



Reference Architectures 2017

Building Microservices on OpenShift Container Platform with Fuse Integration Services

Jeremy Ary

Reference Architectures 2017 Building Microservices on OpenShift Container Platform with Fuse Integration Services

Jeremy Ary
jary@redhat.com

Legal Notice

Copyright © 2017 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This reference architecture continues to build on previous work designing, developing and deploying microservices hosted by JBoss EAP 7 to OpenShift Container Platform. The architecture further extends the functionality of the previous project by utilizing the Red Hat JBoss Fuse Integration Services 2.0 xPaaS image to incorporate an API Gateway, thereby presenting a single API to upstream clients while proxying and routing requests to and from a mixture of event-based and API-based downstream microservices. The API Gateway service and event-based portion of microservices utilize the Spring Boot and Camel archetypes available via Fuse Integration Services 2.0, alongside the OpenShift Container Platform Messaging service backed by Red Hat JBoss AMQ, to showcase orchestration of message-oriented middleware components via a variety of Enterprise Integration Patterns. Lastly, monitoring of the various microservices and event-based coordination is demonstrated on a per-service level, via FIS 2.0's built-in HawtIO integration, and on an all-encompassing aggregated-container basis, via the EFK (Elasticsearch, Fluentd, and Kibana) Logging Stack.

Table of Contents

| | |
|--|-----------|
| COMMENTS AND FEEDBACK | 3 |
| CHAPTER 1. EXECUTIVE SUMMARY | 4 |
| CHAPTER 2. REFERENCE ARCHITECTURE ENVIRONMENT | 5 |
| 2.1. PROJECT MODULES | 5 |
| CHAPTER 3. CREATING THE ENVIRONMENT | 7 |
| 3.1. OVERVIEW | 7 |
| 3.2. BUILD AND DEPLOY | 7 |
| 3.2.1. Creating a New Project | 7 |
| 3.3. MYSQL IMAGES | 7 |
| 3.4. JBOSS EAP 7 XPAAS IMAGES | 9 |
| 3.5. FUSE INTEGRATION SERVICES 2.0 XPAAS IMAGES | 9 |
| 3.6. BUILDING THE SERVICES | 9 |
| 3.6.1. A-MQ Messaging Broker | 10 |
| 3.6.2. FIS 2.0 Services | 11 |
| 3.6.3. EAP Services | 12 |
| 3.7. POPULATING DATABASES | 13 |
| 3.8. REPLICAS | 14 |
| 3.9. AGGREGATED LOGGING | 15 |
| CHAPTER 4. DESIGN AND DEVELOPMENT | 18 |
| 4.1. OVERVIEW | 18 |
| 4.2. APPLICATION STRUCTURE | 18 |
| 4.3. PRESENTATION MODULE | 18 |
| 4.4. CUSTOMIZING RED HAT JBOSS EAP FOR PERSISTENCE | 19 |
| 4.5. GATEWAY SERVICE | 20 |
| 4.6. BILLING & WAREHOUSE SERVICES | 21 |
| 4.7. APPLICATION MONITORING | 23 |
| CHAPTER 5. CONCLUSION | 27 |
| APPENDIX A. AUTHORSHIP HISTORY | 28 |
| APPENDIX B. CONTRIBUTORS | 29 |
| APPENDIX C. REVISION HISTORY | 30 |

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 continues to build on previous work designing, developing and deploying microservices hosted by JBoss EAP 7 to OpenShift Container Platform. The architecture further extends the functionality of the previous project by utilizing the Red Hat JBoss Fuse Integration Services 2.0 xPaaS image to incorporate a microservices API Gateway pattern implementation, thereby presenting a single API to upstream clients while proxying and routing requests to and from a mixture of event-based and API-based downstream microservices. The gateway implementation and event-based portion of microservices utilize the Spring Boot and Camel archetypes available via Fuse Integration Services 2.0, alongside the OpenShift Container Platform Messaging service, backed by Red Hat JBoss A-MQ, to showcase orchestration of message-oriented middleware components via a variety of Enterprise Integration Patterns. Lastly, monitoring of the various microservices and event-based coordination is demonstrated on a per-service level, via FIS 2.0's built-in HawtIO integration, as well as on an all-encompassing aggregated-container basis via the EFK (Elasticsearch, Fluentd, and Kibana) Logging Stack.

The audience for this paper includes enterprise integration architects and developers engaged in either Greenfield or Brownfield projects with the goal of creating a microservice architecture and/or using Messaging, Logging, Fuse Integration Services, and JBoss EAP 7 on OpenShift. While working knowledge of Java EE, JBoss EAP, and Enterprise Integration Patterns is a prerequisite, as long as the reader is provided with a working OCP 3.5 environment, no OpenShift experience is assumed. The paper proceeds to describe, step by step, the process of building and deploying an application that runs on OCP 3.5, leveraging supported xPaaS and database images.

CHAPTER 2. REFERENCE ARCHITECTURE ENVIRONMENT

2.1. PROJECT MODULES

This reference architecture takes advantage of the following provided middleware xPaaS container images and templates:

- ✳ [Red Hat JBoss Fuse Integration Services 2.0 for OpenShift Container Platform](#)
- ✳ [Red Hat JBoss Enterprise Application Platform 7 for OpenShift](#)
- ✳ [Red Hat JBoss A-MQ for OpenShift](#)
- ✳ [EFK Stack Template for Aggregated Container Logs](#)
- ✳ [MySQL Container Image for OpenShift](#)

The architecture designs, develops and deploys a simple distributed microservice architecture on EAP 7 and FIS 2.0, on top of OCP 3.5. The project builds upon a basis design laid out and explained in depth in the reference architecture document on [EAP Microservice Architecture](#), enriching functionality with a microservices Gateway API pattern implementation, event-driven microservices, and monitoring components.

The OCP 3.5 environment used herein includes a single stand-alone master configuration with two deployment nodes, which is not intended for use in a production environment. Further information on prerequisite installation and configuration of the Red Hat Enterprise Linux Operating System, OpenShift Container Platform, and more can be found in Section 3, *Creating the Environment*, of a previous Reference Architecture, [Building JBoss EAP 7 Microservices on OpenShift Container Platform](#). The example application will set up two OpenShift projects, *fis2-msa* and *logging*, to be deployed to the two schedulable nodes of the cluster.

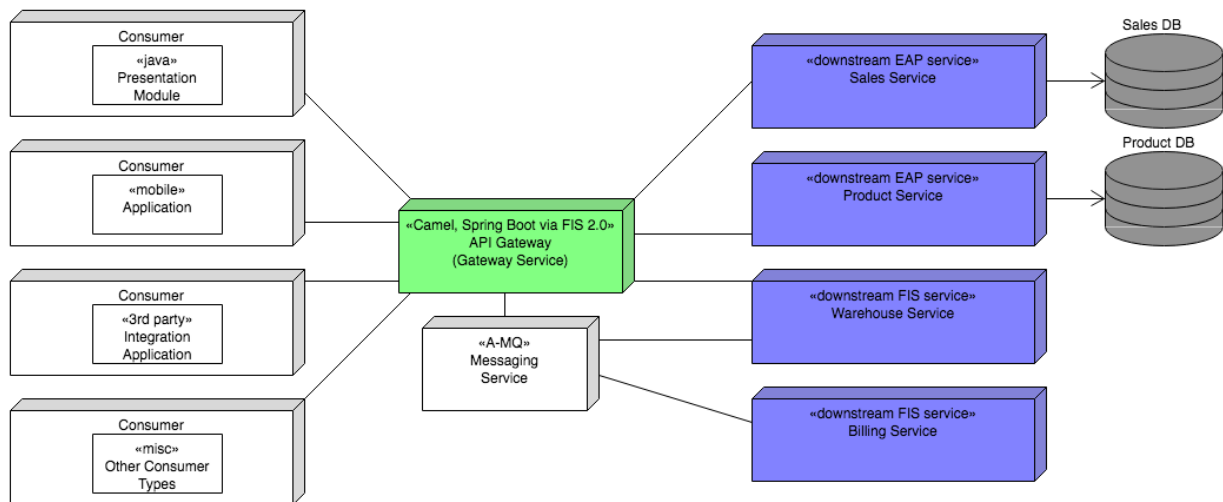
The primary OpenShift project, *fis2-msa*, includes nine services, collectively representing an eCommerce service scenario:

- ✳ *A-MQ (Messaging)*: based on A-MQ - Elastic messaging broker for gateway and event-driven modules
- ✳ *Billing Service*: based on FIS 2.0 - Event-driven module simulating payment processing and validation
- ✳ *Gateway Service*: based on FIS 2.0 - Microservices gateway API module serving to route and proxy API requests from upstream consumers to downstream services
- ✳ *Presentation*: based on EAP 7 - Representation of a possible upstream consumer; simple eCommerce website
- ✳ *Product DB*: based on MySQL - Product information database
- ✳ *Product Service*: based on EAP 7 - RESTful service providing logic regarding product information
- ✳ *Sales DB*: based on MySQL - Transaction and customer information database
- ✳ *Sales Service*: based on EAP 7 - RESTful service providing logic regarding customer and order information
- ✳ *Warehouse Service*: based on FIS 2.0 - Event-driven module simulating warehouse shipping fulfillment of successful transactions

A Secondary project, based on the EFK Logging Stack template, uses the following 3 components to provide unified log access to multiple nodes' containers via a single interface:

- ✎ *Fluentd*: cross-platform data collection providing analyzation of event and application logs
- ✎ *Elasticsearch*: Lucene-based search engine providing full-text search capabilities across the aggregated log collection
- ✎ *Kibana*: data visualization plugin for Elasticsearch

Figure 2.1. Design Diagram



The *Presentation* service represents a possible consumer for the application, sending requests to the microservices API Gateway and serving HTML to the client browser over HTTP. This web tier is stateless, and can therefore be replicated without concern for loss of state and EAP clustering setup. The provisioning of a gateway within the project yields the ability to serve various types of upstream consumers such as mobile, automation, and third-party integration.

Image-based deployments rely on the [Source to Image \(S2I\)](#) functionality to build and deploy.

All OpenShift services can be configured and scaled with the desired number of replicas. In a default OpenShift installation, the nodes are the only targets for service deployment. However, the masters can also be configured as schedulable in order to host service replicas.

CHAPTER 3. CREATING THE ENVIRONMENT

3.1. OVERVIEW

This reference architecture can be deployed on either a production or trial environment. In both cases, it is assumed that *middleware-master* refers to one (or only) OpenShift master host and that the environment includes two OpenShift node hosts with names *middleware-node1* and *middleware-node2*. It is further assumed that OpenShift has been installed by the *root* user and that a regular user has been created with basic access to the host machine, as well as access to OpenShift through its *identity providers*. A user with the *cluster-admin* role binding will also be required for building and utilizing the aggregated logging interface.

More information on prerequisite installation and configuration of the Red Hat Enterprise Linux Operating System, OpenShift Container Platform, and more can be found in Section 3: *Creating the Environment* of a previous Reference Architecture, [Building JBoss EAP 7 Microservices on OpenShift Container Platform](#).

3.2. BUILD AND DEPLOY

3.2.1. Creating a New Project

Utilize remote or direct terminal access to log in to the OpenShift environment as the user who will have ownership of the new project:

```
# oc login -u ocuser
```

Create the new project, specifying the name, display name (that which will be seen in the OCP web console), and a meaningful description:

```
# oc new-project fis2-msa --display-name="FIS2 Microservice
Architecture" --description="Event-driven and RESTful microservices via
microservices API Gateway pattern on OpenShift Container Platform"
```

3.3. MYSQL IMAGES

This reference architecture uses version 5.6 of the supported [MySQL image](#) to build two services providing persistence to the *product* and *sales* services.

To deploy the database services, use the *new-app* command and provide a number of required and optional environment variables along with the desired service name.

To deploy the product database service:

```
$ oc new-app -e MYSQL_USER=product -e MYSQL_PASSWORD=password -e
MYSQL_DATABASE=product -e MYSQL_ROOT_PASSWORD=passwd mysql --
name=product-db
```

To deploy the sales database service:

```
$ oc new-app -e MYSQL_USER=sales -e MYSQL_PASSWORD=password -e
MYSQL_DATABASE=sales -e MYSQL_ROOT_PASSWORD=passwd mysql --name=sales-
db
```

Warning

Database images created with this simple command are ephemeral and result in data loss in the case of a pod restart. Run the image with mounted volumes to enable persistent storage for the database. The data directory where MySQL stores database files is located at `/var/lib/mysql/data`.



Note

Refer to OpenShift Container Platform 3 documentation to configure [persistent storage](#).

Warning

Enabling clustering for database images is currently in [Technology Preview](#) and not intended for production use.

These commands create two OpenShift services, each running **MySQL** in its own container. In each case, a MySQL user is created with the value specified by the `MYSQL_USER` attribute and the associated password. The `MYSQL_DATABASE` attribute results in a database being created and set as the default user database.

To monitor the provisioning of the services, use `oc status`. You can use `oc get events` for further information and troubleshooting.

```
$ oc status
In project FIS2 Microservice Architecture (fis2-msa) on server
https://middleware-master.hostname.example.com:8443

svc/product-db - 172.30.192.152:3306
  dc/product-db deploys openshift/mysql:5.6
    deployment #1 deployed 2 minutes ago - 1 pod
svc/sales-db - 172.30.216.251:3306
  dc/sales-db deploys openshift/mysql:5.6
    deployment #1 deployed 37 seconds ago - 1 pod
2 warnings identified, use 'oc status -v' to see details.
```

The warnings can be further inspected by using the `-v` flag. In this case, they simply refer to the fact that the image has no readiness probe to verify pods are ready to accept traffic or ensure deployment is successful.

Make sure the database services are successfully deployed before moving on to building other services that depend on them. The service log indicates when the database has been successfully deployed. Use `tab` to complete the pod name, for example:

```
$ oc logs product-db-1-3drkp
---> 21:21:50      Processing MySQL configuration files ...
---> 21:21:50      Initializing database ...
---> 21:21:50      Running mysql_install_db ...
```

```
...omitted...
2016-06-04 21:21:58 1 [Note] /opt/rh/rh-
mysql56/root/usr/libexec/mysqld: ready for connections.
Version: '5.6.30'  socket: '/var/lib/mysql/mysql.sock'  port: 3306
MySQL Community Server (GPL)
```

3.4. JBOSS EAP 7 XPAAS IMAGES

The *Sales*, *Product* and *Presentation* services rely on OpenShift S2I for Java EE applications and use the [Red Hat xPaaS EAP Image](#). You can verify the presence of this image stream in the *openshift* project (utilizing a user with access to the *openshift* namespace):

```
# oc get imagestreams -n openshift
NAME          DOCKER REPO
...omitted...
jboss-eap70-openshift
registry.access.redhat.com/jboss-eap-7/eap70-openshift
1.4-30,1.4-28,1.4-26 + 2 more...  2 weeks ago
...omitted...
```

3.5. FUSE INTEGRATION SERVICES 2.0 XPAAS IMAGES

The *Gateway*, *Billing* and *Warehouse* services rely on the [Red Hat JBoss Fuse Integration Services 2.0](#) image. As before, you can verify the presence of this image stream by inspecting the *openshift* namespace:

```
# oc get imagestreams -n openshift
NAME          DOCKER REPO
...
fis-java-openshift  registry.access.redhat.com/jboss-fuse-6/fis-java-
openshift  2.0,2.0-3,1.0 + 2 more...  2 weeks ago
...
```

3.6. BUILDING THE SERVICES

The microservice application source code for this reference architecture is made available in a public git repository at <https://github.com/RHsyseng/FIS2-MSA>. This includes six distinct services, provided as subdirectories within the repository: *Gateway*, *Billing*, *Product*, *Sales*, *Presentation* and *Warehouse*. Templates which configure and deploy each of these modules have been included in a seventh directory, *yaml-templates*. In order to proceed, a copy of each template will be needed locally on the master node. Either use *curl* to fetch all the files, as seen below, or *curl* then execute the *fetch-templates.sh* script which will establish and populate a local copy of the *yaml-templates* directory.

```
$ mkdir yaml-templates
$ cd yaml-templates
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-
MSA/master/yaml-templates/logging-deployer.yaml
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-
MSA/master/yaml-templates/billing-template.yaml
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-
MSA/master/yaml-templates/gateway-template.yaml
```

```
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-MSA/master/yaml-templates/messaging-template.yaml
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-MSA/master/yaml-templates/presentation-template.yaml
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-MSA/master/yaml-templates/product-template.yaml
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-MSA/master/yaml-templates/sales-template.yaml
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-MSA/master/yaml-templates/warehouse-template.yaml
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-MSA/master/yaml-templates/logging-accounts.sh
$ chmod +x logging-accounts.sh
```

Alternatively, using the *fetch-templates.sh* script:

```
$ cd ~
$ curl -O -L https://raw.githubusercontent.com/RHsyseng/FIS2-MSA/master/fetch-templates.sh
$ chmod +x fetch-templates.sh
$ ./fetch-templates.sh

$ ls -la
total 52
drwxrwxr-x.  2 jary jary  272 May 23 01:18 .
drwx----- 10 jary jary 4096 May 23 01:18 ..
-rw-rw-r--.  1 jary jary 2855 May 23 01:18 billing-template.yaml
-rw-rw-r--.  1 jary jary 2855 May 23 01:18 gateway-template.yaml
-rwxrwxr-x.  1 jary jary 2251 May 23 01:18 logging-accounts.sh
-rw-rw-r--.  1 jary jary 9168 May 23 01:18 logging-deployer.yaml
-rw-rw-r--.  1 jary jary 5118 May 23 01:18 messaging-template.yaml
-rw-rw-r--.  1 jary jary 3097 May 23 01:18 presentation-template.yaml
-rw-rw-r--.  1 jary jary 2889 May 23 01:18 product-template.yaml
-rw-rw-r--.  1 jary jary 2847 May 23 01:18 sales-template.yaml
-rw-rw-r--.  1 jary jary 2897 May 23 01:18 warehouse-template.yaml
```

3.6.1. A-MQ Messaging Broker

The first component of the application to be deployed is the Messaging component, which is based on the *amq62-basic* imageStream. The *AMQ_USER* and *AMQ_PASSWORD* environment variables are available for overriding on the *messaging*, *gateway*, *billing*, and *warehouse* templates. Default values may be used by omitting the two *-p* arguments from the *oc process* command. If values are overridden, the remaining 3 modules should be provided with the same credentials via the *process* command as shown below.

```
$ oc process -f messaging-template.yaml -p AMQ_USER=mquser -p
AMQ_PASSWORD=password | oc create -f -
serviceaccount "mqserviceaccount" created
rolebinding "mqserviceaccount-view-role" created
deploymentconfig "broker-amq" created
service "broker-amq-mqtt" created
service "broker-amq-stomp" created
service "broker-amq-tcp" created
service "broker-amq-amqp" created
```

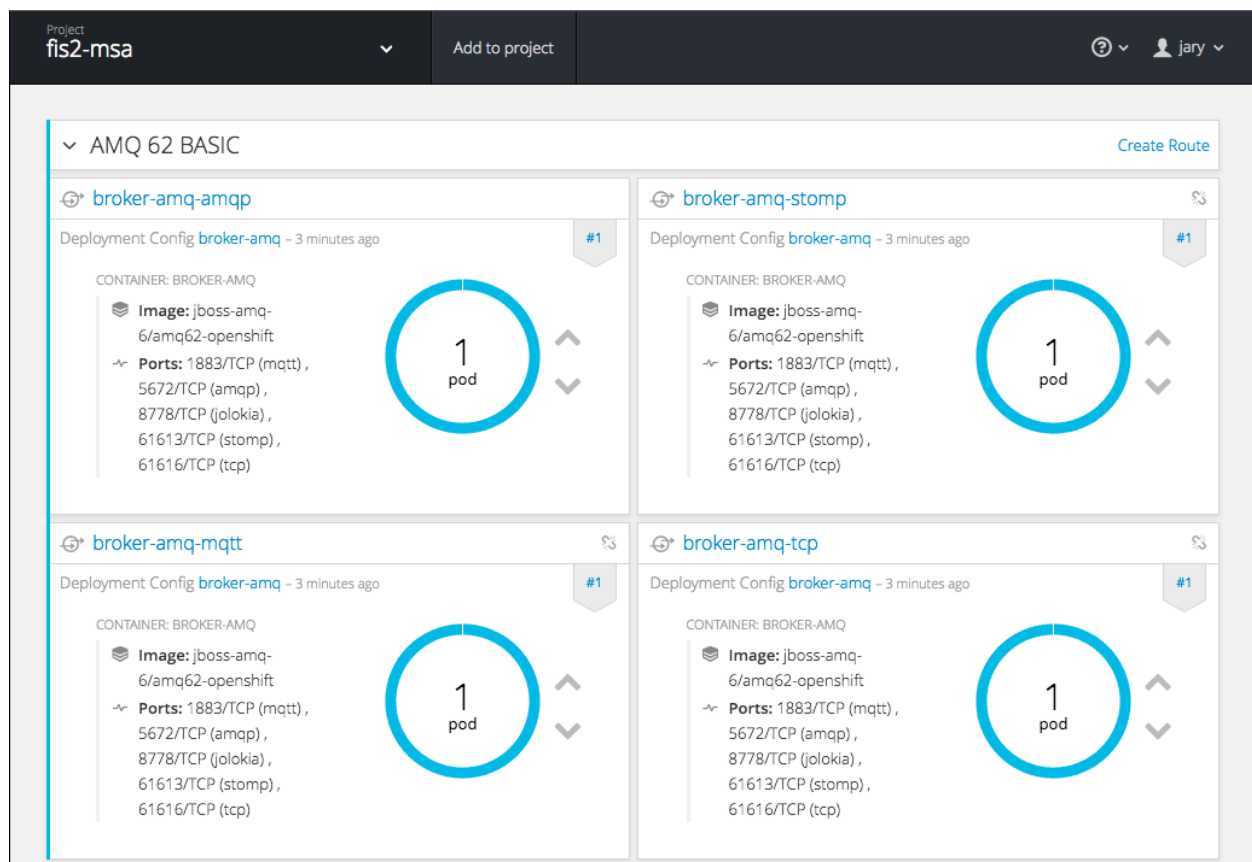
This template will create a service account required by A-MQ, assign the new account the 'view' role, then initiate 4 services, all related to different protocols available to A-MQ. The service configurations within the template link each A-MQ protocol deployment together as dependencies so that they display and scale as a unified service:

```

annotations:
  description: The broker's AMQP port.
  openshift.io/generated-by: OpenShiftNewApp
  service.alpha.openshift.io/dependencies: '[{"name":"broker-amq-
stomp","namespace":"","kind":"Service"}, {"name":"broker-amq-
mqtt","namespace":"","kind":"Service"}, {"name":"broker-amq-
tcp","namespace":"","kind":"Service"}]'

```

Figure 3.1. Unified A-MQ Deployments in Web console



3.6.2. FIS 2.0 Services

Once the A-MQ deployments are complete, initialize the first module to utilize the Fuse Integration Services 2.0 image, the Billing Service, via template:

```

$ oc process -f billing-template.yaml | oc create -f -
buildconfig "billing-service" created
imagestream "billing-service" created
deploymentconfig "billing-service" created
service "billing-service" create

```

THE `oc status` command can be used to monitor the progress of the operation. To monitor the build and deployment process more closely, find the running build and follow the build log:

```
$ oc get builds
```

| NAME | TYPE | FROM | STATUS | STARTED |
|-------------------|--------|------|---------|---------------|
| DURATION | | | | |
| billing-service-1 | Source | Git | Running | 1 seconds ago |
| 1s | | | | |

```
$ oc logs -f bc/billing-service-1
```

Next, initialize the Warehouse Service:

```
$ oc process -f warehouse-template.yaml | oc create -f -
buildconfig "warehouse-service" created
imagestream "warehouse-service" created
deploymentconfig "warehouse-service" created
service "warehouse-service" created
```

Finally, initialize the Gateway Service:

```
$ oc process -f gateway-template.yaml | oc create -f -
buildconfig "gateway-service" created
imagestream "gateway-service" created
deploymentconfig "gateway-service" created
service "gateway-service" created
```

3.6.3. EAP Services

Begin by deploying the *Product* and *Sales* services, bearing in mind that both have a persistence dependency on the aforementioned database services. If credentials were altered from those shown when creating the database services earlier, override the template defaults by providing the `MYSQL_USER` and `MYSQL_PASSWORD` environment variables.

To deploy the *Product* service:

```
$ oc process -f product-template.yaml | oc create -f -
buildconfig "product-service" created
imagestream "product-service" created
deploymentconfig "product-service" created
service "product-service" created
```

To deploy the *Sales* service:

```
$ oc process -f sales-template.yaml | oc create -f -
buildconfig "sales-service" created
imagestream "sales-service" created
deploymentconfig "sales-service" created
service "sales-service" created
```

Once again, `oc status` can be used to monitor progress. To monitor the build and deployment process more closely, find the running build and follow the build log:

```
$ oc get builds
```

| NAME | TYPE | FROM | STATUS | STARTED |
|-----------------|--------|------|---------|---------------|
| DURATION | | | | |
| sales-service-1 | Source | Git | Running | 1 seconds ago |


```
1s
$ oc logs -f bc/sales-service-1
```

Finally, deploy the *Presentation* service, which exposes a web tier that will interact with the microservices API Gateway. Specify the *ROUTE_URL* parameter required for route definition as an environment variable:

```
$ oc process -f presentation-template.yaml -p
ROUTE_URL=presentation.bxms.ose | oc create -f -
buildconfig "presentation" created
imagestream "presentation" created
deploymentconfig "presentation" created
service "presentation" created
route "presentation" created
```

Note that the *Maven* build file for this project specifies a *war* file name of *ROOT*, which results in this application being deployed to the root context of the server. A route is also created so that the service is accessible externally via *ROUTE_URL*. The route tells the deployed router to load balance any requests with the given host name among the replicas of the *presentation* service. This host name should map to the IP address of the hosts where the router has been deployed, not necessarily where the service is hosted. For clients outside of this network and for testing purposes, simply modify your */etc/hosts* file to map this host name to the IP address of the master host.


3.7. POPULATING DATABASES

Once all services have been instantiated and are successfully running, you can utilize the *demo.jsp* page included in the Presentation module to run a one-time initializing script which will populate the product and sales databases with the needed schemas and seed data to make the eCommerce example runnable:


Example: <http://presentation.bxms.ose/demo.jsp>

Figure 3.2. Presentation Landing Page After Population

ABC HD32CS5002 32-inch LED TV

| | | | |
|---|------------------------|---|---|
|  | HD LED Picture Quality | Product Dimensions: 29 x 3 x 17 Product Weight: 17 | \$249.99 Availability: 52 |
| | ConnectShare Movie | | |
| | Wide Color Enhancement | | |
| | Clear Motion Rate 60 | | |

ABC HD42CS5002 42-inch LED TV

| | | | |
|---|------------------------|---|---|
|  | HD LED Picture Quality | Product Dimensions: 37 x 2 x 22 Product Weight: 20 | \$424.95 Availability: 64 |
| | ConnectShare Movie | | |
| | Wide Color Enhancement | | |
| | Clear Motion Rate 60 | | |

The system is now ready to accept new user registrations and transactions.

3.8. REPLICAS

Describe the deployment configuration of your services and verify how many instances have been configured and deployed. For example:

```
$ oc describe dc product-service
Name: product-service
Created: 4 days ago
Labels: app=product-service
Latest Version: 1
Triggers: Config, Image(product-service@latest, auto=true)
Strategy: Rolling
Template:
  Selector: app=product-service,deploymentconfig=product-service
  Replicas: 1
  Containers:
    NAME IMAGE ENV
  ...
  Replicas: 1 current / 1 desired
  Selector: app=product-service,deployment=product-service-1,deploymentconfig=product-service
  Labels: app=product-service,openshift.io/deployment-config.name=product-service
  Pods Status: 1 Running / 0 Waiting / 0 Succeeded / 0 Failed
No events.
```

Based on this default configuration, each service will have at most one replica, meaning the service will be backed by a single pod. In the event of a service container failure or node failure, a new service pod will be deployed to a healthy node as long as there is an active master node. However, it is often desirable to balance load between multiple pods of a service to avoid lengthy downtimes while failed containers are replaced.

Refer to the [OpenShift Developer Guide](#) for deployment configuration details and guidance on properly specifying the number of desired replicas.

To [manually scale](#) a service and verify this feature, use the **oc scale** command and provide the number of desired replicas for a given deployment configuration:

```
$ oc scale dc product-service --replicas=3
deploymentconfig "product-service" scaled
```

There will now be 3 separate pods running this service. To verify, query all pods configured for *product-service*:

```
$ oc get pods -l app=product-service
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------|-------|---------|----------|-----|
| product-service-1-8ag1z | 1/1 | Running | 0 | 40s |
| product-service-1-mhyfu | 1/1 | Running | 0 | 40s |
| product-service-1-mjis5 | 1/1 | Running | 0 | 5m |

Requests received via exposed route or internal call from another OpenShift by its service/host name (eg. Presentation calls to gateway-service) are handled by an internal proxy in such a way that the load is balanced between available replicas, failing over when necessary.

3.9. AGGREGATED LOGGING

In order to view all container logs in a central location for ease of analytics and troubleshooting, utilize the Logging template to instantiate an EFK stack.

Begin by creating a new project to host your logging services:

```
$ oc new-project logging --display-name="Aggregated Container Logging"
--description="Aggregated EFK Logging Stack"
```

Next, ensure the template required for logging has been installed during Ansible-based OpenShift installation:

```
$ oc describe template logging-deployer-template -n openshift
```

If the template is available, you'll see a large amount of output about the template. If the template is not available, you'll see the following error:

```
$ Error from server (NotFound): templates "logging-deployer-template"
not found
```

Should the template not be available, create it from the template which was fetched as part of the *yaml-templates* operations earlier:

```
$ cd ~/yaml-templates
$ oc create -f logging-deployer.yaml
```

Prior to using the template to start up the deployment assistant pod, several account-oriented components need to be created. To simplify this process, a bash script has been provided in the *yaml-templates* directory. Running the script as a user with *cluster-admin* privileges is required. Specifying the name of the newly created logging project as an argument:

```
$ cd ~/yaml-templates
$ ./logging-accounts.sh logging
```

Note that the output of the script may have some *Error* lines, indicating that attempts to remove some components that don't exist were made in order to properly handle incremental executions. These errors are not indicative of a script failure. After executing the script, use the previously mentioned template to create the deployment pod, substituting the *KIBANA_HOSTNAME* and *PUBLIC_MASTER_URL* as needed:

```
oc new-app logging-deployer-template --param KIBANA_HOSTNAME=efk-
kibana.bxms.ose --param ES_CLUSTER_SIZE=2 --param
PUBLIC_MASTER_URL=https://middleware-
master.cloud.lab.eng.bos.redhat.com:8443
```

The deployer pod has now been created and should be monitored for completion:

```
$ oc get pods
NAME                                READY    STATUS    RESTARTS   AGE
logging-deployer-rpqz8              1/1      Running   0           45s

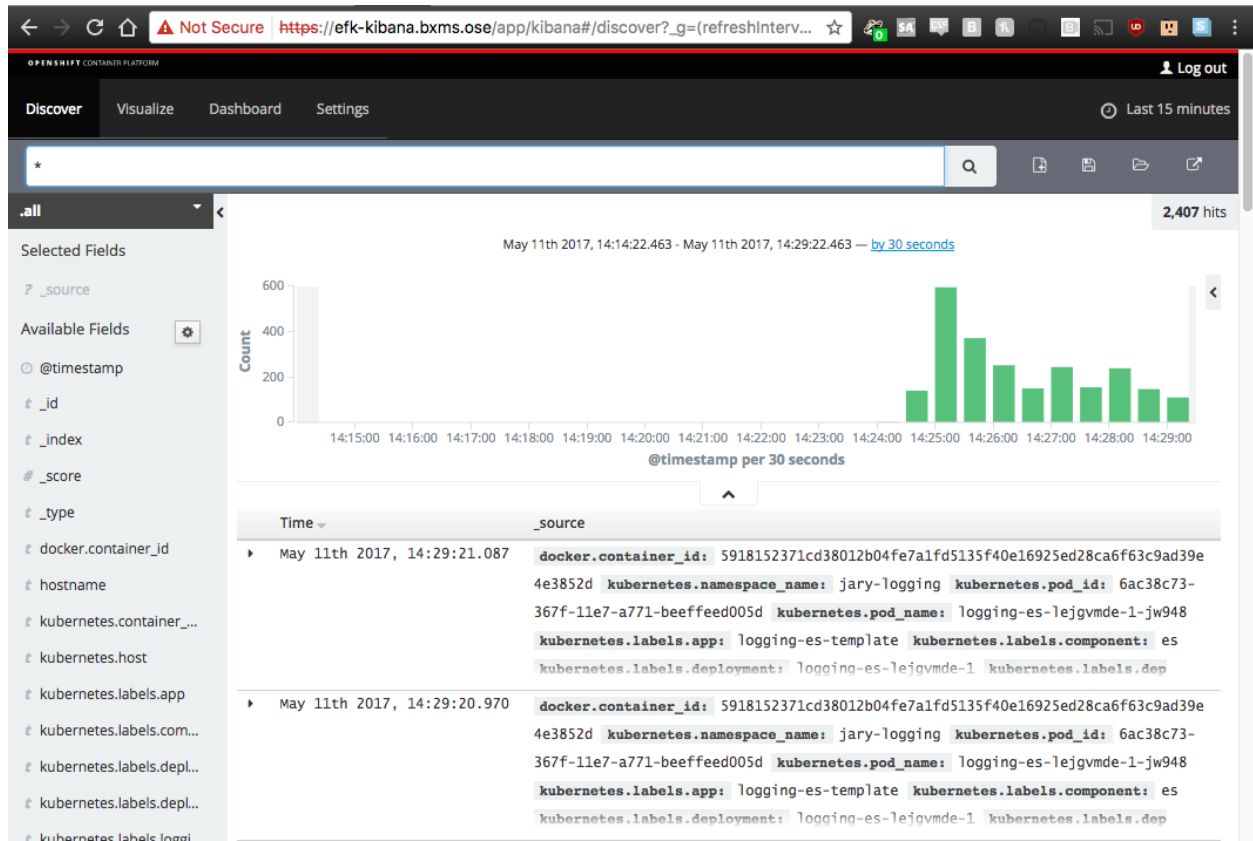
$ oc logs -f pod/logging-deployer-rpqz8
```

Once the deployer has completed its function, there will be a "Success!" entry in the log, followed by a description of several commands to follow next. For this project's configuration, the next and only additional required step is to label each node used to host Fluentd, in order to gather container logs:

```
$ oc label node/middleware-node1.cloud.lab.eng.bos.redhat.com:8443
logging-infra-fluentd=true
$ oc label node/middleware-node2.cloud.lab.eng.bos.redhat.com:8443
logging-infra-fluentd=true
```

Once all pods have started, the aggregated logging interface can be reached via the parameterized URL provided earlier. If you are following the trial environment, be sure to add the address to your */etc/hosts* file pointing to the cluster master. Given the scope of cluster information available via the EFK Stack, a user with the *cluster-admin* role is required to log in to the console.

Figure 3.3. Kibana Web console



CHAPTER 4. DESIGN AND DEVELOPMENT

4.1. OVERVIEW

For a discussion of microservices as a software architectural style, as well as the design and composition of the sample application, refer to the previously published reference architecture on [Building JBoss EAP 7 Microservices on OpenShift Container Platform](#). For further information regarding Fuse Integration Services 2.0 and its role in building Enterprise patterns and solutions, refer to the [Red Hat JBoss Fuse Integration Services 2.0 for OpenShift](#) documentation.

OpenShift provides an ideal platform for deploying, hosting and managing microservices. By deploying each service as an individual docker container, OpenShift helps isolate each service and decouples its lifecycle and deployment from that of other services. OpenShift can configure the desired number of replicas for each service and provide intelligent scaling to respond to varying load. Introducing Fuse Integration Services allows inclusion of a singular API Gateway pattern implementation to upstream clients and the ability to work with both event-driven and API-driven components. Monitoring workloads and container operations also becomes more manageable due to FIS's inclusion of HawtIO and OpenShift's templates for the EFK Logging Stack.

This sample application uses the Source-to-Image (S2I) mechanism to build and assemble reproducible container images from the application source and on top of supported OpenShift images, as well as OpenShift-provided templates for inclusion of both messaging and logging infrastructures.

4.2. APPLICATION STRUCTURE


The source code for the sample application is checked in to a public GitHub [repository](#). The code is organized as six separate directories, each structured as a Maven project with a *pom* file at the root. An aggregation *pom* file is provided at the top level of the root directory to build all six projects if needed, although this build file is neither required nor used by OpenShift.

4.3. PRESENTATION MODULE


The Presentation module of the project represents an upstream consumer instance that communicates with all of the microservices via the API Gateway pattern implementation (gateway-service). It's a graphical interface representing an eCommerce storefront where users can register or login, proceed to build and complete transactions, then track order histories through to completion. All communication with the Gateway is done through RESTful calls via Apache's *HttpClient*.

Figure 4.1. Presentation Module Landing Page

ABC HD32CS5002 32-inch LED TV

| | | | |
|---|------------------------|---|---|
|  | HD LED Picture Quality | Product Dimensions: 29 x 3 x 17 Product Weight: 17 | \$249.99 Availability: 52 |
| | ConnectShare Movie | | |
| | Wide Color Enhancement | | |
| | Clear Motion Rate 60 | | |

ABC HD42CS5002 42-inch LED TV

| | | | |
|---|------------------------|---|---|
|  | HD LED Picture Quality | Product Dimensions: 37 x 2 x 22 Product Weight: 20 | \$424.95 Availability: 64 |
| | ConnectShare Movie | | |
| | Wide Color Enhancement | | |
| | Clear Motion Rate 60 | | |

4.4. CUSTOMIZING RED HAT JBOSS EAP FOR PERSISTENCE

The *Product* and *Sales* services are both API-driven EAP applications and each have a database dependency and use their respective MySQL database to store and retrieve data. For the *Sales* service, the supported [Red Hat xPaaS EAP Image](#) bundles MySQL JDBC drivers, but the driver should be declared in the server configuration file, and a datasource would need to be described, in order for the application to access the database through a connection pool.

To make the necessary customizations, provide an updated server configuration file in the source code of the project built onto the image via S2I. The replacement configuration file should be named [standalone-openshift.xml](#) and placed in a directory called *configuration* at the root of the project.



Note

Some configuration can be performed by simply providing descriptive environment variables. For example, supplying the `DB_SERVICE_PREFIX_MAPPING` variable and value instructs the script to add MySQL and/or PostgreSQL datasources to the EAP instance. Refer to the [documentation](#) for details.

In order to make the required changes to the correct baseline, obtain the latest server configuration file. The supported image is available via the [Red Hat Registry](#). To view the original copy of the file, you can run this container directly:

```
# docker run -it registry.access.redhat.com/jboss-eap-7/eap70-openshift
cat /opt/eap/standalone/configuration/standalone-openshift.xml
```

```
<?xml version="1.0" ?>
```

```
<server xmlns="urn:jboss:domain:4.0">
  <extensions>
    ...
```

Declare the datasource with parameterized variables for the database credentials. For example, to configure the product datasource for the product service:

```
<subsystem xmlns="urn:jboss:domain:datasources:1.2">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ProductDS" enabled="true"
      use-java-context="true" pool-name="ProductDS">
      <connection-url>
        jdbc:mysql://${env.DATABASE_SERVICE_HOST:product-db}
          :${env.DATABASE_SERVICE_PORT:3306}/${env.MYSQL_DATABASE:product}
      </connection-url>
      <driver>mysql</driver>
      <security>
        <user-name>${env.MYSQL_USER:product}</user-name>
        <password>${env.MYSQL_PASSWORD:password}</password>
      </security>
    </datasource>
```

The datasource simply refers to the database driver as *mysql*. Declare the driver class in the same section after the datasource:

```
...
</datasource>
<drivers>
  <driver name="mysql" module="com.mysql">
    <xa-datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</xa-datasource-
class>
  </driver>
```

With the above configuration, environment variables are substituted to specify connection details and the database host name is resolved to the name of the OpenShift service hosting the database.

The default EAP welcome application is [disabled](#) in the Red Hat xPaaS EAP image. To deploy the *Presentation* application to the root context, rename the *warName* to *ROOT* in the **Maven pom** file:

```
<build>
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <plugin>
      <artifactId>maven-war-plugin</artifactId>
      <version>${version.war.plugin}</version>
      <configuration>
        <warName>ROOT</warName>
        <failOnMissingWebXml>>false</failOnMissingWebXml>
      </configuration>
    </plugin>
  </plugins>
</build>
```

4.5. GATEWAY SERVICE

The Gateway Service, along with the other message-oriented services, utilize [Apache Camel Java DSL](#) to define the various needed endpoints and routes for the service. This particular service constructs a series of routes utilizing the [Camel Rest DSL](#) to accept inbound RESTful requests on port 9091. The [spark-rest](#) component is used with the Rest DSL in order to take advantage of URI wildcard capabilities when proxying multiple methods to the same context.

```
String customersUri = "http4://sales-service:8080/customers/?
bridgeEndpoint=true";
String productsUri = "http4://product-service:8080/products/?
bridgeEndpoint=true";

restConfiguration().component("spark-rest").host("0.0.0.0").port(9091);

rest("/billing/process")
    .post()
    .route()
        .to("amq:billing.orders.new?
transferException=true&jmsMessageType=Text")
        .wireTap("direct:warehouse");

rest("/billing/refund/") .post() .to("amq:billing.orders.refund?
transferException=true&jmsMessageType=Text"); rest("/customers")
    .get().toD(customersUri) .post().to(customersUri); rest("/customers/")
        .get().toD(customersUri)
        .post().toD(customersUri)
        .patch().toD(customersUri)
        .delete().toD(customersUri);

rest ("/products")
    .get().toD(productsUri)
    .post().toD(productsUri);

rest ("/products/*")
    .get().toD(productsUri)
    .post().toD(productsUri);

from("direct:warehouse")
    .routeId("warehouseMsgGateway")
    .filter(simple("${bodyAs(String)} contains 'SUCCESS'"))
    .inOnly("amq:topic:warehouse.orders?jmsMessageType=Text");
```

Each `rest()` declaration configures a new Rest endpoint consumer to feed the corresponding Camel route. Billing has two possible method calls, *process* and *refund*. While *process* is a POST with no parameters, *refund* uses a wildcard to account for the *orderId* which will be included in the URI of the request. Similarly, both *customers* and *products* have a variety of "verb" handlers (get, post, etc.) featuring a mixture of URI parameter patterns, all of which are mirrored through to the proxied service intact via wildcard. Finally, a direct route is used to filter successful transaction messages that have been *wire tapped* from the billing process route to pass on for fulfillment via messaging queue.

4.6. BILLING & WAREHOUSE SERVICES

The billing and warehouse services follow an approach that is similar to, although simpler than, the gateway service. As mentioned, both utilize the Camel Java DSL to configure routes for consuming from queues hosted on Red Hat JBoss A-MQ via the messaging component of our project. The

billing service's route takes messages from both the *billing.orders.new* and *billing.orders.refund* queues, demonstrates the unmarshaling capabilities of Camel to convert from JSON message format into POJO objects. It then passes them to a bean for further handling and logic. Once the bean is complete, the result is then marshaled back into JSON format and sent back to the waiting gateway service. The process order route and accompanying bean process is shown below. Note that the conversion between object types is often done by Camel implicitly via [Type Converters](#), but may not always be optimal in an intermediate gateway. In these cases, explicit conversion can be accomplished as demonstrated below.

Process requests and bean handler:

```
from("amq:billing.orders.new")
    .routeId("processNewOrders")
    .unmarshal(dataFormatFactory.formatter(Transaction.class))
    .bean(billingService, "process")
    .marshal(dataFormatFactory.formatter(Result.class));

---

public Result process(Transaction transaction) {

    Result result = new Result();

    logInfo("Asked to process credit card transaction: " +
transaction);
    result.setName(transaction.getCustomerName());
    result.setOrderNumber(transaction.getOrderNumber());
    result.setCustomerId(transaction.getCustomerId());

    Calendar now = Calendar.getInstance();
    Calendar calendar = Calendar.getInstance();
    calendar.clear();
    calendar.set(transaction.getExpYear(), transaction.getExpMonth(),
1);

    if (calendar.after(now)) {
        result.setTransactionNumber(random.nextInt(90000000) + 10000000);
        result.setTransactionDate(now.getTime());
        result.setStatus(Status.SUCCESS);
    } else {
        result.setStatus(Status.FAILURE);
    }

    return result;
}
```

After tagging an ownership header to the message as the simplistic means of claiming the message for processing in our example application, the warehouse route directly passes the transaction result message on to a processing bean, where the order is "fulfilled", and then a new call to the gateway is performed in order to allow the sales service to mark the order as shipped.

```
from("amq:topic:warehouse.orders?clientId=" + warehouseId)
    .routeId("fulfillOrder")

    .unmarshal(dataFormatFactory.formatter(Result.class))
```

```

        .process(new Processor() {
            @Override
            public void process(Exchange exchange) throws Exception {
                /*
                    In production cases, multiple warehouse instances would
be subscribed to
                    the warehouse.orders topic, so this processor could be
used to
                    referenced a shared data grid clustered over all
warehouse instances.
                    With proper geographical and inventory level
information, a decision
                    could be made as to whether this specific instance is
the optimal
                    warehouse to fulfill the request or not. Note that doing
so would
                    require a lock mechanism in the shared cache if the
choice algorithm
                    could potentially allow duplicate optimal choices.
                */

                // in this demo, only a single warehouse instance will be
used, so just
                // claim all messages and return them
                exchange.getIn().setHeader("ownership", "true");
            }
        })

        .filter(simple("${headers.ownership} == 'true'"))
            .bean(warehouseService, "fulfillOrder");

    ---

    public void fulfillOrder(Result result) throws Exception {

        HttpClient client = new DefaultHttpClient();
        JSONObject jsonObject = new JSONObject();
        jsonObject.put("status", "Shipped");

        URIBuilder uriBuilder = new URIBuilder("http://gateway-
service:9091/customers/"
            + result.getCustomerId()
            + "/orders/" + result.getOrderNumber());
        HttpPatch patch = new HttpPatch(uriBuilder.build());
        patch.setEntity(new StringEntity(jsonObject.toString(),
        ContentType.APPLICATION_JSON));
        logInfo("Executing " + patch);
        HttpResponse response = client.execute(patch);
        String responseString = EntityUtils.toString(response.getEntity());
        logInfo("Got response " + responseString);
    }

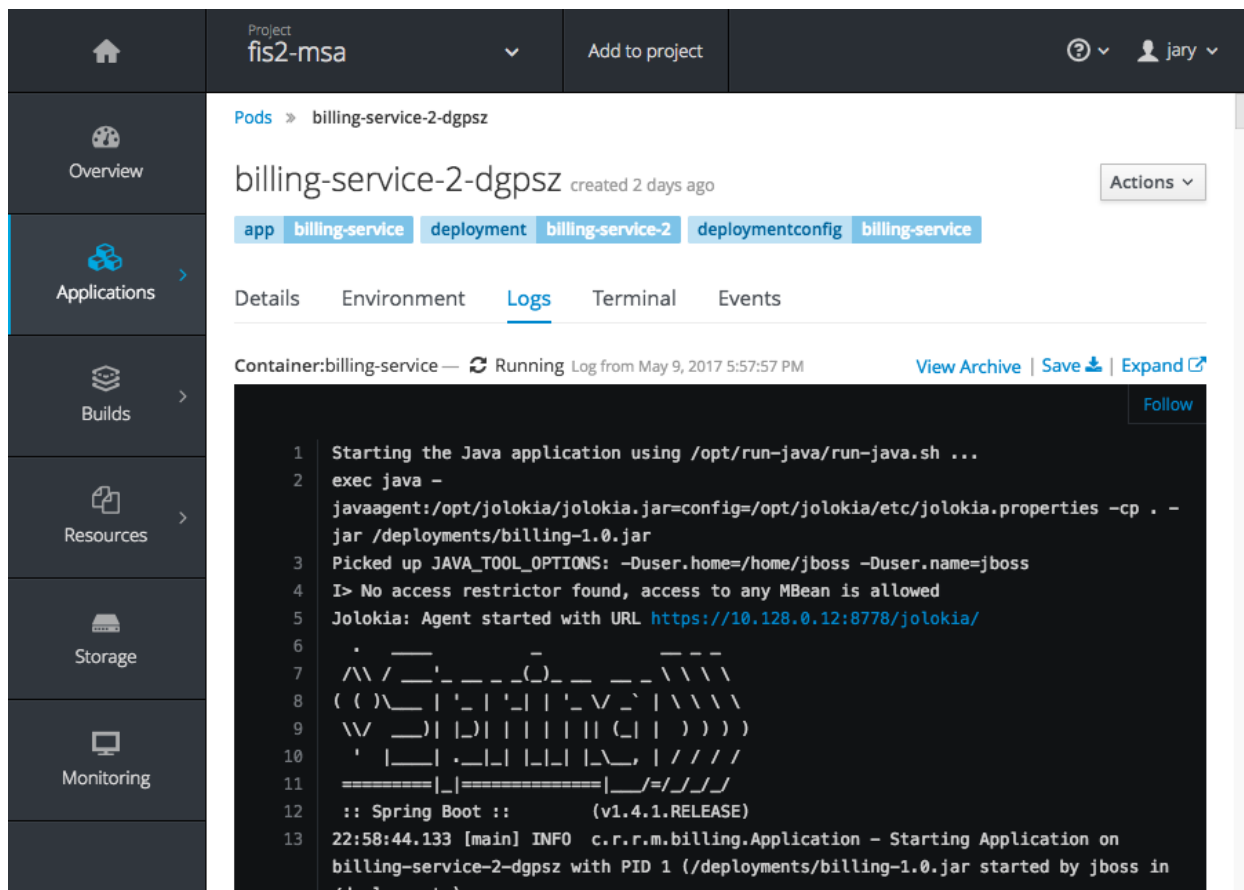
```

4.7. APPLICATION MONITORING

When originally configuring services in Chapter 3, each Fuse Integration Services component was

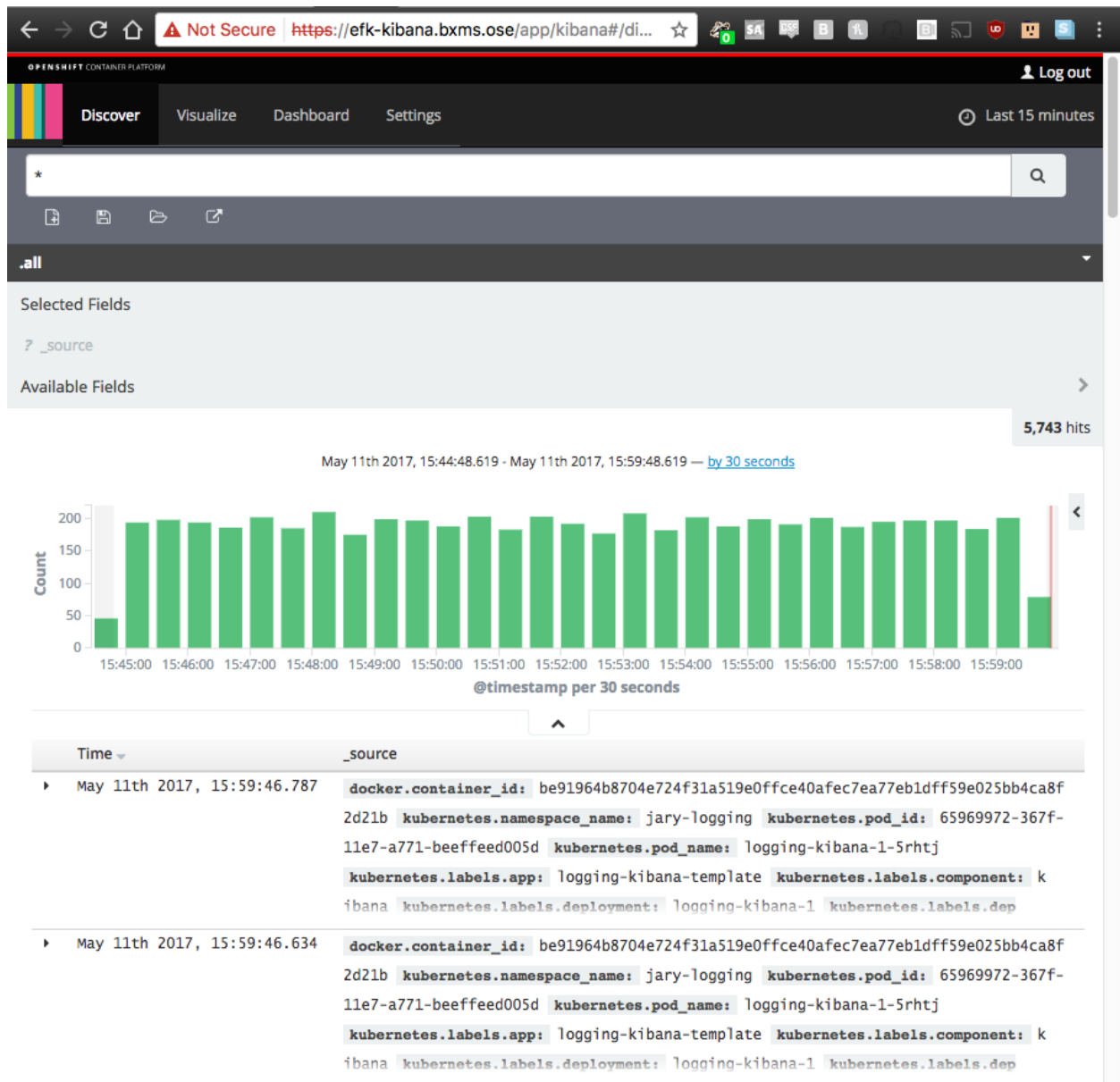
modified to include *jolokia* as the *name* attribute for the container's port 8778. In doing so, the system enables and exposes the integrated HawtIO console for monitoring and analytics at the container level. Once enabled, the tool can be reached in the OpenShift web console by opening the Logs tab of the target Pod and following the *View Archive* link now shown directly above the console log output:

Figure 4.2. Pod Log & View Archive



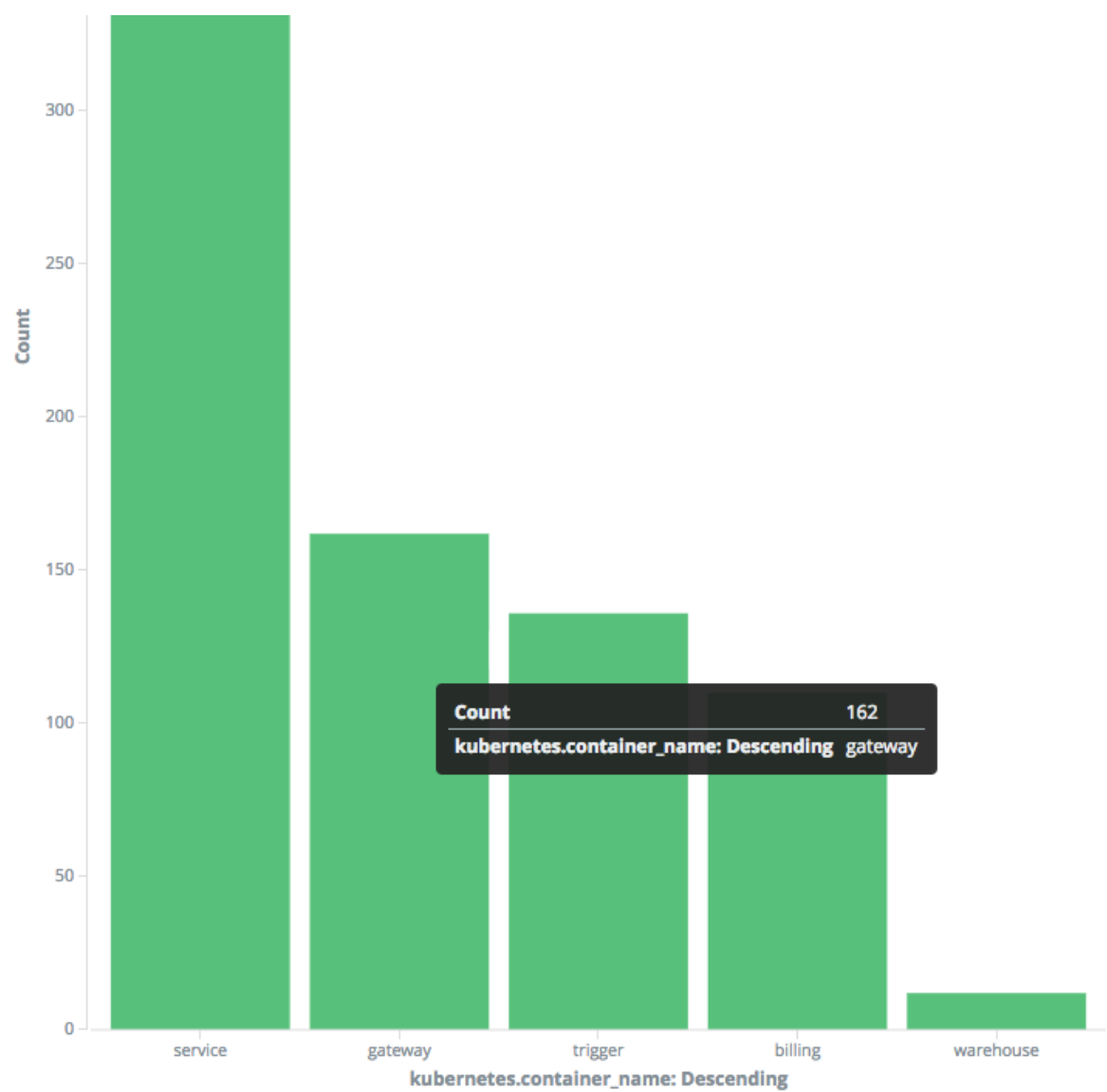
The individual container and the aggregated container logging, built on top of the EFK Stack within the *logging* project, offer a very similar interface. However, as Fluentd has been distributed across all nodes to collect and analyze all container output and feed it to Elasticsearch, the logging project version of the web console is all-inclusive, allowing for greater analyzation and side-by-side comparison and visualization of all actions across the nodes' containers.

Figure 4.3. Aggregated Console View



As shown below, with all logs centralized, metrics are collected and presented based on specific keywords defined by Fluentd. In this case, the chart shows a side-by-side count of events from each container, categorized with a labeled name.

Figure 4.4. Aggregated Visual View



CHAPTER 5. CONCLUSION

Leveraging OpenShift Container Platform and Red Hat JBoss Middleware xPaaS images together yields a powerful platform capable of hosting a highly-available and elastic microservice architecture. This reference architecture provides an example of such an architecture, with a microservices API Gateway pattern implementation at its center to allow for a variety of upstream consumer types to interact with both event-driven and RESTful downstream services in a singular, consistent way. In doing so, this document shows how Fuse Integration Services 2.0 can aid in development and configuration of Enterprise Integration Pattern components such as the gateway implementation and event-driven microservices. With this simplified approach to implementing integration patterns, development efforts remain focused on business requirements and related feature implementation. In closing, Red Hat JBoss Middleware offerings within OpenShift Container Platform, such as FIS 2.0, A-MQ Messaging, EFK Stack Aggregated Logging and Enterprise Application Platform, enable authoring of complete, maintainable, highly-available and elastic cloud-native microservice architecture solutions.

APPENDIX A. AUTHORSHIP HISTORY

| Revision | Release Date | Author(s) |
|----------|--------------|------------|
| 1.0 | May 2017 | Jeremy Ary |
| 1.1 | June 2017 | Jeremy Ary |

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 |
|-----------------|---|--------------------------|
| Babak Mozaffari | Manager, Software Engineering and Consulting Engineer | Technical Content Review |
| Keith Babo | Senior Principal Product Manager | Technical Content Review |
| Frederic Giloux | Senior Consultant | Technical Content Review |
| Calvin Zhu | Principal Software Engineer | Technical Content Review |

APPENDIX C. REVISION HISTORY

| | | |
|----------------|------------|----|
| Revision 1.0-0 | 2017-05-12 | JA |
|----------------|------------|----|