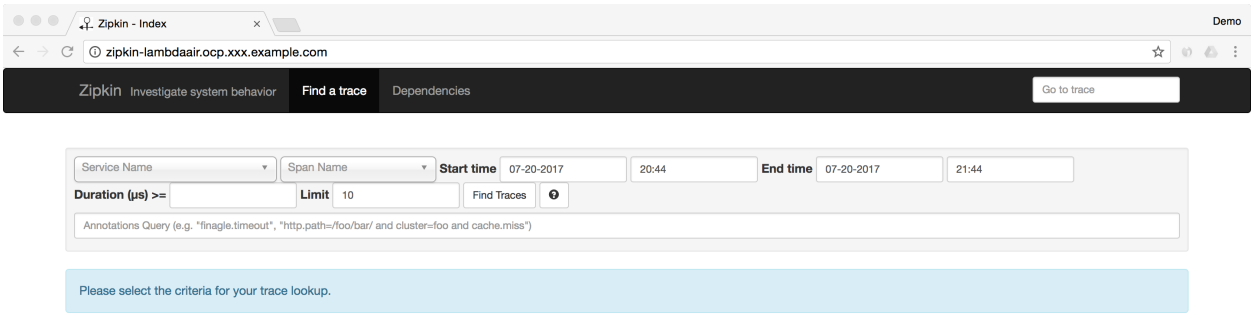
```
$ oc get pods
NAME                   READY     STATUS    RESTARTS   AGE
zipkin-1-k0dv6         1/1       Running   0          5m
zipkin-mysql-1-g44s7   1/1       Running   0          4m
```

Once the deployment is complete, you will be able to access the *Zipkin* console. Discover its address by querying the routes:

```
$ oc get routes
NAME      HOST/PORT
PATH      SERVICES   PORT      TERMINATION   WILDCARD
zipkin    zipkin-lambdaair.ocp.xxx.example.com             zipkin
9411                    None
```

Use the displayed URL to access the console from a browser and verify that it works correctly:

Figure 4.1. Zipkin Console



4.6. SERVICE DEPLOYMENT

To deploy the Spring Boot services implemented in Java, navigate to the root of each directory and use *Maven* to build the project, while passing the *openshift* profile to deploy the built image to OpenShift. Repeat this process for *Airports*, *Flights*, *Presentation*, *Sales*, *Salesv2*, and *Zuul*:

```
$ pushd Airports
~/LambdaAir/Airports ~/LambdaAir

$ mvn clean fabric8:deploy -Popenshift

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
------------
[INFO] Building Lambda Air 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------
------------
...
[INFO] OpenShift platform detected
[INFO] Using project: lambdaair
[INFO] Creating a Service from openshift.yml namespace lambdaair name
airports
[INFO] Created Service: target/fabric8/applyJson/lambdaair/service-
airports.json
[INFO] Updating ImageStream flights from openshift.yml
[INFO] Creating a DeploymentConfig from openshift.yml namespace
lambdaair name airports
[INFO] Created DeploymentConfig:
target/fabric8/applyJson/lambdaair/deploymentconfig-airports.json
[INFO] F8: HINT: Use the command oc get pods -w to watch your pods
start up
[INFO] ------------------------------------------------------------
------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
------------
[INFO] Total time: 01:05 min
[INFO] Finished at: 2017-07-21T05:20:06+00:00
[INFO] Final Memory: 55M/890M
[INFO] ------------------------------------------------------------
------------
```

Once the maven build has completed, query the pods and make sure the new service is running:

```
$ oc get pods
NAME                   READY     STATUS      RESTARTS   AGE
airports-1-4xkfv        1/1       Running     0          41s
airports-s2i-1-build    0/1       Completed   0          1m
zipkin-1-k0dv6         1/1       Running     0          1h
zipkin-mysql-1-g44s7   1/1       Running     0          1h
```

Repeat this process for the other 5 services:

```
$ popd && pushd Flights
~/LambdaAir/Flights ~/LambdaAir

$ mvn clean fabric8:deploy -Popenshift

[INFO] Scanning for projects...
[INFO]
```

```
[INFO] ------------------------------------------------------------
------------
[INFO] Building Lambda Air 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------
------------
...
```

After *Flights*, continue on and switch the directory to *Presentation*, *Sales*, *Salesv2*, and *Zuul*, using the same maven command to build and deploy each one. Verify that the service pod is running after each build.

Once all services have been built and deployed, there should be a total of 8 running pods, including the 2 Zipkin pods from before and a new pod for each of the 6 services:

```
$ oc get pods
NAME                       READY     STATUS      RESTARTS   AGE
airports-1-72kng           1/1       Running     0          18m
airports-s2i-1-build       0/1       Completed   0          21m
flights-1-4xkfv            1/1       Running     0          15m
flights-s2i-1-build        0/1       Completed   0          16m
presentation-1-k2xlz       1/1       Running     0          10m
presentation-s2i-1-build   0/1       Completed   0          11m
sales-1-fqxjd              1/1       Running     0          7m
sales-s2i-1-build          0/1       Completed   0          8m
salesv2-1-s1wq0            1/1       Running     0          5m
salesv2-s2i-1-build        0/1       Completed   0          6m
zipkin-1-k0dv6             1/1       Running     0          1h
zipkin-mysql-1-g44s7       1/1       Running     0          1h
zuul-1-2jkj0               1/1       Running     0          1m
zuul-s2i-1-build           0/1       Completed   0          2m
```

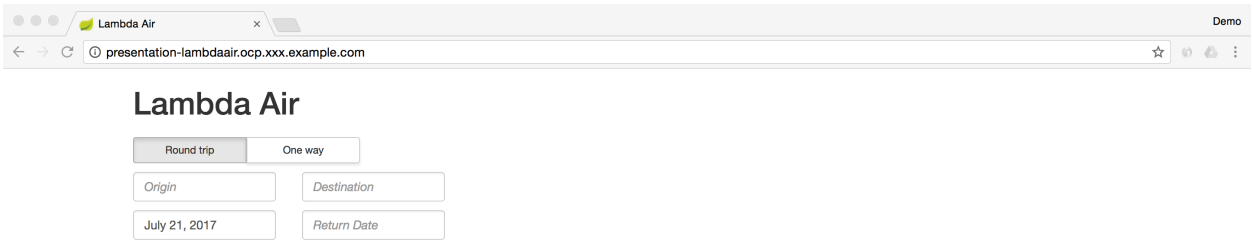
4.7. FLIGHT SEARCH

The *presentation* service also creates a route. Once again, list the routes in the OpenShift project:

```
$ oc get routes
NAME           HOST/PORT                                       PATH
SERVICES       PORT      TERMINATION   WILDCARD
presentation   presentation-lambdaair.ocp.xxx.example.com
presentation   8080                    None
zipkin         zipkin-lambdaair.ocp.xxx.example.com
zipkin         9411                    None
```

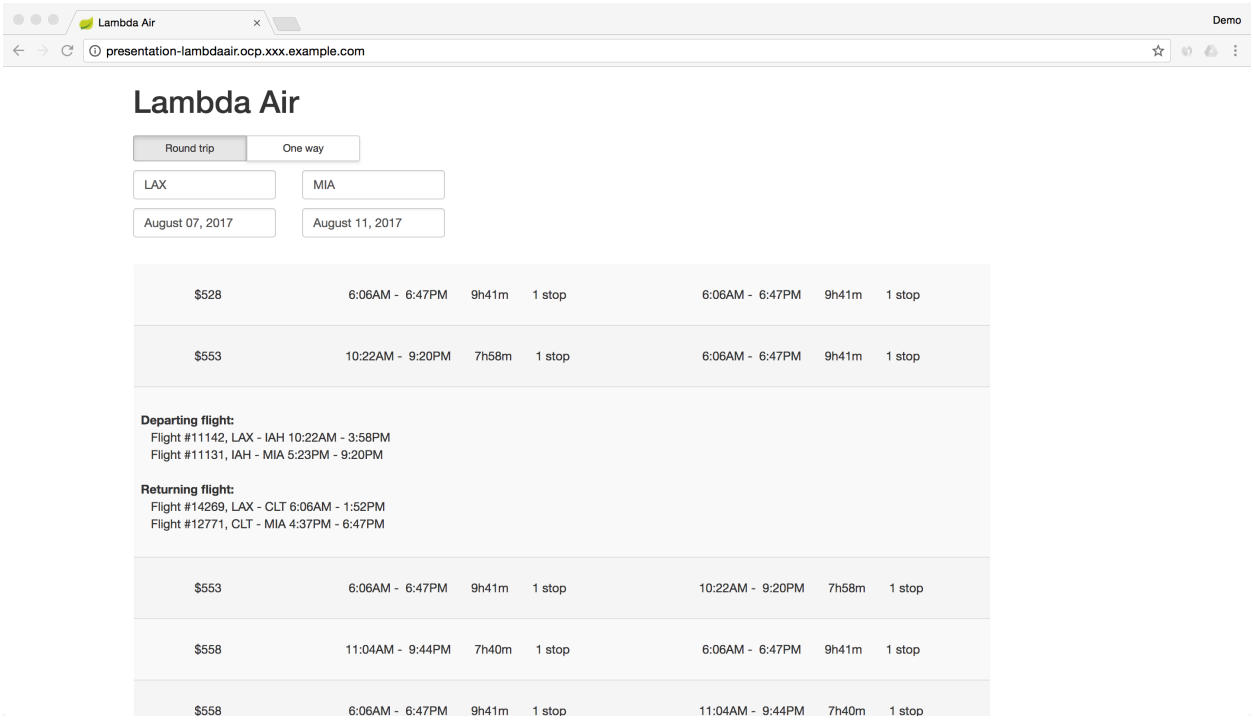
Use the URL of the route to access the HTML application from a browser, and verify that it comes up:

Figure 4.2. Lambda Air Landing Page



Search for a flight by entering values for each of the four fields. The first search may take a bit longer, so wait a few seconds for the response:

Figure 4.3. Lambda Air Flight Search



4.8. EXTERNAL CONFIGURATION

The *Presentation* service configures *Hystrix* with a thread pool size of 20 in its application properties. Confirm this by searching the logs of the presentation pod after a flight search operation and verify that the batch size is the same:

```
$ oc logs presentation-1-k2xlz | grep batch
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 20
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 13
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 20
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 13
tickets
```

```
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 20
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 13
tickets
```

Create a new *application.yml* file that assumes a higher number of *Sales* service pods relative to *Presentation* pods:

```
$ vi application.yml
```

Enter the following values:

```
hystrix:
  threadpool:
    SalesThreads:
      coreSize: 30
      maxQueueSize: 300
      queueSizeRejectionThreshold: 300
```

Create a *configmap* using the *oc* utility based on this file:

```
$ oc create configmap presentation --from-file=application.yml

configmap "presentation" created
```

Edit the *Presentation* deployment config and mount this *ConfigMap* as /deployments/config, where it will automatically be part of the Spring Boot application classpath:

```
$ oc edit dc presentation
```

Add a new volume with an arbitrary name, such as *config-volume*, that references the previously created *configmap*. The *volumes* definition is a child of the *template spec*. Next, create a volume mount under the container to reference this volume and specify where it should be mounted. The final result is as follows, with the new lines highlighted:

```
...
        resources: {}
        securityContext:
          privileged: false
        terminationMessagePath: /dev/termination-log
        volumeMounts:
        - name: config-volume
          mountPath: /deployments/config
      volumes:
        - name: config-volume
          configMap:
            name: presentation
      dnsPolicy: ClusterFirst
      restartPolicy: Always
...
```

Once the deployment config is modified and saved, OpenShift will deploy a new version of the service that will include the overriding properties. This change is persistent and pods created in the future with this new version of the deployment config will also mount the yaml file.

List the pods and note that a new pod is being created to reflect the change in the deployment config, which is the mounted file:

```
$ oc get pods
NAME                         READY     STATUS      RESTARTS   AGE
airports-1-72kng             1/1       Running     0          18m
airports-s2i-1-build         0/1       Completed   0          21m
flights-1-4xkfv              1/1       Running     0          15m
flights-s2i-1-build          0/1       Completed   0          16m
presentation-1-k2xlz         1/1       Running     0          10m
presentation-2-deploy        0/1       ContainerCreating   0          3s
presentation-s2i-1-build     0/1       Completed   0          11m
sales-1-fqxjd                1/1       Running     0          7m
sales-s2i-1-build            0/1       Completed   0          8m
salesv2-1-s1wq0              1/1       Running     0          5m
salesv2-s2i-1-build          0/1       Completed   0          6m
zipkin-1-k0dv6               1/1       Running     0          1h
zipkin-mysql-1-g44s7         1/1       Running     0          1h
zuul-1-2jkj0                 1/1       Running     0          1m
zuul-s2i-1-build             0/1       Completed   0          2m
```

Wait until the second version of the pod has started in the running state. The first version will be terminated and subsequently removed:

```
$ oc get pods
NAME                         READY     STATUS      RESTARTS   AGE
...
presentation-2-pxx85         1/1       Running     0          5m
presentation-s2i-1-build     0/1       Completed   0          1h
...
```

Once this has happened, use the browser to do one or several more flight searches. Then verify the updated thread pool size by searching the logs of the new presentation pod and verify the batch size:

```
$ oc logs presentation-2-pxx85 | grep batch
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 30
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 3
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 30
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 3
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 30
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 3
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 30
tickets
... c.r.r.o.b.l.p.s.API_GatewayController    : Will price a batch of 3
tickets
```

Notice that with the mounted overriding properties, pricing happens in concurrent batches of 30 instead of 20 items now.

4.9. A/B TESTING

Copy the groovy script provided in the *Zuul* project over to the shared storage for this service:

```
$ cp Zuul/misc/ABTestingFilterBean.groovy /mnt/zuul/volume/
```

Create a persistent volume for the *Zuul* service. External groovy scripts placed in this location can provide dynamic routing.

```
$ oc create -f Zuul/misc/zuul-pv.json
persistentvolume "groovy" created
```

Also create a persistent volume claim:

```
$ oc create -f Zuul/misc/zuul-pvc.json
persistentvolumeclaim "groovy-claim" created
```

Verify that the claim is bound to the persistent volume:

```
$ oc get pvc
NAME           STATUS    VOLUME              CAPACITY   ACCESSMODES
AGE
groovy-claim   Bound     groovy              1Gi        RWO
7s
zipkin-mysql   Bound     zipkin-mysql-data   1Gi        RWO
2h
```

Attach the persistent volume claim to the deployment config as a directory called groovy on the root of the filesystem:

```
$ oc volume dc/zuul --add --name=groovy --type=persistentVolumeClaim --
claim-name=groovy-claim --mount-path=/groovy
deploymentconfig "zuul" updated
[bmozaffa@middleware-master LambdaAir]$ oc get pods
NAME                       READY     STATUS              RESTARTS
AGE
airports-1-72kng           1/1       Running             0
1h
airports-s2i-1-build       0/1       Completed           0
1h
flights-1-4xkfv            1/1       Running             0
1h
flights-s2i-1-build        0/1       Completed           0
1h
presentation-2-pxx85       1/1       Running             0
32m
presentation-s2i-1-build   0/1       Completed           0
1h
sales-1-fqxjd              1/1       Running             0
1h
sales-s2i-1-build          0/1       Completed           0
1h
salesv2-1-s1wq0            1/1       Running             0
1h
salesv2-s2i-1-build        0/1       Completed           0
```

```
1h
zipkin-1-k0dv6              1/1       Running             0
2h
zipkin-mysql-1-g44s7        1/1       Running             0
2h
zuul-1-2jkj0                1/1       Running             0
1h
zuul-2-deploy               0/1       ContainerCreating   0
4s
zuul-s2i-1-build            0/1       Completed           0
1h
```

Once again, the change prompts a new deployment and terminates the original *zuul* pod, once the new version is started up and running.

Wait until the second version of the pod reaches the running state:

```
$ oc get pods | grep zuul
zuul-2-gz7hl                1/1       Running     0          7m
zuul-s2i-1-build            0/1       Completed   0          1h
```

Return to the browser and perform one or more flight searches. Then return to the OpenShift environment and look at the log for the zuul pod.

If the IP address received from your browser ends in an odd number, the groovy script filters pricing calls and sends them to version 2 of the *sales* service instead. This will be clear in the *zuul* log:

```
$ oc logs zuul-2-gz7hl
...
... groovy.ABTestingFilterBean               : Caller IP address is
10.3.116.79
Running filter
... groovy.ABTestingFilterBean               : Caller IP address is
10.3.116.79
Running filter
```

In this case, the logs from *salesv2* will show tickets being priced with a modified algorithm:

```
$ oc logs salesv2-1-s1wq0
... c.r.r.o.b.l.sales.service.Controller     : Priced ticket at 463
with lower hop discount
... c.r.r.o.b.l.sales.service.Controller     : Priced ticket at 425
with lower hop discount
... c.r.r.o.b.l.sales.service.Controller     : Priced ticket at 407
with lower hop discount
... c.r.r.o.b.l.sales.service.Controller     : Priced ticket at 549
with lower hop discount
... c.r.r.o.b.l.sales.service.Controller     : Priced ticket at 509
with lower hop discount
... c.r.r.o.b.l.sales.service.Controller     : Priced ticket at 598
with lower hop discount
... c.r.r.o.b.l.sales.service.Controller     : Priced ticket at 610
with lower hop discount
```

If that is not the case and your IP address ends in an even number, it will still be printed but the *Running filter* statement will not appear:

```
$ oc logs zuul-2-gz7hl
...
... groovy.ABTestingFilterBean              : Caller IP address is
10.3.116.78
... groovy.ABTestingFilterBean              : Caller IP address is
10.3.116.78
... groovy.ABTestingFilterBean              : Caller IP address is
10.3.116.78
... groovy.ABTestingFilterBean              : Caller IP address is
10.3.116.78
```

In this case, you can change the filter criteria to send IP addresses with an even digit to the new version of pricing algorithm, instead of the odd ones:

```
$ vi /mnt/zuul/volume/ABTestingFilterBean.groovy
```

```
...
if( lastDigit % 2 == 0 )
{
   //Even IP address will be filtered
    true
}
else
{
   //Odd IP address won't be filtered
    false
}
...
```

Deploy a new version of the *zuul* service to pick up the updated groovy script:

```
$ oc rollout latest zuul
deploymentconfig "zuul" rolled out
```

Once the new pod is running, do a flight search again and check the logs. The calls to pricing should go to the *salesv2* service now, and logs should appear as previously described.

CHAPTER 5. DESIGN AND DEVELOPMENT

5.1. OVERVIEW

The source code for the *Lambda Air* application is made available in a public github repository. This chapter briefly covers each microservice and its functionality while reviewing the pieces of the software stack used in the reference architecture.

5.2. RESOURCE LIMITS

OpenShift allows administrators to set constraints to limit the number of objects or amount of compute resources that are used in each project. While these constraints apply to projects in the aggregate, each pod may also request minimum resources and/or be constrained with limits on its memory and CPU use.

The OpenShift template provided in the project repository uses this capability to request that at least 20% of a CPU core and 200 megabytes of memory be made available to its container. Twice the processing power and four times the memory may be provided to the container, if necessary and available, but no more than that will be assigned.

```
resources:
  limits:
    cpu: "400m"
    memory: "800Mi"
  requests:
    cpu: "200m"
    memory: "200Mi"
```

When the fabric8 Maven plugin is used to create the image and direct edits to the deployment configuration are not convenient, resource fragments can be used to provide the desired snippets. This application provides *deployment.yml* files to leverage this capability and set resource requests and limits on Spring Boot projects:

```
spec:
  replicas: 1
  template:
    spec:
      containers:
        - resources:
            requests:
              cpu: '200m'
              memory: '400Mi'
            limits:
              cpu: '400m'
              memory: '800Mi'
```

Control over the memory and processing use of individual services is often critical. Proper configuration of these values, as specified above, is seamless to the deployment and administration process. However, it can be helpful to set up resource quotas in projects for the purpose of enforcing their inclusion in pod deployment configurations.

5.3. SPRING BOOT REST SERVICE

5.3.1. Overview

The *Airports* service is the simplest microservice of the application, which makes it a good point of reference for building a basic Spring Boot REST service.

5.3.2. Spring Boot Application Class

To designate a Java project as a Spring Boot application, include a Java class that is annotated with *SpringBootApplication* and implements the default Java *main* method:

```
package com.redhat.refarch.obsidian.brownfield.lambdaair.airports;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

It is also good practice to declare the application name, which can be done as part of the common application properties. This application uses application.yml files that begin with each project's name:

```
spring:
  application:
    name: airports
```

5.3.3. Maven Project File

Each microservice project includes a **Maven** *POM* file, which in addition to declaring the module properties and dependencies, also includes a profile definition to use **fabric8-maven-plugin** to create and deploy an OpenShift image.

The *POM* file uses a property to declare the base image containing the operating system and **Java Development Kit (JDK)**. All the services in this application build on top of a **Red Hat Enterprise Linux (RHEL)** base image, containing a supported version of **OpenJDK**:

```
<properties>
...
    <fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
openshift</fabric8.generator.from>
</properties>
```

To easily include the dependencies for a simple Spring Boot application that provides a REST service, declare the following two artifacts:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

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

Every service in this application also declares a dependency on the **`Spring Boot Actuator`** component, which includes a number of additional features to help monitor and manage your application.

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

When a dependency on the *Actuator* is declared, *fabric8* generates default OpenShift health probes that communicate with Actuator services to determine whether a service is running and ready to service requests:

```
    livenessProbe:
      failureThreshold: 3
      httpGet:
        path: /health
        port: 8080
        scheme: HTTP
      initialDelaySeconds: 180
      periodSeconds: 10
      successThreshold: 1
      timeoutSeconds: 1
    readinessProbe:
      failureThreshold: 3
      httpGet:
        path: /health
        port: 8080
        scheme: HTTP
      initialDelaySeconds: 10
      periodSeconds: 10
      successThreshold: 1
      timeoutSeconds: 1
```

5.3.4. Spring Boot REST Controller

To receive and process REST requests, include a Java class annotated with *RestController*:

```
...
import org.springframework.web.bind.annotation.RestController;

@RestController
public class Controller
```

Specify the listening port for this service in the application properties:

```
server:
  port: 8080
```

Each REST operation is implemented by a Java method. Business operations typically require specifying request arguments:

```
@RequestMapping( value = "/airports", method = RequestMethod.GET )
public Collection<Airport> airports(
  @RequestParam( value = "filter", required = false ) String filter)
{
 ...
```

5.3.5. Startup Initialization

The *Airports* service uses eager initialization to load airport data into memory at the time of startup. This is implemented through an *ApplicationListener* that listens for a specific type of event:

```
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.stereotype.Component;

@Component
public class ApplicationInitialization implements
ApplicationListener<ContextRefreshedEvent>
{
 @Override
 public void onApplicationEvent(ContextRefreshedEvent
contextRefreshedEvent)
```

5.4. RIBBON AND LOAD BALANCING

5.4.1. Overview

The *Flights* service has a similar structure to that of the *Airports* service, but relies on, and calls the *Airports* service. As such, it makes use of *Ribbon* and the generated OpenShift service for high availability.

5.4.2. RestTemplate and Ribbon

To quickly and easily declare the required dependencies to use Ribbon, add the following artifact as a Maven dependency:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-ribbon</artifactId>
   <version>1.2.5.RELEASE</version>
</dependency>
```

This application also makes use of the Jackson JSR 310 libraries to correctly serialize and deserialize Java 8 date objects:

```
<dependency>
```

```
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
    <version>2.8.8</version>
</dependency>
```

Declare a load balanced *RestTemplate* and use injection to assign it to a field:

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

@Autowired
private RestTemplate restTemplate;
```

For outgoing calls, simply use the restTemplate field:

```
Airport[] airportArray = restTemplate.getForObject(
"http://zuul/airports/airports", Airport[].class );
```

The service address provided as the host part of the URL is resolved through Ribbon, based on values provided in application properties:

```
zuul:
  ribbon:
    listOfServers: zuul:8080
```

In this case, Ribbon expects a list of statically defined service addresses, but a single one is provided with the hostname of *zuul* with port 8080. Zuul uses the second part of the address, the root web context, to redirect the request through statically or dynamic routing, as explained later in this document.

The provided hostname of *zuul* is the OpenShift service name, and is resolved to the cluster IP address of the service, then routed to an internal OpenShift load balancer. The OpenShift service name is determined when a service is created using the *oc* tool, or when deploying an image using the fabric8 Maven plugin, it is declared in the service yaml file.

Ribbon is effectively not load balancing requests, but rather sending them to an OpenShift internal load balancer, which is aware of replication and failure of service instances, and can redirect the request properly.

5.5. SPRING BOOT MVC

5.5.1. Overview

The Presentation service makes minimal use of **`Spring MVC`** to serve the client-side HTML application to calling browsers.

5.5.2. ModelAndView Mapping

After including the required Maven dependencies, the Presentation service provides a simple Controller class to serve HTML content:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping( "/" )
public class WebRequestController
{
 @GetMapping
 public ModelAndView list()
 {
  return new ModelAndView( "index" );
 }
}
```

This class declares that the service will be listening to HTTP GET requests on the root context of the application. It serves the index file, provided as an index.html in the templates directory, back to the browser.

Templates typically allow parameter substitution, but as previously mentioned, this service makes very minimal use of *Spring MVC* functionality.

5.5.3. Bower Package Manager

The *Presentation* service uses Bower package manager to declare, download and update JavaScript libraries. Libraries, versions and components to download (or rather, those to ignore) are specified in a bower JSON file. Running *bower install* downloads the declared libraries to the bower_components directory, which can in turn be imported in the HTML application.

5.5.4. PatternFly

The HTML application developed for this reference architecture uses PatternFly to provide consistent visual design and improved user experience.

PatternFly stylesheets are imported in the main html:

```
<!-- PatternFly Styles -->
<link href="bower_components/patternfly/dist/css/patternfly.min.css" rel="stylesheet"
   media="screen, print"/>
<link href="bower_components/patternfly/dist/css/patternfly-additions.min.css" rel="stylesheet"
   media="screen, print"/>
```

The associated JavaScript is also included in the header:

```
<!-- PatternFly -->
<script src="bower_components/patternfly/dist/js/patternfly.min.js"></script>
```

5.5.5. JavaScript

The presentation tier of this application is built in HTML5 and relies heavily on JavaScript. This includes using *ajax* calls to the *API gateway*, as well as minor changes to HTML elements that visible and displayed to the user.

5.5.5.1. jQuery UI

Some features of the jQuery UI library, including autocomplete for airport fields, are utilized in the presentation layer.

5.5.5.2. jQuery Bootstrap Table

To display flight search results in a dynamic table with pagination, and the ability to expand each row to reveal more data, a jQuery Bootstrap Table library is included and utilized.

5.6. HYSTRIX

5.6.1. Overview

The *Presentation* service includes a second listening controller, this time a REST controller, that acts as an API gateway. The API gateway makes simple REST calls to the *Airports* service, similar to the previously discussed Flights service, but also calls the *Sales* service to get pricing information and uses a different pattern for this call. *Hystrix* is used to avoid a large number of hung threads and lengthy timeouts when the *Sales* service is down. Instead, flight information can be returned without providing a ticket price. The reactive interface of *Hystrix* is also leveraged to implement parallel processing.

5.6.2. Circuit Breaker

Hystrix provides multiple patterns for the use of its API. The *Presentation* service uses *Hystrix* commands for its outgoing calls to *Sales*. This is implemented as a *Hystrix* command:

```
private class PricingCall extends HystrixCommand<Itinerary>
{
 private Flight flight;

 PricingCall(Flight flight)
 {
  super( HystrixCommandGroupKey.Factory.asKey( "Sales" ),
HystrixThreadPoolKey.Factory.asKey( "SalesThreads" ) );
  this.flight = flight;
 }

 @Override
 protected Itinerary run() throws Exception
 {
  try
  {
   return restTemplate.postForObject( "http://zuul/sales/price",
flight, Itinerary.class );
  }
  catch( Exception e )
  {
   logger.log( Level.SEVERE, "Failed!", e );
   throw e;
  }
 }

 @Override
```

```
 protected Itinerary getFallback()
 {
  logger.warning( "Failed to obtain price, " +
getFailedExecutionException().getMessage() + " for " + flight );
  return new Itinerary( flight );
 }
}
```

After being instantiated and provided a flight for pricing, the command takes one of two routes. When successful and able to reach the service being called, the *run* method is executed which uses the now-familiar pattern of calling the service through Ribbon and the OpenShift service abstraction. However, if an error prevents us from reaching the *Sales* service, *getFallback()* provides a chance to recover from the error, which in this case involves returning the itinerary without a price.

The fallback scenario can happen simply because the call has failed, but also in cases when the circuit is open (tripped). Configure Hystrix as part of the service properties to specify when a thread should time out and fail, as well as the queue used for concurrent processing of outgoing calls.

To configure the command timeout for a specific command (and not globally), the *HystrixCommandKey* is required. This defaults to the command class name, which is *PricingCall* in this implementation.

Configure thread pool properties for this specific thread pool by using the specified thread pool key of *SalesThreads*.

hystrix.command.PricingCall.execution.isolation.thread.timeoutInMilliseconds: 2000

```
hystrix:
  threadpool:
    SalesThreads:
      coreSize: 20
      maxQueueSize: 200
      queueSizeRejectionThreshold: 200
```

5.6.3. Concurrent Reactive Execution

We assume technical considerations have led to the *Sales* service accepting a single flight object in its API. To reduce lag time and take advantage of horizontal scaling, the service uses Reactive Commands for batch processing of pricing calls.

The API gateway service queries and stores the configured thread pool size as a field:

```
@Value("${hystrix.threadpool.SalesThreads.coreSize}")
private int threadSize;
```

The thread size is later used as the batch size for the concurrent calls to calculate the price of a flight:

```
int batchLimit = Math.min( index + threadSize, itineraries.length );
for( int batchIndex = index; batchIndex < batchLimit; batchIndex++ )
{
 observables.add( new PricingCall( itineraries[batchIndex]
).toObservable() );
}
```

The Reactive zip operator is used to process the calls for each batch concurrently and store results in a collection. The number of batches depends on the ratio of total flights found to the batch size, which is set to 20 in this service configuration.

5.7. OPENSHIFT CONFIGMAP

5.7.1. Overview

While considering the concurrent execution of pricing calls, it should be noted that the API gateway is itself multi-threaded, so the batch size is not the final determinant of the thread count. In this example of a batch size of 20, with a maximum queue size of 200 and the same threshold leading to rejection, receiving more than 10 concurrent query calls can lead to errors. These values should be fine-tuned based on realistic expectations of load as well as the horizontal scaling of the environment.

This configuration can be externalized by creating a *ConfigMap* for each OpenShift environment, with overriding values provided in a properties file that is then provided to all future pods.

5.7.2. Property File Mount

Refer to the steps in creating the environment for detailed instructions on how to create an external application properties and mounting it in the pod. The property file is placed in the application class path and the provided values supersede those of the application.

5.8. ZIPKIN

5.8.1. Overview

This reference architecture uses Spring Sleuth to collect and broadcast tracing data to OpenZipkin, which is deployed as an OpenShift service and backed by a persistent MySQL database image. The tracing data can be queried from the *Zipkin* console, which is exposed through an OpenShift route. Logging integration is also possible, although not demonstrated, to use trace IDs to tie in together the distributed execution of the same business request.

5.8.2. MySQL Database

This reference architecture uses the provided and support OpenShift MySQL image to store persistent zipkin data.

5.8.2.1. Persistent Volume

To enable persistent storage for the MySQL database image, this reference architecture creates and mounts a logical volume that is expose through **NFS**. An OpenShift persistent volume exposes the storage to the image. Once the storage is set up and shared by the NFS server:

```
$ oc create -f zipkin-mysql-pv.json
persistentvolume "zipkin-mysql-data" created
```

5.8.2.2. MySQL Image

This reference architecture provides a single OpenShift template to create a database image, the

database schema required for *OpenZipkin*, and the *OpenZipkin* image itself. This template relies on the MySQL image definition that is available by default in the *openshift* project.

5.8.2.3. Database Initialization

This reference architecture demonstrates the use of lifecycle hooks to initialize a database after the pod has been created. Specifically, a *post hook* is used as follows:

```
      recreateParams:
        post:
          failurePolicy: Abort
          execNewPod:
            containerName: mysql
            command:
            - /bin/sh
            - -c
            - hostname && sleep 10 && /opt/rh/rh-mysql57/root/usr/bin/mysql -h
$DATABASE_SERVICE_NAME -u $MYSQL_USER -D $MYSQL_DATABASE -
p$MYSQL_PASSWORD -P 3306 < /docker-entrypoint-initdb.d/init.sql && echo Initialized database
            env:
            - name: DATABASE_SERVICE_NAME
              value: ${DATABASE_SERVICE_NAME}
            volumes:
            - mysql-init-script
```

Notice that the hook uses the command line *mysql* utility to run the SQL script located */docker-entrypoint-initdb.d/init.sql*. Some database images standardize on this location for initialization, in which case a lifecycle hook is not required.

The SQL script to create the schema is embedded in the template as a config map. It is then declared as a volume and mounted at its final path under */docker-entrypoint-initdb.d/*.

5.8.3. OpenZipkin Image

The template uses the image provided by OpenZipkin:

```
image: openzipkin/zipkin:1.19.2
```

Required parameters for *OpenZipkin* to access the associated *MySQL* database are either configured or generated as part of the same template. Database passwords are randomly generated by OpenShift as part of the template and stored in a *secret*, which makes them inaccessible to users and administrators in the future. That is why a template message is printed to allow a one-time access to the database password for monitoring and troubleshooting purposes.

To create the *Zipkin* service:

```
$ oc new-app -f LambdaAir/Zipkin/zipkin-mysql.yml
--> Deploying template "lambdaair/" for "zipkin-mysql.yml" to project
lambdaair



     ---------
     MySQL database service, with persistent storage. For more
information about using this template, including OpenShift
considerations, see https://github.com/sclorg/mysql-
```

```
container/blob/master/5.7/README.md.

     NOTE: Scaling to more than one replica is not supported. You must
have persistent volumes available in your cluster to use this template.

     The following service(s) have been created in your project:
zipkin-mysql.

            Username: zipkin
            Password: Y4hScBSPH5bAhDL2
       Database Name: zipkin
      Connection URL: mysql://zipkin-mysql:3306/

     For more information about using this template, including
OpenShift considerations, see https://github.com/sclorg/mysql-
container/blob/master/5.7/README.md.


     * With parameters:
        * Memory Limit=512Mi
        * Namespace=openshift
        * Database Service Name=zipkin-mysql
        * MySQL Connection Username=zipkin
        * MySQL Connection Password=Y4hScBSPH5bAhDL2 # generated
        * MySQL root user Password=xYVNsuRXRV5xqu4A # generated
        * MySQL Database Name=zipkin
        * Volume Capacity=1Gi
        * Version of MySQL Image=5.7

--> Creating resources ...
    secret "zipkin-mysql" created
    service "zipkin" created
    service "zipkin-mysql" created
    persistentvolumeclaim "zipkin-mysql" created
    configmap "zipkin-mysql-cnf" created
    configmap "zipkin-mysql-initdb" created
    deploymentconfig "zipkin" created
    deploymentconfig "zipkin-mysql" created
    route "zipkin" created
--> Success
    Run 'oc status' to view your app.
```

5.8.4. Spring Sleuth

While the *Zipkin* service allows distributed tracing data to be aggregated, persisted and used for reporting, this application relies on *Spring Sleuth* to correlate calls and send data to Zipkin.

Integration with Ribbon and other framework libraries make it very easy to use Spring Sleuth in the application. Include the libraries by declaring a dependency in the project Maven file:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-sleuth-zipkin</artifactId>
   <version>1.2.1.RELEASE</version>
</dependency>
```

Also specify in the application properties the percentage of requests that should be traced, as well as the address to the zipkin server. Once again, we rely on the OpenShift service abstract to reach zipkin.

```
spring:
  sleuth:
    sampler:
      percentage: 1.0
  zipkin:
    baseUrl: http://zipkin/
```

The percentage value of 1.0 means 100% of requests will be captured.

These two steps are enough to collect tracing data, but a Tracer object can also be injected into the code for extended functionality. While every remote call can produce and store a trace by default, adding a tag can help to better understand zipkin reports. The service also creates and demarcates tracing *spans* of interest to collect more meaningful tracing data.

5.8.4.1. Baggage Data

While *Spring Sleuth* is primarily intended as a distributed tracing tool, its ability to correlate distributed calls can have other practical uses as well. Every created *span* allows the attachment of arbitrary data, called a **baggage item**, that will be automatically inserted into the HTTP header and seamlessly carried along with the business request from service to service, for the duration of the *span*. This application is interested in making the original caller's IP address available to every microservice. In an OpenShift environment, the calling IP address is stored in the HTTP header under a standard key. To retrieve and set this value on the *span*:

```
querySpan.setBaggageItem( "forwarded-for", request.getHeader( "x-
forwarded-for" ) );
```

This value will later be accessible from any service within the same call *span* under the header key of *baggage-forwarded-for*. It is used by the *Zuul* service in a **`Groovy`** script to perform dynamic routing.

5.9. ZUUL

5.9.1. Overview

This reference architecture uses *Zuul* as a central proxy for all calls between microservices. By default, the service uses static routing as defined in its application properties:

```
zuul:
  routes:
    airports:
      path: /airports/**
      url: http://airports:8080/
    flights:
      path: /flights/**
      url: http://flights:8080/
    sales:
      path: /sales/**
      url: http://sales:8080/
```

The *path* provided in the above rules uses the first part of the web address to determine the service to be called, and the rest of the address as the context.

5.9.2. A/B Testing

To implement A/B testing, the Salesv2 service introduces a minor change in the algorithm for calculating fares. Dynamic routing is provided by *Zuul* through a filter that filters some of the requests.

Calls to other services are not filtered:

```
if(
!RequestContext.currentContext.getRequest().getRequestURI().matches("/s
ales.*") )
{
    //Won't filter this request URL
    false
}
```

Only those calls to the *Sales* service that originate from an IP address ending in an odd digit are filtered:

```
String caller = new HTTPRequestUtils().getHeaderValue("baggage-
forwarded-for");
logger.info("Caller IP address is " + caller)
int lastDigit = caller.reverse().take(1) as Integer
if( lastDigit % 2 == 0 )
{
    //Even IP address won't be filtered
    false
}
else
{
    //Odd IP address will be filtered
    true
}
```

If the caller has an odd digit at the end of their IP address, the request is rerouted. That means the run method of the filter is executed, which changes the route host:

```
@Override
Object run() {
    println("Running filter")
    RequestContext.currentContext.routeHost = new
URL("http://salesv2:8080")
}
```

To enable dynamic routing without changing application code, shared storage is made available to the OpenShift nodes and a persistent volume is created and claimed. With the volume set up and the groovy filter in place, the OpenShift deployment config can be adjusted administratively to mount a directory as a volume:

```
$ oc volume dc/zuul --add --name=groovy --type=persistentVolumeClaim --
claim-name=groovy-claim --mount-path=/groovy
```

This results in all groovy scripts under the groovy directory being found. The *zuul* application code anticipates the introduction of dynamic routing filters by seeking and applying any groovy script under this path:

```
for( Resource resource : new
PathMatchingResourcePatternResolver().getResources(
"file:/groovy/.groovy" ) ) { logger.info( "Found and will load groovy
script " + resource.getFilename() ); sources.add( resource ); } if(
sources.size() == 1 ) { logger.info( "No groovy script found under
/groovy/.groovy" );
}
```

CHAPTER 6. CONCLUSION

Spring Boot applications designed based on the microservices architectural style are often very easy to migrate to, and deploy on, **Red Hat® OpenShift Container Platform 3**. Most of the open source libraries commonly found in such applications can run on OpenShift without any changes.

This paper and its accompanying technical implementation seek to serve as a useful reference for such a migration, while providing a proof of concept that can easily be replicated in a customer environment.

APPENDIX A. AUTHORSHIP HISTORY

Revision	Release Date	Author(s)
1.0	Aug 2017	Babak Mozaffari

APPENDIX B. CONTRIBUTORS

We would like to thank the following individuals for their time and patience as we collaborated on this process. This document would not have been possible without their many contributions.

Contributor	Title	Contribution
John Clingan	Senior Principal Product Manager	Subject Matter Review, Technical Content Review
Charles Moulliard	Manager, Software Engineering	Technical Content Review
Burr Sutter	Manager, Developer Evangelism	Technical Content Review
Christophe Laprun	Principal Software Engineer	Technical Content Review

APPENDIX C. REVISION HISTORY

Revision 1.0-0	**August 2017**	**BM**