Reference Architectures 2017

WildFly Swarm Microservices on Red Hat OpenShift Container Platform 3

Last Updated: 2017-12-09

Reference Architectures 2017 WildFly Swarm Microservices on Red Hat OpenShift Container Platform 3

Babak Mozaffari
refarch-feedback@redhat.com

Abstract

This reference architecture demonstrates the design, development, and deployment of WildFly Swarm 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 `WildFly Swarm` microservices on `Red Hat® OpenShift Container Platform 3`. The reference application is built with a number of open source components, commonly found in WildFly Swarm applications.

`Red Hat OpenShift Application Runtimes (RHOAR)` is an ongoing effort by Red Hat to provide official OpenShift images with a combination of fully supported Red Hat software and popular third-party open-source components. With the first public release, most core libraries of WildFly Swarm are fully supported, while libraries like `Hystrix` have been tested and verified on top of supported components including Undertow, OpenJDK, and the base image itself.

The reference architecture serves as a potential blueprint for cloud-native application development on WildFy Swarm, OpenShift Container Platform and other commonly associated software. This architecture can also help guide the migration and deployment of existing WildFly Swarm microservices to 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 WildFly Swarm. WildFly Swarm can use various components by selectively declaring dependency on a number of *fractions*. This paper focuses on the use of WildFly Swarm with **`Undertow`** as the underlying web listener layer, 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 different client libraries. These libraries in turn support many other tools and frameworks 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.

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.

2.2.4. JAX-RS Client

The **`JAX-RS Client API`** is an addition to JAX-RS 2.0, and a standard supported specification of Java EE 7. This client API allows integration of third-party libraries through its filters, interceptors and features.

This reference architecture uses the JAX-RS Client API to call services. Jaeger provides integration with the JAX-RS Client API and allows distributed tracing by intercepting calls and inserting the required header information.

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.

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.

2.3.4. ZooKeeper

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

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.

This reference architecture builds microservices on top of OpenShift and uses the OpenShift service capability to perform the role of a service registry.

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 can be 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.

This reference architecture uses Jaeger for instrumenting services and Jaeger service containers in the back-end to collect and produce distributed tracing reports.

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 desired 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.

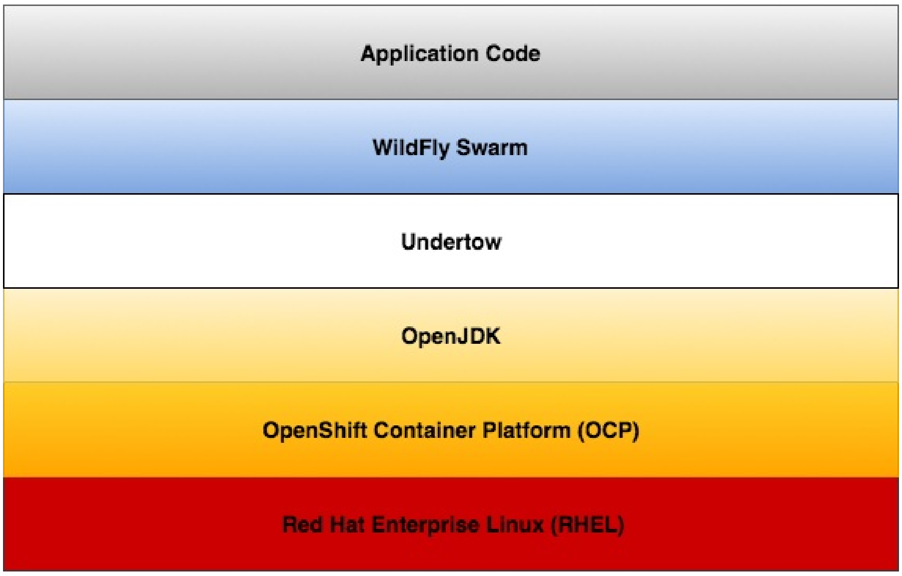
2.8.4. Custom Proxy

While Istio remains under development, this application builds a custom reverse proxy component capable of both static and dynamic routing rules. The reverse proxy component developed as part of this reference architecture application relies on the open-source HTTP-Proxy-Servlet project for proxy capability. The servlet is extended to introduce a mapping framework, allowing for simple declarative mapping comparable to Zuul, in addition to support for JavaScript. Rules written in JavaScript can be very simple, or take full advantage of standard JavaScript and the JVM surrounding it. The mapping framework supports a chained approach, so scripts can simply focus on overriding static rules when necessary.

CHAPTER 3. REFERENCE ARCHITECTURE ENVIRONMENT

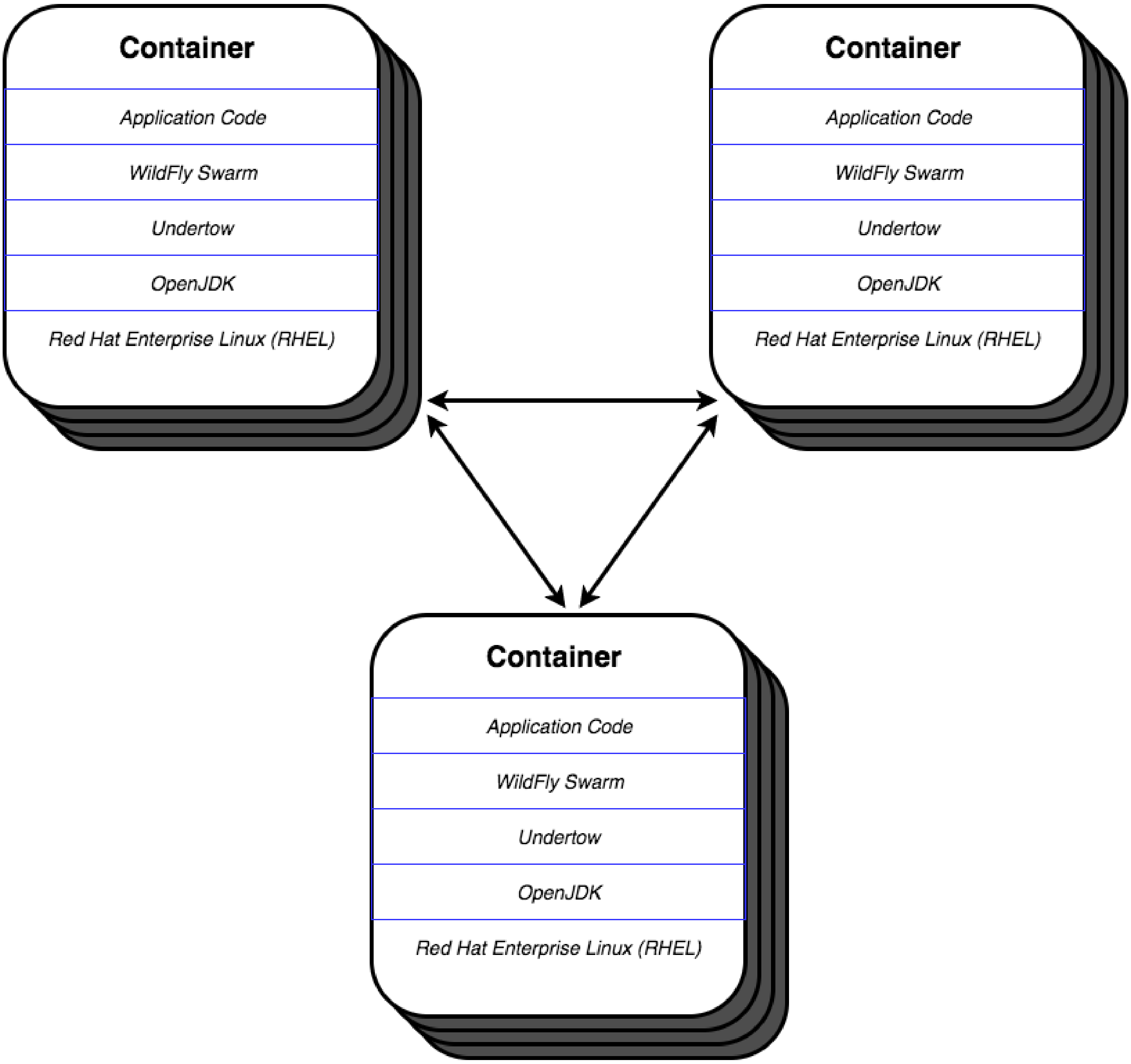
This reference architecture demonstrates an airline ticket search system built in the microservice architectural style. Each individual microservice is implemented as a *REST* service on top of WildFly Swarm with Undertow as the web server, deployed on an OpenShift image with a supported **`OpenJDK`**.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

Figure 3.1. Microservice Software Stack



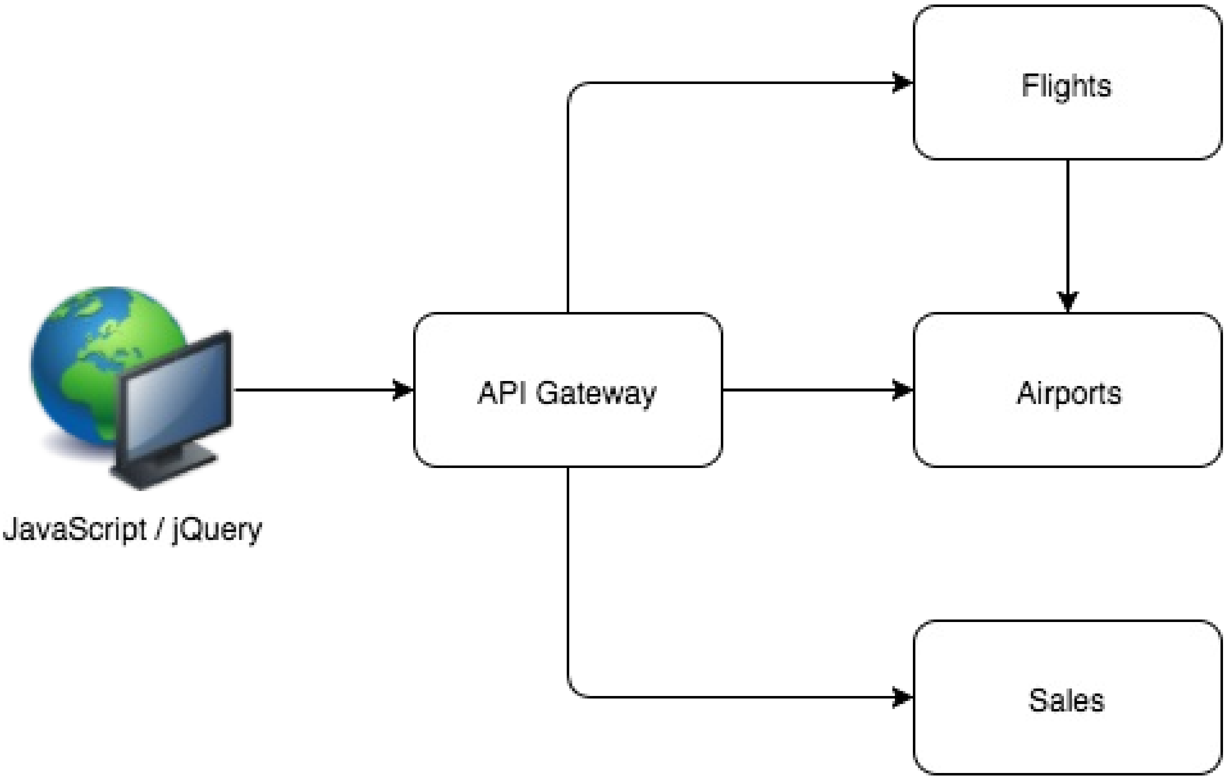
Each microservice instance runs in a container instance, with one container per OpenShift pod and one pod per service replica. At its core, an application built in the microservice architectural style consists of a number of replicated containers calling each other:

Figure 3.2. Container Software Stack



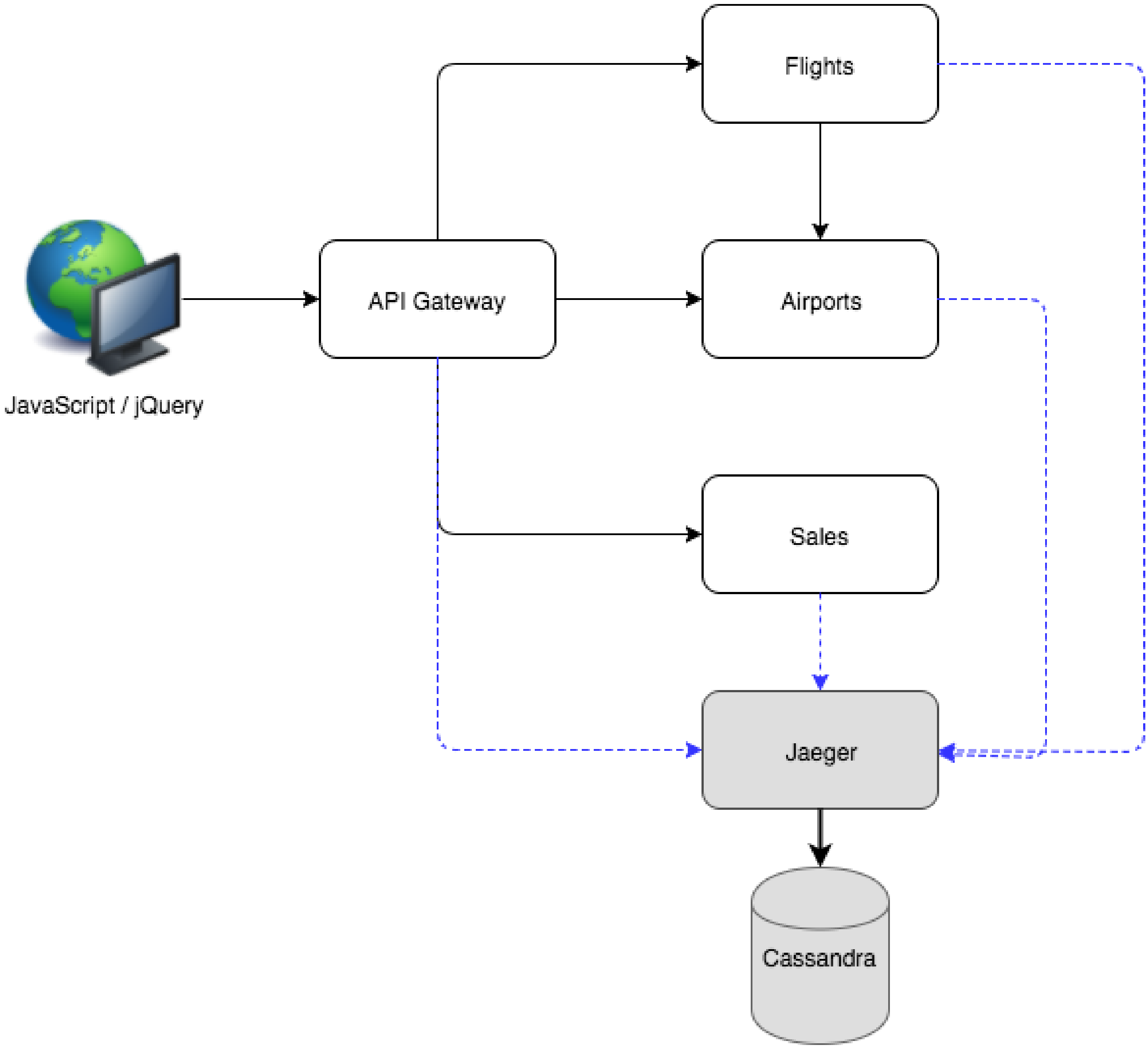
The core functionality of the application is provided by microservices, each fulfilling a single responsibility. One service acts as the API gateway, calling individual microservices and aggregating the response so it can be consumed easier.

Figure 3.3. Functional Diagram



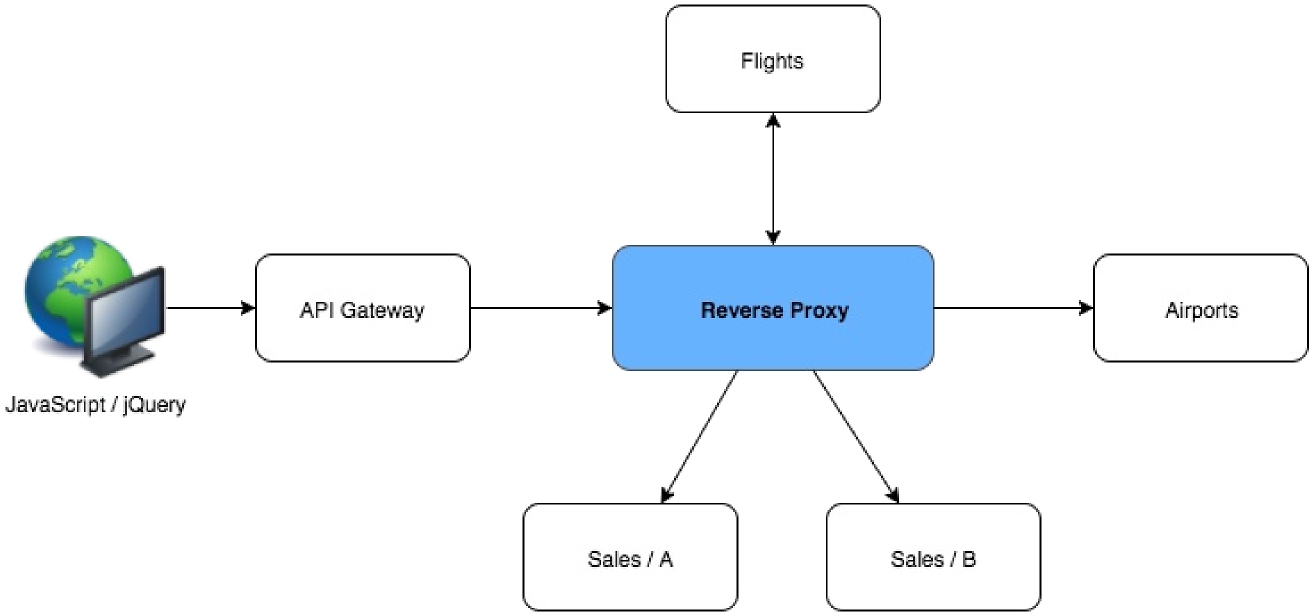
The architecture makes extended use of **`Jaeger`** for distributed tracing. The Jaeger production templates are used to run the backend service with a **`Cassandra`** datastore, and tracing data is sent to it from every service in the application.

Figure 3.4. Jaeger Calls



Finally, the reference architecture uses an edge service to provide static and dynamic routing. The result is that all service calls are actually directed to the reverse proxy and it forwards 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. Reverse 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/wildfly-swarm-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 3GB of storage and create a volume group for it, as well as a logical volume of 1GB for each required persistent volume:

```
$ sudo pvcreate /dev/vdc
$ sudo vgcreate wildfly-swarm /dev/vdc
$ sudo lvcreate -L 1G -n cassandra-data wildfly-swarm
$ sudo lvcreate -L 1G -n cassandra-logs wildfly-swarm
$ sudo lvcreate -L 1G -n edge wildfly-swarm
```

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

```
$ sudo mkfs.ext4 /dev/wildfly-swarm/cassandra-data
$ sudo mkdir -p /mnt/wildfly-swarm/cassandra-data
$ sudo mount /dev/wildfly-swarm/cassandra-data /mnt/wildfly-
swarm/cassandra-data

$ sudo mkfs.ext4 /dev/wildfly-swarm/cassandra-logs
$ sudo mkdir -p /mnt/wildfly-swarm/cassandra-logs
$ sudo mount /dev/wildfly-swarm/cassandra-logs /mnt/wildfly-
swarm/cassandra-logs
```

```
$ sudo mkfs.ext4 /dev/wildfly-swarm/edge
$ sudo mkdir -p /mnt/wildfly-swarm/edge
$ sudo mount /dev/wildfly-swarm/edge /mnt/wildfly-swarm/edge
```

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="WildFly Swarm Microservices on Red Hat OpenShift Container
Platform 3"
Now using project "lambdaair" on server "https://ocp-
master1.xxx.example.com:8443".
```

4.5. JAEGER DEPLOYMENT

Jaeger uses the Cassandra database for storage, which in turn requires OpenShift persistent volumes to be created. Edit *Jaeger/jaeger-pv.yml* and provide a valid NFS server and path for each entry, before proceeding. Once the file has been corrected, use it to create the six persistent volumes:

```
$ oc create -f Jaeger/jaeger-pv.yml
persistentvolume "cassandra-pv-1" created
persistentvolume "cassandra-pv-2" created
persistentvolume "cassandra-pv-3" created
persistentvolume "cassandra-pv-4" created
persistentvolume "cassandra-pv-5" created
persistentvolume "cassandra-pv-6" created
```

Validate that the persistent volumes are available:

```
$ oc get pv
NAME               CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
AGE
cassandra-pv-1     1Gi        RWO           Recycle         Available
11s
cassandra-pv-2     1Gi        RWO           Recycle         Available
11s
cassandra-pv-3     1Gi        RWO           Recycle         Available
11s
cassandra-pv-4     1Gi        RWO           Recycle         Available
11s
cassandra-pv-5     1Gi        RWO           Recycle         Available
11s
cassandra-pv-6     1Gi        RWO           Recycle         Available
11s
```

With the persistent volumes in place, use the provided version of the Jaeger production template to deploy both the Jaeger server and the Cassandra database services. The template also uses a volume claim template to dynamically create a data and log volume claim for each of the three pods:

```
volumeClaimTemplates:
- metadata:
    name: cassandra-data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
- metadata:
    name: cassandra-logs
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

```
$ oc new-app -f Jaeger/jaeger-production-template.yml

--> Deploying template "swarm/jaeger-template" for "Jaeger/jaeger-
production-template.yml" to project lambdaair

     Jaeger
     ---------
     Jaeger Distributed Tracing Server

     * With parameters:
        * Jaeger Service Name=jaeger
        * Image version=0.6
        * Jaeger Cassandra Keyspace=jaeger_v1_dc1
        * Jaeger Zipkin Service Name=zipkin

--> Creating resources ...
    service "cassandra" created
    statefulset "cassandra" created
    job "jaeger-cassandra-schema-job" created
    deployment "jaeger-collector" created
```

```
    service "jaeger-collector" created
    service "zipkin" created
    deployment "jaeger-query" created
    service "jaeger-query" created
    route "jaeger-query" created
--> Success
    Run 'oc status' to view your app.
```

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

NAME                                   READY     STATUS    RESTARTS    AGE
cassandra-0                            1/1       Running      0          4m
cassandra-1                            1/1       Running      2          4m
cassandra-2                            1/1       Running      3          4m
jaeger-cassandra-schema-job-7d58m      0/1       Completed    0          4m
jaeger-collector-418097188-b090z       1/1       Running      4          4m
jaeger-query-751032167-vxr3w           1/1       Running      3          4m
```

It may take a few minutes for the deployment process to complete, at which point there should be five pods in the *Running* state with a database loading job that is completed.

Next, deploy the Jaeger agent. This reference architecture deploys the agent as a single separate pod:

```
$ oc new-app Jaeger/jaeger-agent.yml

--> Deploying template "swarm/jaeger-jaeger-agent" for "Jaeger/jaeger-
agent.yml" to project lambdaair

--> Creating resources ...
    deploymentconfig "jaeger-agent" created
    service "jaeger-agent" created
--> Success
    Run 'oc status' to view your app.
```

NOTE

The Jaeger agent may be deployed in multiple ways, or even bypassed entirely through direct HTTP calls to the collector. Another option is bundling the agent as a sidecar to every microservice, as documented in the Jaeger project itself. Select an appropriate approach for your production environment

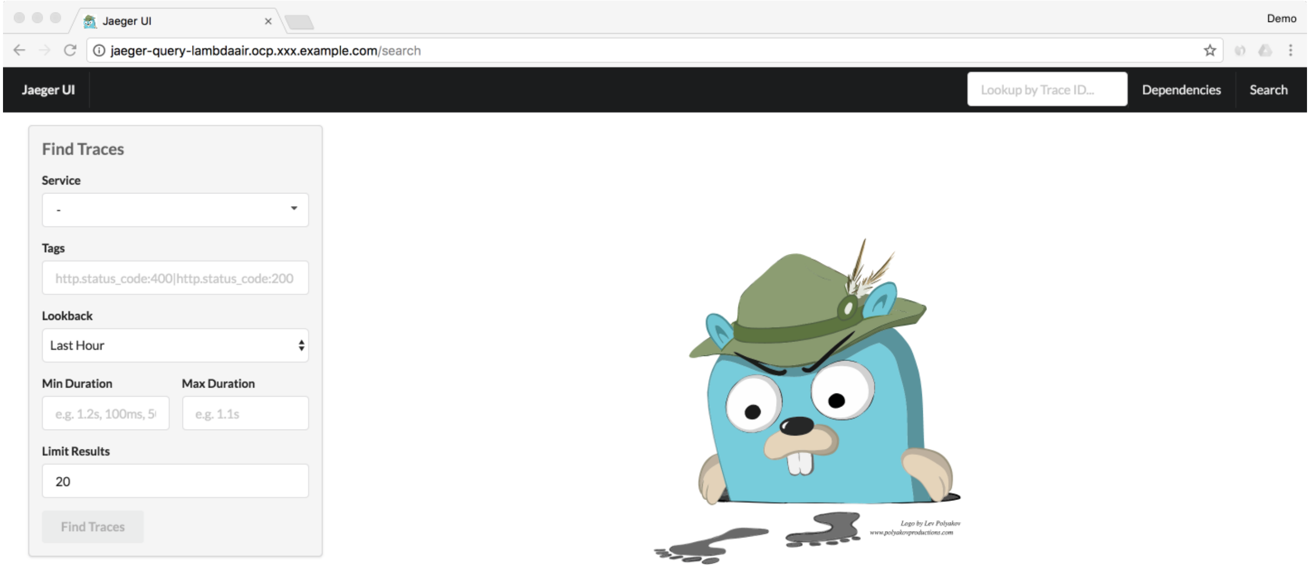
Next, to access the Jaeger console, first discover its address by querying the route:

```
$ oc get routes

NAME            HOST/PORT
PATH        SERVICES          PORT        TERMINATION    WILDCARD
jaeger-query    jaeger-query-lambdaair.ocp.xxx.example.com
jaeger-query    <all>         edge/Allow      None
```

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

Figure 4.1. Jaeger UI



4.6. SERVICE DEPLOYMENT

To deploy a WildFly Swarm service, use *Maven* to build the project, with the *fabric8:deploy* target for the *openshift* profile to deploy the built image to OpenShift. For convenience, an aggregator *pom* file has been provided at the root of the project that delegates the same Maven build to all 6 configured modules:

```
$ mvn clean fabric8:deploy -Popenshift

[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------------------------------------------------
---------
[INFO] Building Lambda Air 1.0-SNAPSHOT
[INFO] ----------------------------------------------------------------
---------
...
...
...
[INFO] --- fabric8-maven-plugin:3.5.30:deploy (default-cli) @ aggregation
---
[WARNING] F8: No such generated manifest file
/Users/bmozaffa/RedHatDrive/SysEng/Microservices/WildFlySwarm/wildfly-
swarm-msa-ocp/target/classes/META-INF/fabric8/openshift.yml for this
project so ignoring
[INFO] ----------------------------------------------------------------
---------
[INFO] Reactor Summary:
[INFO]
[INFO] Lambda Air ..................................... SUCCESS [02:26
min]
[INFO] Lambda Air ..................................... SUCCESS [04:18
min]
[INFO] Lambda Air ..................................... SUCCESS [02:07
min]
[INFO] Lambda Air ..................................... SUCCESS [02:42
```

```
min]
[INFO] Lambda Air ..................................... SUCCESS [01:17
min]
[INFO] Lambda Air ..................................... SUCCESS [01:13
min]
[INFO] Lambda Air ..................................... SUCCESS [
1.294 s]
[INFO] ----------------------------------------------------------------
---------
[INFO] BUILD SUCCESS
[INFO] ----------------------------------------------------------------
---------
[INFO] Total time: 14:16 min
[INFO] Finished at: 2017-12-08T16:57:11-08:00
[INFO] Final Memory: 81M/402M
[INFO] ----------------------------------------------------------------
---------
```

Once all services have been built and deployed, there should be a total of 11 running pods, including the 5 Jaeger pods from before, and a new pod for each of the 6 services:

```
$ oc get pods
NAME                                  READY     STATUS      RESTARTS    AGE
airports-1-bn1gp                      1/1       Running     0           24m
airports-s2i-1-build                  0/1       Completed   0           24m
cassandra-0                           1/1       Running     0           55m
cassandra-1                           1/1       Running     2           55m
cassandra-2                           1/1       Running     3           55m
edge-1-nlb4b                          1/1       Running     0           12m
edge-s2i-1-build                      0/1       Completed   0           13m
flights-1-n0lbx                       1/1       Running     0           11m
flights-s2i-1-build                   0/1       Completed   0           11m
jaeger-agent-1-g8s9t                  1/1       Running     0           39m
jaeger-cassandra-schema-job-7d58m     0/1       Completed   0           55m
jaeger-collector-418097188-b090z      1/1       Running     4           55m
jaeger-query-751032167-vxr3w          1/1       Running     3           55m
presentation-1-dscwm                  1/1       Running     0           1m
presentation-s2i-1-build              0/1       Completed   0           1m
sales-1-g96zm                         1/1       Running     0           4m
sales-s2i-1-build                     0/1       Completed   0           5m
sales2-1-36hww                        1/1       Running     0           3m
sales2-s2i-1-build                    0/1       Completed   0           4m
```

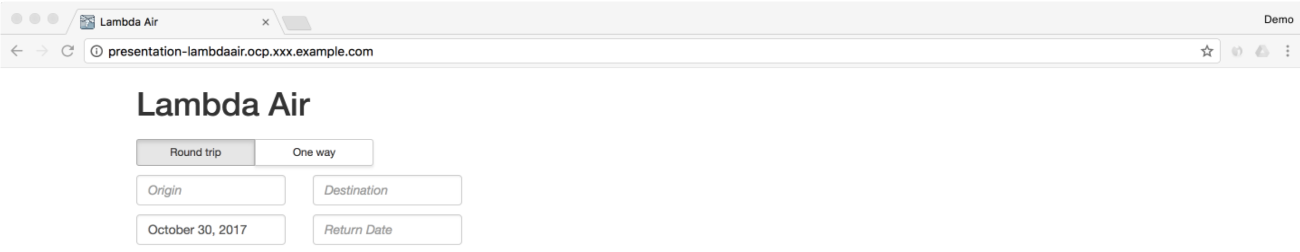
4.7. FLIGHT SEARCH

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
jaeger-query    jaeger-query-lambdaair.ocp.xxx.example.com
jaeger-query    <all>     edge/Allow    None
presentation    presentation-lambdaair.ocp.xxx.example.com
presentation    8080                    None
```

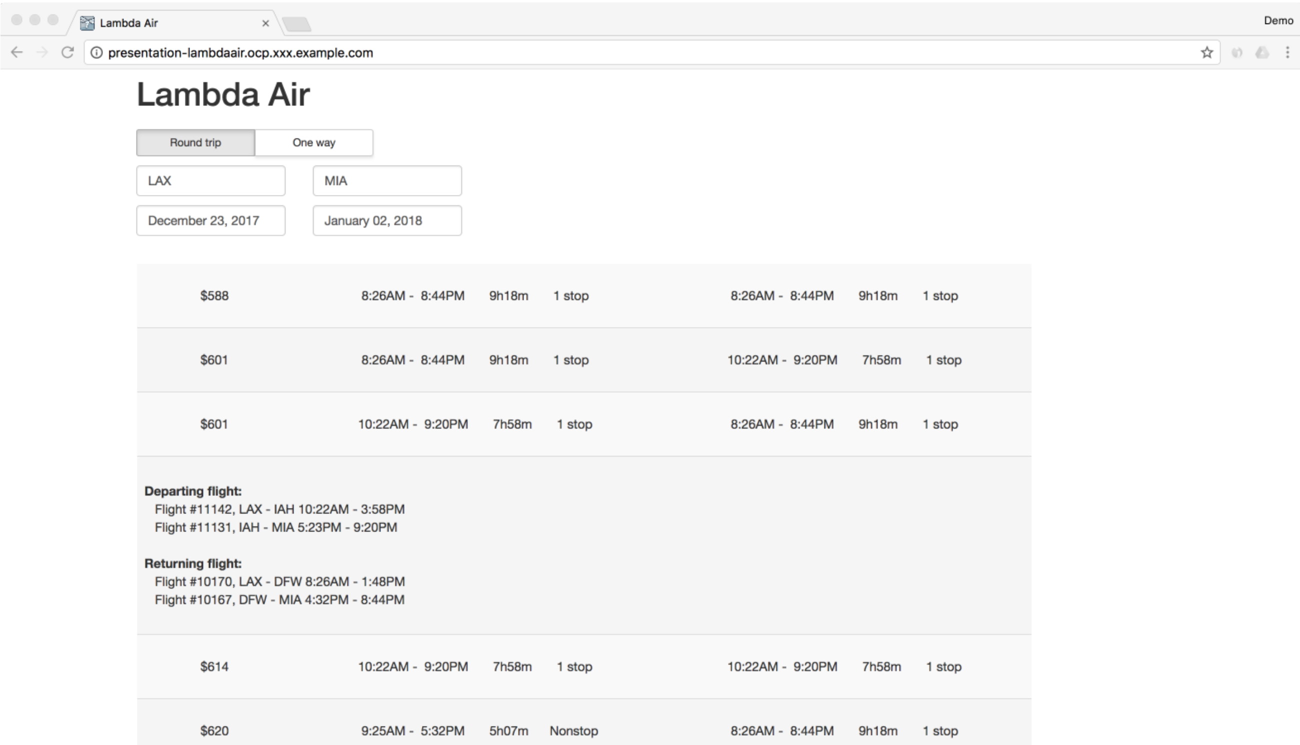
Use the URL of the route to access the HTML application from a browser, and verify that it comes up:

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 environment 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-dscwm | grep batch
... ...presentation.service.API_GatewayController    : Will price a batch
of 20 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 13 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 20 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 13 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
```

```
of 20 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 13 tickets
```

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

```
$ vi project-defaults.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=project-defaults.yml

configmap "presentation" created
```

Edit the *Presentation* deployment config and mount this *ConfigMap* as /deployments/config, where it will automatically be part of the WildFly Swarm 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/project-defaults.yml
          subPath: project-defaults.yml
      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-bn1gp                    1/1       Running     0          24m
airports-s2i-1-build                0/1       Completed   0          24m
cassandra-0                         1/1       Running     0          55m
cassandra-1                         1/1       Running     2          55m
cassandra-2                         1/1       Running     3          55m
edge-1-nlb4b                        1/1       Running     0          12m
edge-s2i-1-build                    0/1       Completed   0          13m
flights-1-n0lbx                     1/1       Running     0          11m
flights-s2i-1-build                 0/1       Completed   0          11m
jaeger-agent-1-g8s9t                1/1       Running     0          39m
jaeger-cassandra-schema-job-7d58m   0/1       Completed   0          55m
jaeger-collector-418097188-b090z    1/1       Running     4          55m
jaeger-query-751032167-vxr3w        1/1       Running     3          55m
presentation-1-dscwm                1/1       Running     0          1m
presentation-2-deploy               0/1       ContainerCreating   0
3s
presentation-s2i-1-build            0/1       Completed   0          1m
sales-1-g96zm                       1/1       Running     0          4m
sales-s2i-1-build                   0/1       Completed   0          5m
sales2-1-36hww                      1/1       Running     0          3m
sales2-s2i-1-build                  0/1       Completed   0          4m
```

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-2-36dt9       1/1       Running     0          2s
...
```

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-2-36dt9 | grep batch
... ...presentation.service.API_GatewayController    : Will price a batch
of 30 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 3 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 30 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 3 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 30 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
of 3 tickets
... ...presentation.service.API_GatewayController    : Will price a batch
```

```
of 30 tickets
... ...presentation.service.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 JavaScript file provided in the *Edge* project over to the shared storage for this service:

```
$ cp Edge/misc/routing.js /mnt/wildfly-swarm/edge/
```

Create a persistent volume for the *Edge* service. External JavaScript files placed in this location can provide dynamic routing.

```
$ oc create -f Edge/misc/edge-pv.yml
persistentvolume "edge" created
```

Also create a persistent volume claim:

```
$ oc create -f Edge/misc/edge-pvc.yml
persistentvolumeclaim "edge" created
```

Verify that the claim is bound to the persistent volume:

```
$ oc get pvc
NAME                    STATUS     VOLUME            CAPACITY    ACCESSMODES
STORAGECLASS AGE
cassandra-data-cassandra-0   Bound     cassandra-pv-1   1Gi         RWO
39m
cassandra-data-cassandra-1   Bound     cassandra-pv-2   1Gi         RWO
39m
cassandra-data-cassandra-2   Bound     cassandra-pv-3   1Gi         RWO
39m
cassandra-logs-cassandra-0   Bound     cassandra-pv-4   1Gi         RWO
39m
cassandra-logs-cassandra-1   Bound     cassandra-pv-5   1Gi         RWO
39m
cassandra-logs-cassandra-2   Bound     cassandra-pv-6   1Gi         RWO
39m
edge                    Bound      edge              1Gi         RWO
3s
```

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

```
$ oc volume dc/edge --add --name=edge --type=persistentVolumeClaim --
claim-name=edge --mount-path=/edge
deploymentconfig "edge" updated
```

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

Wait until the second version of the pod reaches the running state. Then return to the browser and perform one or more flight searches. After that, return to the OpenShift environment and look at the log for the edge pod.

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

```
$ oc logs edge-2-fzgg0
...
... INFO  [....impl.JavaScriptMapper] (default task-4) Rerouting to B
instance for IP Address 10.3.116.235
... INFO  [....impl.JavaScriptMapper] (default task-7) Rerouting to B
instance for IP Address 10.3.116.235
... INFO  [....impl.JavaScriptMapper] (default task-8) Rerouting to B
instance for IP Address 10.3.116.235
... INFO  [....impl.JavaScriptMapper] (default task-11) Rerouting to B
instance for IP Address 10.3.116.235
```

In this case, the logs from *sales2* will show tickets being priced with a modified algorithm:

```
$ oc logs sales2-1-36hww
... INFO  [...service.Controller] (default task-27) Priced ticket at 463
... INFO  [...service.Controller] (default task-27) Priced ticket at 425
... INFO  [...service.Controller] (default task-27) Priced ticket at 407
... INFO  [...service.Controller] (default task-27) Priced ticket at 549
... INFO  [...service.Controller] (default task-27) Priced ticket at 509
... INFO  [...service.Controller] (default task-27) Priced ticket at 598
... INFO  [...service.Controller] (default task-27) Priced ticket at 610
```

If that is not the case and your IP address ends in an even number, you will not see any logging at the *INFO* level by the JavaScript and need to turn up the verbosity to clearly see it be executed. 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.

```
$ cat /mnt/wildfly-swarm/edge/routing.js

if( mapper.getServiceName( request ) == "sales" )
{
 var ipAddress = mapper.getBaggageItem( request, "forwarded-for" );
 mapper.fine( 'Got IP Address as ' + ipAddress );
 if( ipAddress )
 {
  var lastDigit = ipAddress.substring( ipAddress.length - 1 );
  mapper.fine( 'Got last digit as ' + lastDigit );
  if( lastDigit % 2 == 0 )
  {
   mapper.info( 'Rerouting to B instance for IP Address ' + ipAddress );
   //Even IP address, reroute for A/B testing:
   hostAddress = mapper.getRoutedAddress( request, "http://sales2:8080" );
  }
 }
}
```

This is a simple matter of editing the file and deploying a new version of the *edge* service to pick up the updated script:

```
$ oc rollout latest edge
deploymentconfig "edge" 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 *sales2* 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. MAVEN PROJECT MODEL

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.

5.2.1. Supported Software Components

Software components used in this reference architecture application fall into three separate categories:

- Red Hat Supported Software
- Tested and Verified Software Components
- Community Open-Source Software

The use of Maven BOM files to declare library dependencies helps distinguish between these categories.

5.2.1.1. Red Hat Supported Software

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

</properties>
```

Further down in the POM file, the dependency section references a BOM file in the Red Hat repository that maintains a list of supported versions and libraries for WildFly Swarm:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>bom</artifactId>
            <version>${version.wildfly.swarm}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
...
```

This BOM file allows the project Maven files to reference WildFly fractions without providing a version, and import the supported library versions:

```
    <!-- WildFly Swarm Fractions -→
    <dependency>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>monitor</artifactId>
    </dependency>
    <dependency>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>jaxrs</artifactId>
    </dependency>
...
    <!-- CDI needed to inject system properties with @RequestScoped -→
    <dependency>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>cdi</artifactId>
    </dependency>
```

5.2.1.2. Tested and Verified Software Components

To use tested and verified components, a project Maven file would also reference the *bom-certified* file in its dependency management section:

```
<dependencyManagement>
...
    <dependencies>
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>bom-certified</artifactId>
            <version>${version.wildfly.swarm}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

This BOM file maintains a list of library versions that have been tested and verified, allowing their use in the project POM:

```
    <dependency>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>hystrix</artifactId>
    </dependency>
```

5.2.1.3. Community Open-Source Software

The reference architecture application also makes occasional use of open-source libraries that are neither supported nor tested and verified by Red Hat. In such cases, the POM files do not make use of dependency management, and directly import the required version of each library:

```
    <!-- Community fraction - Jaeger OpenTracing -→
    <dependency>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>jaeger</artifactId>
        <version>${version.wildfly.swarm.community}</version>
```

```
</dependency>
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-jaxrs2</artifactId>
  <version>0.0.9</version>
</dependency>
<!-- Hystrix plugin included in this library and works regardless of Feign -→
<dependency>
  <groupId>io.github.openfeign.opentracing</groupId>
  <artifactId>feign-hystrix-opentracing</artifactId>
  <version>0.0.5</version>
</dependency>
```

5.2.2. OpenShift Health Probes

Every service in this application also declares a dependency on the `WildFly Swarm Monitor` fraction, which provides access to the application runtime status on each node.

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>monitor</artifactId>
</dependency>
```

When a dependency on the *Monitor* is declared, *fabric8* generates default OpenShift health probes that communicate with Monitor 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. 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 for the Jaeger agent 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 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 the WildFly Swarm 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.

5.4. WILDFLY SWARM REST SERVICE

5.4.1. Overview

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

5.4.2. WildFly Swarm REST Service

To easily include the dependencies for a simple WildFly Swarm application that provides a REST service, declare the following artifact:

```
<dependency>
   <groupId>org.wildfly.swarm</groupId>
   <artifactId>jaxrs</artifactId>
</dependency>
```

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

```
...
```

```
import javax.ws.rs.Path;
...

@Path("/")
public class Controller
```

This is enough to create a JAX-RS service that listens on the default port of 8080 on the root context.

Each REST operation is implemented by a Java method. Business operations typically require specifying the HTTP verb, request and response media types, and request arguments:

```
@GET
@Path("/airports")
@Produces(MediaType.APPLICATION_JSON)
public Collection<Airport> airports(@QueryParam( "filter" ) String filter)
{
 ...
```

5.4.3. Startup Initialization

The *Airports* service uses eager initialization to load airport data into memory at the time of startup. This is implemented through a *ServletContextListener* that is called as the Servlet context is initialized and destroyed:

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class ApplicationInitialization implements ServletContextListener
{

 @Override
 public void contextInitialized(ServletContextEvent sce)
```

5.5. JAX-RS CLIENT AND LOAD BALANCING

5.5.1. 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 the *JAX-RS Client* and the generated OpenShift service for high availability.

5.5.2. JAX-RS Client

The JAX-RS Client is made available along side the JAX-RS service library and is therefore already imported for these projects.

To obtain a new instance of the JAX-RS Client, use the factory method provided by the *ClientBuilder* class. Convenience methods for working with the JAX-RS client are included in the *RestClient* class within each project:

```
private static Client getClient()
```

```
{
 Client client = ClientBuilder.newClient();
 ...
```

To make calls to a service using the JAX-RS Client, a *WebTarget* object must be obtained using the destination address. The convenience method provided for this purpose assumes that service addresses are externalized as system properties and retrievable through the service name:

```
public static WebTarget getWebTarget(String service, Object... path)
{
 Client client = getClient();
 WebTarget target = client.target( System.getProperty( "service." +
service + ".baseUrl" ) );
 for( Object part : path )
 {
  target = target.path( String.valueOf( part ) );
 }
 return target;
}
```

Given a *WebTarget*, a convenience method helps make the request, parse the response and return the right object type:

```
public static <T> T invokeGet(WebTarget webTarget, Class<T> responseType)
throws HttpErrorException, ProcessingException
{
 Response response = webTarget.request( MediaType.APPLICATION_JSON ).get();
 return respond( response, responseType );
}
```

Parsing the response is just one line when the request is successful, but it is important to also check the response code for errors, and react appropriately:

```
private static <T> T respond(Response response, Class<T> responseType)
throws HttpErrorException
{
 if( response.getStatus() >= 400 )
 {
  HttpErrorException exception = new HttpErrorException( response );
  logger.info( "Received an error response for the HTTP request: " +
exception.getMessage() );
  throw exception;
 }
 else if( responseType.isArray() )
 {
  return response.readEntity( new GenericType<>( responseType ) );
 }
 else
 {
  return response.readEntity( responseType );
 }
}
```

Using these convenience methods, services can be called in 1-2 easy lines, for example:

```
WebTarget webTarget = RestClient.getWebTarget( "airports", "airports" );
Airport[] airportArray = RestClient.invokeGet( webTarget, Airport[].class
);
```

The service address provided to the convenience method is resolved based on values provided in the configuration properties:

service:
  airports:
    baseUrl: http://edge:8080/airports

The service address is resolved, based on the service name, to a URL with the hostname of *edge* with port 8080. The Edge service uses the second part of the address, the root web context, to redirect the request through static or dynamic routing, as explained later in this document.

The provided hostname of *edge* 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.

To emphasize, it should be noted that all calls are being routed to an OpenShift internal load balancer, which is aware of replication and failure of service instances, and can redirect the request properly.

5.6. WILDFLY SWARM WEB APPLICATION

5.6.1. Overview

The Presentation service uses based WildFly Swarm WebApp capability to expose HTML and JavaScript and run a client-side application in the browser.

5.6.2. Context Disambiguation

To avoid a clash between the JAX-RS and Web Application listeners, the *Presentation* service declares a JAX-RS *Application* with a root web context of */gateway*. This allows the *index.html* to capture requests sent to the root context:

```
import javax.ws.rs.ApplicationPath;

@ApplicationPath( "/gateway" )
public class Application extends javax.ws.rs.core.Application
{
}
```

5.6.3. 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.

5.6.4. 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>
```

5.6.5. 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.

5.6.5.1. jQuery UI

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

5.6.5.2. 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.

5.7. HYSTRIX

5.7.1. Overview

The *Presentation* service also contains a REST service, 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.

5.7.2. 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 HttpErrorException, ProcessingException
  {
   WebTarget webTarget = getWebTarget( "sales", "price" );
   return invokePost( webTarget, flight, Itinerary.class );
  }

  @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 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
```

5.7.3. 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 configured thread pool size is injected into the API gateway service as a field:

```
@Inject
@ConfigurationValue( "hystrix.threadpool.SalesThreads.coreSize" )
private int threadSize;
```

To enable injection, the *API_GatewayController* class must be annotated as a bean:

```
@RequestScoped
public class API_GatewayController
```

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.

5.8. OPENSHIFT CONFIGMAP

5.8.1. 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.

5.8.2. 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.

5.9. JAEGER

5.9.1. Overview

This reference architecture uses Jaeger and the **`OpenTracing API`** to collect and broadcast tracing data to the Jaeger back end, which is deployed as an OpenShift service and backed by persistent Cassandra database images. The tracing data can be queried from the *Jaeger* console, which is exposed through an OpenShift route.

5.9.2. Cassandra Database

5.9.2.1. Persistence

The Cassandra database configured by the default *jaeger-production_template* is an ephemeral datastore with 3 replicas. This reference architecture adds persistence to Cassandra by configuring persistent volume.

5.9.2.2. Persistent Volume Claims

This reference architecture uses volumeClaimTemplates to dynamically create the required number of persistent volume claims:

```
volumeClaimTemplates:
- metadata:
    name: cassandra-data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
- metadata:
    name: cassandra-logs
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

These two volume claim templates generate a total of 6 persistent volume claims for the three Cassandra replicas.

5.9.2.3. Persistent Volume

In most cloud environments, corresponding persistent volumes would be available or dynamically provisioned. The reference architecture lab creates and mounts a logical volume that is exposed through **NFS**. In total, 6 OpenShift persistent volumes serve to expose the storage to the image. Once the storage is set up and shared by the NFS server:

```
$ oc create -f Jaeger/jaeger-pv.yml
persistentvolume "cassandra-pv-1" created
persistentvolume "cassandra-pv-2" created
persistentvolume "cassandra-pv-3" created
persistentvolume "cassandra-pv-4" created
persistentvolume "cassandra-pv-5" created
persistentvolume "cassandra-pv-6" created
```

5.9.3. Jaeger Image

Other than configuring persitence, this reference architecture uses the Jaeger production template of the jaeger-openshift project as provided in the latest release, at the time of writing. This results in the use of version 0.6 images from jaeger-tracing:

```
- description: The Jaeger image version to use
  displayName: Image version
  name: IMAGE_VERSION
  required: false
  value: "0.6"
```

5.9.4. Jaeger Tracing Client

While the *Jaeger* service allows distributed tracing data to be aggregated, persisted and used for reporting, this application also relies on the client-side Java implementation of the *OpenTracing API* by *Jaeger* to correlate calls and send data to the server.

Integration with JAX-RS and other framework libraries make it very easy to use Jaeger in the application. Include the libraries by declaring a dependency in the project Maven file:

```
<!-- Jaeger OpenTracing -→ <dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>jaeger</artifactId>
</dependency>
```

To collect distributed tracing data for calls made from the JAX-RS Client, include the *opentracing-jaxrs2* library:

```
<dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-jaxrs2</artifactId>
    <version>0.0.9</version>
</dependency>
```

Use the JAX-RS Feature provided in this library to intercept outgoing calls from JAX-RS Client. This application registers the *Feature* in its convenience method that abstracts away the JAX-RS Client configuration:

```
client.register( ClientTracingFeature.class );
```

Also specify in the application properties, the percentage of requests that should be traced, as well as the connection information for the Jaeger server. Once again, we rely on the OpenShift service abstract to reach the Jaeger service, and jaeger-agent is the OpenShift service name:

JAEGER_AGENT_HOST: jaeger-agent
JAEGER_AGENT_PORT: 6831
JAEGER_SERVICE_NAME: presentation
JAEGER_REPORTER_LOG_SPANS: true
JAEGER_SAMPLER_TYPE: const
JAEGER_SAMPLER_PARAM: 1

The sampler type is set to const, indicating that the Constant Sampler should be used. The sampling rate of 1, meaning 100%, is therefore already implied but is a required configuration that should be left in. The Jaeger service name affects how tracing data from this service is reported by the Jaeger console, and the agent host and port is used to reach the Jaeger agent through UDP.

These steps are enough to collect tracing data, but a *Tracer* object can also be [retrieved] by the code for extended functionality, by calling *GlobalTracer.get()*. 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, by treating the span as a Java resource, to collect more meaningful tracing data.

5.9.4.1. Baggage Data

While the *Jaeger* client library 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 *uberctx-forwarded-for*. It is used by the *Edge* service in JavaScript to perform dynamic routing.

5.10. EDGE SERVICE

5.10.1. Overview

This reference architecture uses a reverse proxy for all calls between microservices. The reverse proxy is a custom implementation of an edge service with support for both declarative static routing, as well as dynamic routing through simple JavaScript, or other pluggable implementations.

5.10.2. Usage

By default, the *Edge* service uses static declarative routing as defined in its application properties:

```
edge:
  proxy:
    presentation:
      address: http://presentation:8080
    airports:
      address: http://airports:8080
    flights:
      address: http://flights:8080
    sales:
      address: http://sales:8080
```

It should be noted that declarative mapping is considered static, only because routing is based on the service address without regard for the content of the request or other contextual information. It is still possible to override the mapping properties through an OpenShift ConfigMap, as outlined in the section on Hystrix properties, to change the mapping rules for a given web context.

Dynamic routing through JavaScript is a simple matter of reassigning the implicit hostAddress variable:

```
if( mapper.getServiceName( request ) == "sales" )
{
 var ipAddress = mapper.getBaggageItem( request, "forwarded-for" );
 mapper.fine( 'Got IP Address as ' + ipAddress );
 if( ipAddress )
 {
  var lastDigit = ipAddress.substring( ipAddress.length - 1 );
  mapper.fine( 'Got last digit as ' + lastDigit );
  if( lastDigit % 2 == 0 )
  {
   mapper.info( 'Rerouting to B instance for IP Address ' + ipAddress );
   //Even IP address, reroute for A/B testing:
   hostAddress = mapper.getRoutedAddress( request, "http://sales2:8080" );
```

```
    }
  }
}
```

Mapping through the application properties happens first, and if a script does not modify the value of *hostAddress*, the original mapping remains effective.

5.10.3. Implementation Details

This edge service extends Smiley's HTTP Proxy Servlet, an open-source reverse proxy implementation using Java Servlet and *Apache HttpClient*. This library provides pluggable dynamic routing and has been tested and used in the community.

To use this component, the Edge service extends the provided proxy Servlet:

```
@WebServlet( name = "Edge", urlPatterns = "/*" )
public class EdgeService extends ProxyServlet
```

The implementation does not require any initialization, so an empty method is provided:

```
 @Override
 protected void initTarget() throws ServletException
 {
  //No target URI used
 }
```

The main logic for the Servlet class is provided in the *service* method. The implementation for this proxy uses its own routing rules to set the destination host and address as the ATTR_TARGET_HOST and ATTR_TARGET_URI request attributes, respectively. The *mapping* object is used to obtain the destination based on the request:

```
@Override
protected void service(HttpServletRequest servletRequest,
HttpServletResponse servletResponse) throws ServletException, IOException
{
 try
 {
  String fullAddress = mapping.getHostAddress( servletRequest );
  URI uri = new URI( fullAddress );
  logger.fine( "Will forward request to " + fullAddress );
  servletRequest.setAttribute( ATTR_TARGET_HOST, URIUtils.extractHost( uri
) );
  servletRequest.setAttribute( ATTR_FULL_URI, uri.toString() );
  URI noQueryURI = new URI( uri.getScheme(), uri.getUserInfo(),
    uri.getHost(), uri.getPort(), uri.getPath(), null, null );
  servletRequest.setAttribute( ATTR_TARGET_URI, noQueryURI.toString() );
  super.service( servletRequest, servletResponse );
 }
 catch( URISyntaxException e )
 {
  throw new ServletException( e );
 }
}
```

The *ProxyServlet* also relies on another method to find the routing address based on the request. The *Edge* implementation maps the route in a single step, so the result above is stored as a request attribute and can be returned from the second method:

```
@Override
protected String rewriteUrlFromRequest(HttpServletRequest servletRequest)
{
 return (String)servletRequest.getAttribute( ATTR_FULL_URI );
}
```

Mapping is provided by a separate framework as part of the same service. The integration with this Servlet is simple and performed through the injection of the *MappingConfiguration* object:

```
@Inject
private MappingConfiguration mapping;
```

MappingConfiguration retains a chain of mappers it uses for routing:

```
@ApplicationScoped
public class MappingConfiguration
{
 private static Logger logger = Logger.getLogger(
MappingConfiguration.class.getName() );
 private List<Mapper> mapperChain = new ArrayList<>();

 public MappingConfiguration()
 {
  Mapper[] candidates = new Mapper[]{PropertyMapper.getInstance(),
    JavaScriptMapper.getInstance()};
  for( Mapper candidate : candidates )
  {
   if( candidate.initialize() )
   {
    mapperChain.add( candidate );
   }
  }
 }

 public String getHostAddress(HttpServletRequest request)
 {
  if( mapperChain.isEmpty() )
  {
   logger.severe( "No mapper configured, will return null" );
   return null;
  }
  else
  {
   Iterator<Mapper> mapperIterator = mapperChain.iterator();
   String hostAddress = mapperIterator.next().getHostAddress( request,
null );
   logger.fine( "Default mapper returned " + hostAddress );
   while( mapperIterator.hasNext() )
   {
    Mapper mapper = mapperIterator.next();
    hostAddress = mapper.getHostAddress( request, hostAddress );
```

```
        logger.fine( "Mapper " + mapper + " returned " + hostAddress );
      }
      return hostAddress;
    }
  }
}
```

The default implementation uses a PropertyMapper and a JavaScriptMapper. The property mapper looks up the root web context in system properties, typically defined in a *project-defaults.yml* file, and uses the value as the destination address.

The JavaScript mapper looks for */edge/routing.js* on the file system and if found, executes the script. The HTTP request is injected into the JavaScript context as the *request* variable and the destination address returned by the previous mapper in the chain is injected as *hostAddress*. The mapper object itself is also inject as *mapper* to provide convenience methods to look up Jaeger baggage items and construct a URL with a new host address. After execution, the modified value for *hostAddress* is read from the context and used.

5.10.4. A/B Testing

To implement A/B testing, the Sales2 service introduces a minor change in the algorithm for calculating fares. Dynamic routing is provided by *Edge* through JavaScript.

Only calls to the *Sales* service are potentially filtered:

```
if( mapper.getServiceName( request ) == "sales" )
{
 ...
}
```

From those calls, requests that originate from an IP address ending in an even digit are filtered, by modified the value of *hostAddress*:

```
 var ipAddress = mapper.getBaggageItem( request, "forwarded-for" );
 mapper.fine( 'Got IP Address as ' + ipAddress );
 if( ipAddress )
 {
  var lastDigit = ipAddress.substring( ipAddress.length - 1 );
  mapper.fine( 'Got last digit as ' + lastDigit );
  if( lastDigit % 2 == 0 )
  {
   mapper.info( 'Rerouting to B instance for IP Address ' + ipAddress );
   //Even IP address, reroute for A/B testing:
   hostAddress = mapper.getRoutedAddress( request, "http://sales2:8080" );
  }
 }
```

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 JavaScript in place, the OpenShift deployment config can be adjusted administratively to mount a directory as a volume:

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

This results in a lookup for a *routing.js* file under the edge directory. If found, its content is executed with the default JavaScript Engine of the JDK and any changes to the host address value is returned:

```
FileReader fileReader = new FileReader( JS_FILE_NAME );

Bindings bindings = engine.createBindings();
bindings.put( "request", request );
bindings.put( "hostAddress", hostAddress );

engine.eval( fileReader, bindings );

return (String)bindings.get( "hostAddress" );
```

CHAPTER 6. CONCLUSION

WildFly Swarm provides a platform that is well-suited to the development of microservices, while taking advantage of mature Java EE patterns and implementations. In addition to the functionality of popular and widely used Java EE specifications, WildFly Swarm also integrates with a large number of modern libraries designed to solve common challenges in microservice development. WildFly Swarm fractions provide a powerful and easy integration point for new and additional features. Services built on this platform are then easily deployed on **`Red Hat® OpenShift Container Platform 3`**.

This paper and its accompanying technical implementation seek to serve as a useful reference for building an application in the microservice architectural style, 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	Nov 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
Ken Finnigan	Principal Software Engineer	Technical Content Review, WildFly Swarm
Gary Brown	Principal Software Engineer	Jaeger Tracing
Pavol Loffay	Software Engineer	Jaeger Tracing
Martin Kouba	Senior Software Engineer	Project Build Review

APPENDIX C. REVISION HISTORY

Revision 1.0-0	**November 2017**	**BM**