Reference Architectures 2017
Building Polyglot Microservices on OpenShift Container Platform 3

Calvin Zhu

Reference Architectures 2017 Building Polyglot Microservices on OpenShift Container Platform 3

Calvin Zhu
refarch-feedback@redhat.com

Legal Notice

Abstract

The choice of technology, programming language and supporting frameworks is often a compromise between the needs of the various parts of the application, and the skill set of the developers. One advantage of the distributed nature of microservices is the ability to abandon this one-size-fits-all approach and select the best fit for each service. This reference architecture builds upon previous work describing microservices, and a pattern for building a microservice architecture using Red Hat JBoss Enterprise Application Platform 7.0 on top of Red Hat OpenShift, to build a polyglot microservice environment, hosted on an on-premise OpenShift cloud. In addition to the previously designed MySQL and EAP services, two new services based on Ruby and Node.js images are included in the microservice architecture environment.

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

The choice of technology, programming language and supporting frameworks is often a compromise between the needs of the various parts of the application, and the skill set of the developers. One advantage of the distributed nature of microservices is the ability to abandon this one-size-fits-all approach and select the best fit for each service.

This reference architecture builds upon previous work describing microservices, and a pattern for building a microservice architecture using **`Red Hat JBoss Enterprise Application Platform 7.0`** on top of **`Red Hat OpenShift`**, to build a **polyglot** microservice environment, hosted on an on-premise OpenShift cloud.

`OpenShift Container Platform 3 by Red Hat` is designed for on-premise, public, or hybrid cloud deployments. Built with proven open source technologies, Red Hat OpenShift is a container platform that helps application development and IT operations teams create and deploy apps with the speed and consistency that business demands.

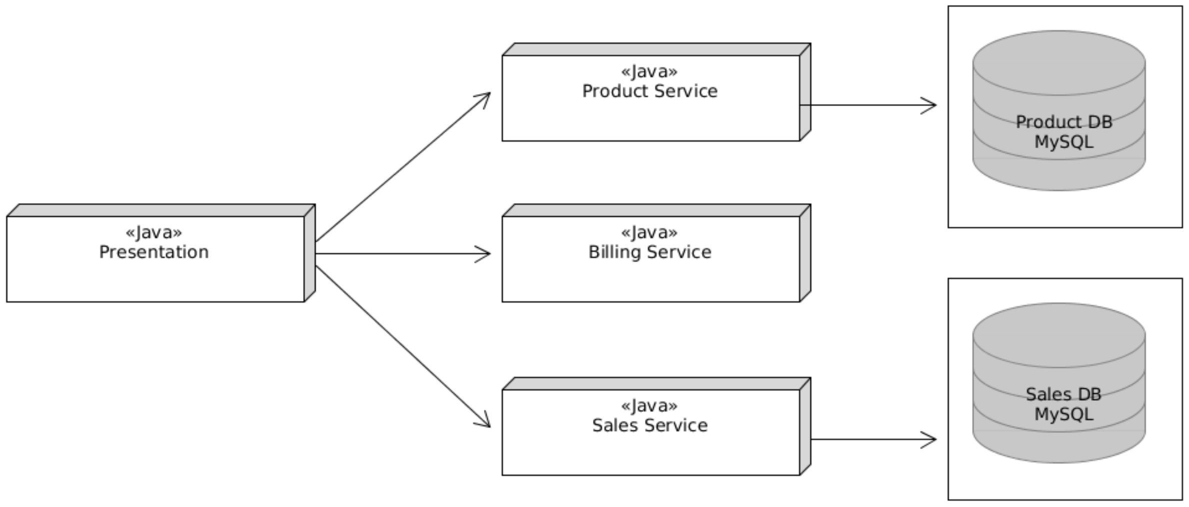
This effort assumes a properly configured Red Hat OpenShift production environment and refers the reader to documents and resources to help install and configure such an environment. The microservice architecture project is largely imported from previous reference architecture work, with individual services replaced with new implementations based on **`Ruby`** and **`Node.js`**.

CHAPTER 2. REFERENCE ARCHITECTURE ENVIRONMENT

This reference architecture designs, develops and deploys a distributed polyglot microservice architecture on top of Red Hat OpenShift. The application architecture is based on the design laid out and explained in depth in the reference architecture document on EAP Microservice Architecture, implemented on Red Hat OpenShift in previous reference architecture work.

The starting point for this project includes six services, where two of these are database services based on the standard and supported MySQL image. The remaining four images in the original reference architecture are based on the supported JBoss EAP xPaaS image and rely on the Source to Image (S2I) functionality to build and deploy. The EAP images are customized for two of these services to add a MySQL datasource.

Figure 2.1. Service Diagram, only Java EE implementation



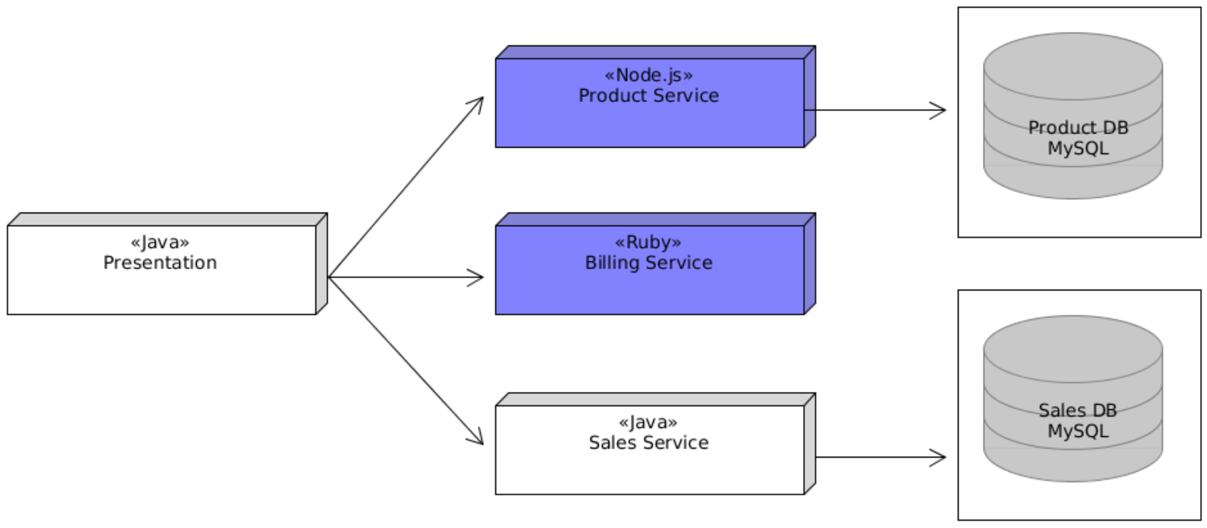
The services in the original implementation are loosely coupled, relying only on REST web service calls to communicate with one another. As a result, any of them can be replaced with a new implementation, as long as the API and behavior is not changed.

In this reference architecture, a new implementation is provided for two of these services, replacing the Java EE implementation with Ruby and Node.js, respectively.

The *Product* service is rewritten in Node.js, while the *Billing* service is rewritten in Ruby. No changes are required to the rest of the environment, and the new *Product* service will continue to talk to the same *Product* MySQL database.

This is the new diagram after replacing two services. The services in blue are the ones that have changed:

Figure 2.2. Service Diagram, polyglot implementation



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

The *presentation* service is the only entry point to the application and serves HTML to the client browser over HTTP. This web tier is stateless, and can therefore be replicated without concern for multicast communication or alternative EAP clustering setup. This service relies on the *product*, *sales*, and *billing* services, while the first two in turn rely on the MySQL *product-db* and *sales-db* services.

CHAPTER 3. CREATING THE ENVIORNMENT

3.1. OVERVIEW

This reference architecture may be deployed with either a single, or three master hosts. In both cases, it is assumed that *ocp-master1* refers to one (or the only) Red Hat OpenShift master host and that the environment includes two Red Hat OpenShift node hosts with the host names of *ocp-node1* and *ocp-node2*.

It is further assumed that Red Hat 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 Red Hat OpenShift through its *identity providers*.

3.2. BUILD AND DEPLOY

3.2.1. Creating a New Project

Log in to a master host as the *root* user and create a new project, assigning the administrative rights of the project to the previously created Red Hat OpenShift user:

```
# oadm new-project msa \ --display-name="OpenShift 3 MSA" \ --
description="This is a polyglot microservice architecture environment
built on on OCP v3.4" \ --admin=ocuser
Created project msa
```

3.2.2. Red Hat OpenShift Login

Once the project has been created, all remaining steps can be performed as the regular Red Hat OpenShift user. Log in to the master host machine or switch the user, and use the *oc* utility to authenticate against Red Hat OpenShift:

```
# su - ocuser
$ oc login -u ocuser --certificate-authority=/etc/origin/master/ca.crt
\ --server=https://ocp-master1.hostname.example.com:8443
Authentication required for https://ocp-
master1.hostname.example.com:8443 (openshift)
Username: ocuser
Password: PASSWORD
Login successful.

Using project "msa".
Welcome! See 'oc help' to get started.
```

The recently created *msa* project is the only project for this user, which is why it is automatically selected as the default working project of the user.

3.2.3. MySQL Images

This reference architecture includes two database services built on the supported MySQL image. This reference architecture uses version 5.6 of the MySQL image.

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 by Red Hat 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 Red Hat 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 OpenShift 3 MSA on EAP 7 (msa) on server https://ocp-
master1.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 deploying other services that may depend on them. The service log clearly shows if 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)
```

3.2.3.1. Creating tables for the Product service

While the *Sales* service uses Hibernate to create the required tables in the MySQL database, the *Product* service uses SQL statements to access the database, so the tables also need to be manually created using the following SQL statements.

Log into the *product-db* image from the MySQL client and use the following SQL statements to create three tables.

```
USE product;

CREATE TABLE Product (SKU BIGINT NOT NULL AUTO_INCREMENT, DESCRIPTION
VARCHAR(255), HEIGHT NUMERIC(5,2) NOT NULL, LENGTH NUMERIC(5,2) NOT
NULL, NAME VARCHAR(255), WEIGHT
NUMERIC(5,2) NOT NULL, WIDTH NUMERIC(5,2) NOT NULL, FEATURED BOOLEAN
NOT NULL, AVAILABILITY INTEGER NOT NULL, IMAGE VARCHAR(255), PRICE
NUMERIC(7,2) NOT NULL, PRIMARY KEY (SKU)) AUTO_INCREMENT = 10001;

CREATE TABLE Keyword (KEYWORD VARCHAR(255) NOT NULL, PRIMARY KEY
(KEYWORD));

CREATE TABLE PRODUCT_KEYWORD (ID BIGINT NOT NULL AUTO_INCREMENT,
KEYWORD VARCHAR(255) NOT NULL, SKU BIGINT NOT NULL, PRIMARY KEY (ID));


ALTER TABLE PRODUCT_KEYWORD ADD INDEX FK_PRODUCT_KEYWORD_PRODUCT (SKU),
add constraint FK_PRODUCT_KEYWORD_PRODUCT FOREIGN KEY (SKU) REFERENCES
Product (SKU);

ALTER TABLE PRODUCT_KEYWORD ADD INDEX FK_PRODUCT_KEYWORD_KEYWORD
(KEYWORD), add constraint FK_PRODUCT_KEYWORD_KEYWORD FOREIGN KEY
(KEYWORD) REFERENCES Keyword (KEYWORD);
```

3.2.4. JBoss EAP 7 xPaaS Images

The *Sales* and *Presentation* services rely on Red Hat 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 as the *root* user, but first switch from the default to the *openshift* project as the root user:

```
# oc project openshift
Now using project "openshift" on server "https://ocp-
master1.hostname.example.com:8443".
```

Query the configured image streams for the project:

```
# oc get imagestreams
NAME       DOCKER REPO
...omitted...
jboss-eap70-openshift
registry.access.redhat.com/jboss-eap-7/eap70-openshift
latest,1.4,1.3 + 2 more...         3 weeks ago
...omitted...
```

3.2.5. Ruby Images

The *Billing* service relies on Red Hat OpenShift S2I for Ruby applications and uses the Ruby image. This reference architecture uses version 2.3 of the Ruby image.

3.2.6. Node.js Images

The *Product* service relies on Red Hat OpenShift S2I for Node.js applications and uses the Node.js image. This reference architecture uses version 4.0 of the Node.js image.

3.2.7. Building the Services

The microservice application for this reference architecture is made available in a public git repository at https://github.com/RHsyseng/MSA-Polyglot-OCP. This includes four distinct services, provided as subdirectories of this repository: *Billing*, *Product*, *Sales*, and *Presentation*.

Sales, and *Presentation* are implemented in Java and use the JBoss EAP 7 xPaaS Images. The *Billing* service is implemented in Ruby, and the *Product* service is implemented in Node.js.

Start by building and deploying the *Billing* service, which has no dependencies on either a database or another service. Switch back to the regular user with the associated *msa* project and run:

```
$ oc new-app ruby~https://github.com/RHsyseng/MSA-Polyglot-OCP.git --
context-dir=ruby_billing --name=billing-service
--> Found image dc510ba (10 weeks old) in image stream "ruby" in
project "openshift" under tag "2.3" for "ruby"

    Ruby 2.3
    --------
    Platform for building and running Ruby 2.3 applications

    Tags: builder, ruby, ruby23, rh-ruby23
    * A source build using source code from
https://github.com/RHsyseng/MSA-Polyglot-OCP.git will be created
      * The resulting image will be pushed to image stream "billing-
service:latest"
      * Use 'start-build' to trigger a new build
    * This image will be deployed in deployment config "billing-
service"
    * Port 8080/tcp will be load balanced by service "billing-service"
      * Other containers can access this service through the hostname
"billing-service"

--> Creating resources with label app=billing-service ...
    imagestream "billing-service" created
    buildconfig "billing-service" created
    deploymentconfig "billing-service" created
    service "billing-service" created
--> Success
    Build scheduled, use 'oc logs -f bc/billing-service' to track its
progress.
    Run 'oc status' to view your app.
```

Once again, *oc status* 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
```

Once this service has successfully deployed, use similar commands to deploy the *Product* and *Sales* services, bearing in mind that both have a database dependency and rely on previous MySQL services. Change any necessary default database parameters by passing them as environment variables.

To deploy the *Product* service:

```
$ oc new-app -e MYSQL_USER=product -e MYSQL_PASSWORD=password
nodejs~https://github.com/RHsyseng/MSA-Polyglot-OCP.git --context-
dir=nodejs_product --name=product-service
```

To deploy the *Sales* service.

```
$ oc new-app -e MYSQL_USER=sales -e MYSQL_PASSWORD=password jboss-
eap70-openshift~https://github.com/RHsyseng/MSA-Polyglot-OCP.git --
context-dir=Sales --name=sales-service
```

Finally, deploy the *Presentation* service, which exposes a web tier and an aggregator that uses the three previously deployed services to fulfill the business request:

```
$ oc new-app jboss-eap70-openshift~https://github.com/RHsyseng/MSA-
Polyglot-OCP.git --context-dir=Presentation --name=presentation
```

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.

Once all four services have successfully deployed, the *presentation* service can be accessed through a browser to verify application functionality. First create a route to expose this service to clients outside the Red Hat OpenShift environment:

```
$ oc expose service presentation --hostname=msa.example.com
NAME    HOST/PORT    PATH    SERVICE    LABELS             TLS
TERMINATION
presentation msa.example.com  presentation app=presentation
```

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.

3.2.8. 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
...omitted...
 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 not have any more than one replica, which means the Red Hat OpenShift service will be backed by a single pod. In the event of the failure of a service container or Red Hat OpenShift node, as long as there is an active master host, a new pod for the service will be deployed to a healthy node. However, it is often desirable to balance load between multiple pods of a service and also avoid a lengthy downtime while a failed pod is replaced.

Refer to the Red Hat OpenShift Developer Guide for deployment configuration details and 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*, for example:

```
$ 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
```

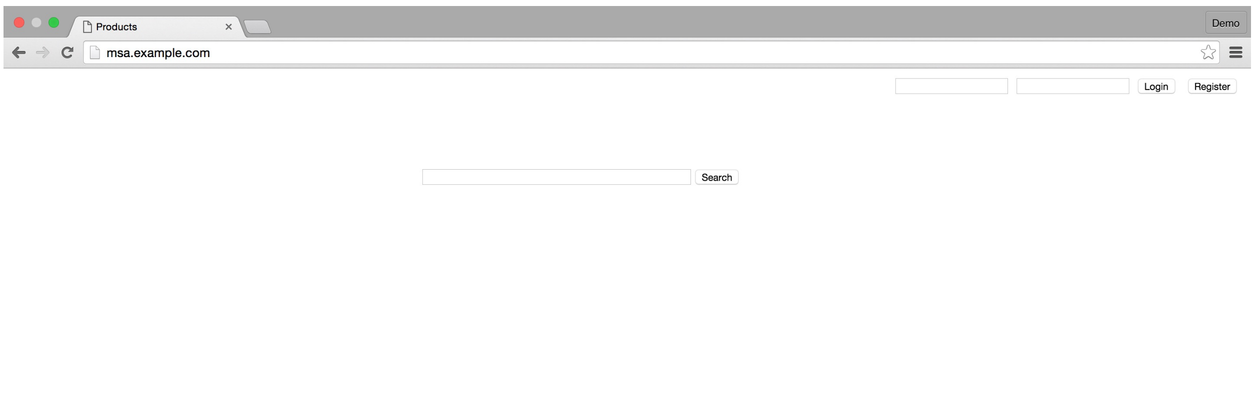
Traffic that is received through an exposed route or from an existing Red Hat OpenShift service referencing another service by its service / host name (as is the case with the *presentation* service calling the other services) is handled by an internal proxy and balances the load between available replicas, while failing over when necessary.

3.3. RUNNING THE APPLICATION

3.3.1. Browser Access

To use the application, simply point your browser to the address exposed by the route. This address should ultimately resolve to the IP address of the Red Hat OpenShift host where the router is deployed.

Figure 3.1. Application Homepage before initialization

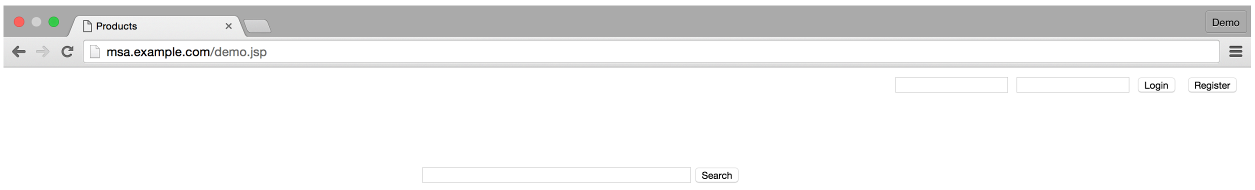


At this stage, the database tables are still empty and content needs to be created for the application to function properly.

3.3.2. Sample Data

The application includes a demo page that when triggered, populates the database with sample data. To use this page and populate the sample data, point your browser to http://msa.example.com/demo.jsp:

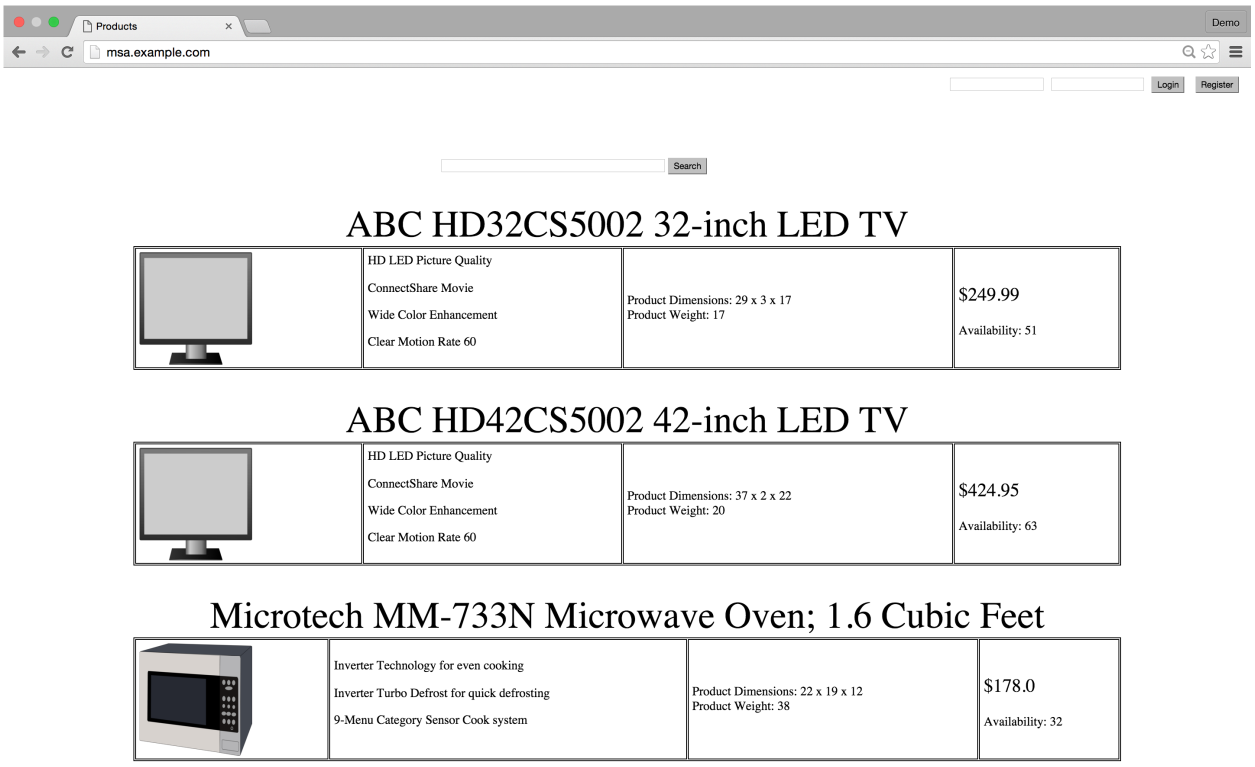
Figure 3.2. Trigger sample data population



3.3.3. Featured Product Catalog

After populating the product database, the demo page redirects your browser to the route address, but this time you will see the featured products listed:

Figure 3.3. Application Homepage after initialization



ABC HD32CS5002 32-inch LED TV

	HD LED Picture Quality ConnectShare Movie Wide Color Enhancement Clear Motion Rate 60	Product Dimensions: 29 x 3 x 17 Product Weight: 17	$249.99 Availability: 51

ABC HD42CS5002 42-inch LED TV

	HD LED Picture Quality ConnectShare Movie Wide Color Enhancement Clear Motion Rate 60	Product Dimensions: 37 x 2 x 22 Product Weight: 20	$424.95 Availability: 63

Microtech MM-733N Microwave Oven; 1.6 Cubic Feet

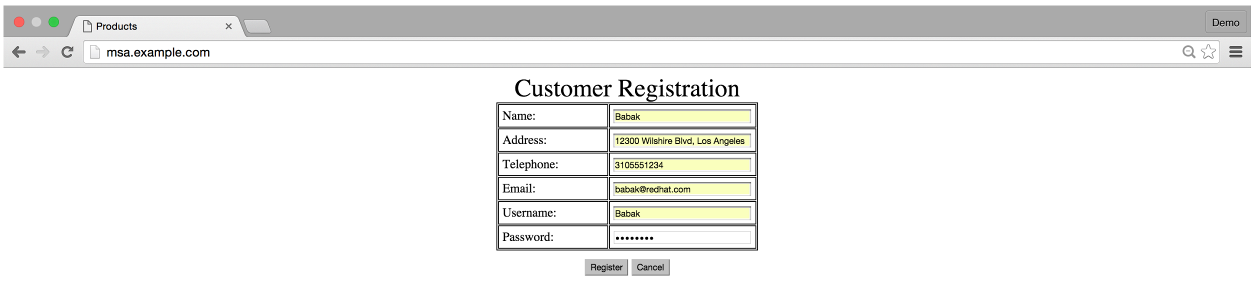
	Inverter Technology for even cooking Inverter Turbo Defrost for quick defrosting 9-Menu Category Sensor Cook system	Product Dimensions: 22 x 19 x 12 Product Weight: 38	$178.0 Availability: 32

3.3.4. User Registration

Anonymous users are constrained to browsing inventory, viewing featured products and searching the catalog. To use other features that result in calls to the *Sales* and *Billing* services, a valid customer must be logged in.

To register a customer and log in, click on the *Register* button in the top-right corner of the screen and fill out the registration form:

Figure 3.4. Customer Registration Form



Customer Registration

Name:	Babak
Address:	12300 Wilshire Blvd, Los Angeles
Telephone:	3105551234
Email:	babak@redhat.com
Username:	Babak
Password:	••••••••

After registration, the purchase button allows customers to add items to their shopping cart, to subsequently visit the shopping cart and check out, and review their order history.

For details on application functionality and how each feature is implemented by the provided services and exposes through their REST API, refer to the previously published reference architecture on Building microservices with JBoss EAP 7.

CHAPTER 4. DESIGN AND DEVELOPMENT

4.1. OVERVIEW

Red Hat OpenShift provides an ideal platform for deploying, hosting and managing microservices. By deploying each service as an individual **`Docker`** container, Red Hat OpenShift helps isolate each service and decouples its lifecycle and deployment from that of other services. Red Hat OpenShift can configure the desired number of replicas for each service and provide intelligent scaling to respond to varying load.

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 Red Hat OpenShift images.

This reference architecture builds on previously published reference architecture papers. Both the *Sales*, and *Presentation* services in this reference architecture remain unchanged; please refer to Building microservices with JBoss EAP 7 for further details on the design and implementation of these two services. The *Billing* and *Product* services are replaced, with Ruby and Node.js used as the technology to implement the two services respectively. Further information about the design and implementation of these two services follows.

4.2. APPLICATION STRUCTURE

The source code for the sample application is checked in to a public GitHub repository. An aggregation *POM* file is provided at the top level of the root directory to build the two Java projects (*Sales* and *Presentation*) if needed, although this build file is neither required nor used by Red Hat OpenShift.

4.3. CUSTOMIZING THE APPLICATION SERVER

The *Product* and *Sales* services each have a database dependency and use their respective MySQL database to store and retrieve data. For the *Sales* service, the supported xPaaS Middleware Images bundles MySQL JDBC drivers, but the driver would have to 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 customizations to a server, provide an updated server configuration file. 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 Red Hat OpenShift images for details.

In order to make the required changes to the correct baseline, obtain the latest server configuration file. The supported image is located at *registry.access.redhat.com/jboss-eap-7/eap70-openshift*. To view the original copy of the file, you can run this Docker 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>
```

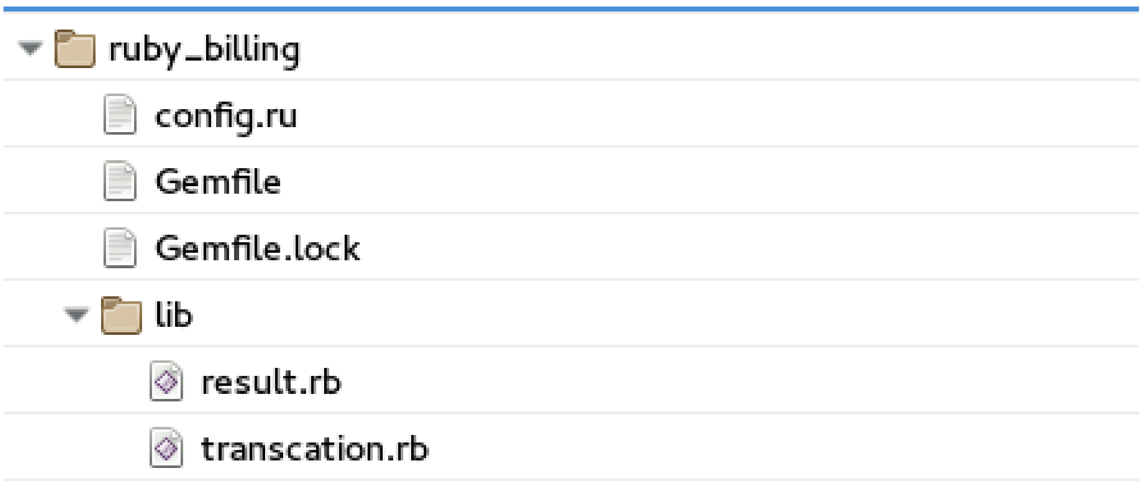
With the above configuration, environment variables are substituted to specify connection details and the database host name is resolved to the name of the Red Hat OpenShift service hosting the database.

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>
        <!-- Java EE 7 doesn’t require web.xml, Maven needs to catch up! -->
        <failOnMissingWebXml>false</failOnMissingWebXml>
      </configuration>
    </plugin>
  </plugins>
</build>
```

4.4. BILLING SERVICE IN RUBY

The implementation of the *Billing* service in Ruby consists of the following files:


ruby_billing
config.ru
Gemfile
Gemfile.lock
lib
result.rb
transcation.rb

- *config.ru* is the entry point and contains the business logic for the *Billing* service
- *transcation.rb* is a simple class providing the required fields for a purchase transaction
- *result.rb* provides the required fields to transport the response data in JSON format
- Gemfile contains project gem dependencies
- Gemfile.lock captures a complete snapshot of all the gems in Gemfile along with their associated dependencies

4.5. BILLING SERVICE DETAILS

The content of *transcation.rb* is as follows:

```
class Transcation

  attr_accessor :creditCardNumber, :expMonth, :expYear,
:verificationCode, :billingAddress, :customerName, :orderNumber,
:amount

end
```

In addition to fields, *result.rb* also provides a to_json method which returns a JSON string representing the object:

```
class Result

  attr_accessor :status, :name, :orderNumber, :transactionDate,
:transactionNumber
```

```
def to_json(*a)
  {
     'status'   => @status,
     'name'         => @name,
 'orderNumber' => @orderNumber,
 'transactionDate' => @transactionDate,
 'transactionNumber' => @transactionNumber
  }.to_json(*a)
end

end
```

The dependencies declared in *Gemfile* include *Puma*, a Ruby web server built for concurrency, and *Sinatra*, a light Ruby web application framework used for REST web services:

```
source 'https://rubygems.org'

gem 'sinatra'
gem 'puma'
```

The package configuration declared in *Gemfile.lock* is as follows:

```
GEM
  remote: https://rubygems.org/
  specs:
    puma (3.4.0)
    rack (1.6.4)
    rack-protection (1.5.3)
      rack
    sinatra (1.4.7)
      rack (~> 1.5)
      rack-protection (~> 1.4)
      tilt (>= 1.3, < 3)
    tilt (2.0.5)

PLATFORMS
  ruby

DEPENDENCIES
  puma
  sinatra
```

The business logic for the service, within *config.ru*, has 2 sections. The first section sets up the required libraries, and ensures the content type is JSON:

```
require 'bundler/setup'
require 'json'
require_relative 'lib/result'
require_relative 'lib/transcation'

Bundler.require(:default)

class Application < Sinatra::Base


before do
```

```
 content_type 'application/json'
end
```

The second section provides methods to handle the business request.

The *refund* method is a simple reply message simulating the refund process:

```
post '/billing/refund/:id' do
 "refund for transcation number #{params[:id]}"
end
```

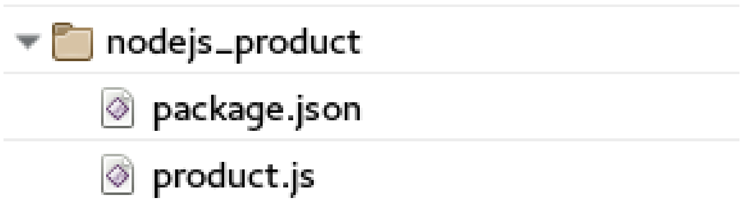
The *process* method validates the credit card expiration date, then returns a JSON response:

```
post '/billing/process' do
 begin
   post_data =  JSON.parse request.body.read
   if post_data.nil? or !post_data.has_key?('creditCardNumber')  or
!post_data.has_key?('verificationCode')
     puts "ERROR, no credit card number or verification code!"
   else
     transcation = Transcation.new
  transcation.creditCardNumber = post_data['creditCardNumber']
  transcation.expMonth = post_data['expMonth']
  transcation.expYear = post_data['expYear']
  transcation.verificationCode = post_data['verificationCode']
  transcation.billingAddress = post_data['billingAddress']
  transcation.customerName = post_data['customerName']
  transcation.orderNumber = post_data['orderNumber']
  transcation.amount = post_data['amount']

  puts "creditCardNumber  #{transcation.creditCardNumber}"
  puts "expMonth          #{transcation.expMonth}"
  puts "expYear           #{transcation.expYear}"
  puts "billingAddress    #{transcation.billingAddress}"
  puts "customerName      #{transcation.customerName}"
  puts "orderNumber       #{transcation.orderNumber}"
  puts "amount            #{transcation.amount}"

  result = Result.new
  result.name = transcation.customerName
  result.orderNumber = transcation.orderNumber
  result.transactionDate = DateTime.now
  result.transactionNumber = 9000000 + rand(1000000)

  if transcation.expYear.to_i < Time.now.year.to_i or
(transcation.expYear.to_i == Time.now.year.to_i and
transcation.expMonth.to_i <= Time.now.month.to_i)
   result.status = "FAILURE"
  else
   result.status = "SUCCESS"
  end

  result.to_json
```

```
    end
  end
end
```

4.6. PRODUCT SERVICE IN NODE.JS

The structure of the *Product* service, written in Node.js, is as follows:



The implementation for the Node.js service is provided in the *product.js* file. The additional *package.json* file simply contains metadata about the module, including the list of dependencies to install from *npm* when running npm install.

4.7. PRODUCT SERVICE DETAILS

The *package.json* serves as documentation for packages this project depends on, and also specifies the version of packages this project is using:

```
{
  "name": "productWs",
  "version": "0.0.1",
  "description": "Product service implemented in Node.js with mySQL",
  "main": "product.js",
  "dependencies": {
    "express": "~4.14.0",
    "mysql": "2.11.1",
    "body-parser": "1.15.2"
  },
  "scripts": {
    "start": "node product.js"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/RHsyseng/MSA-Polyglot-OCP.git"
  },
  "author": "Calvin Zhu",
  "license": "BSD",
  "bugs": {
    "url": "https://github.com/RHsyseng/MSA-Polyglot-OCP/issues"
  },
  "homepage": "https://github.com/RHsyseng/MSA-Polyglot-OCP"
}
```

The business logic for *Product* service is implemented in the form of several business methods, to respond to REST Web Service calls from the *Presentation* service.

The first part sets up the required libraries, also setting up the *http* body parser for JSON input.

```
var express = require('express');
var app     = express();
var bodyParser = require('body-parser');

app.use(bodyParser.json());
```

The following section deals with the MySQL database connection pool setup. The database parameters are passed from Openshift environment variables.

```
const dbUser = process.env.MYSQL_USER;
const dbPassword = process.env.MYSQL_PASSWORD;

var dbHost = process.env.MYSQL_HOST;
var dbDatabase = process.env.MYSQL_DATABASE;

if (dbHost == null) dbHost = 'product-db';
if (dbDatabase == null) dbDatabase = 'product';

const mysql  = require('mysql');
const pool = mysql.createPool({
  connectionLimit : 5,
  host            : dbHost,
  user            : dbUser,
  password        : dbPassword,
  database        : dbDatabase
});
```

They match the variables set up in the image creation script earlier:

MYSQL_USER=product,MYSQL_PASSWORD=password

The first method responds to *HTTP GET* requests and searches the *Product* table, based on either the *featured* column, or *keyword*, through a join.

```
//get either featured products or products with keyword
app.get('/product/products', function(req, httpRes) {
 if(req.query.featured == null && req.query.keyword == null) {
  httpRes.statusCode = 400;
  return httpRes.send('All products cannot be returned, need to provide
a search condition');
 }


 pool.getConnection(function(err, conn) {
  if (req.query.featured != null) {
   conn.query('select sku, availability, description, featured=1 as
featured, height, image, length, name, price, weight, width from
Product where featured=true', function(err, records) {
    if(err) throw err;
    httpRes.json(records);
   });
  } else if (req.query.keyword != null){
   conn.query('select sku, availability, description, featured=1 as
featured, height, image, length, name, price, weight, width from
Product where SKU in (select SKU from PRODUCT_KEYWORD where Keyword =
?)', req.query.keyword, function(err, records) {
    if(err) throw err;
    httpRes.json(records);
   });

  }
      conn.release();
 });
});
```

Another *GET* method is provided to search the *Product* table based on the *SKU* number.

```
//get based on sku #
app.get('/product/products/:sku', function(req, httpRes) {
 pool.getConnection(function(err, conn) {
  conn.query('select sku, availability, description, featured=1 as
featured, height, image, length, name, price, weight, width from
Product where SKU = ? ', req.params.sku, function(err, records) {
   if(err) throw err;
   httpRes.json(records[0]);
  });
      conn.release();
 });
});
```

Individual methods are provided to add new keywords and new products respectively, in response to *HTTP POST* request. The second method needs to insert rows into two tables, and therefore requires transaction setup:

```
//add keyword through post
app.post('/product/keywords', function(req, httpRes) {
 const record= { KEYWORD: req.body.keyword};
 pool.getConnection(function(err, conn) {
  conn.query('INSERT INTO Keyword SET ?', record, function(err,
records) {
   if(err) throw err;
   const result = {
    keyword : req.body.keyword,
      products : null}
     httpRes.json(result);
  });
      conn.release();
 });

});


//add product through post
app.post('/product/products', function(req, httpRes) {
 //To use "let" need strict mode in node version 4.*
 "use strict";
 pool.getConnection(function(err, dbconn) {

  // Begin transaction
  dbconn.beginTransaction(function(err) {
     if (err) { throw err; }
   let featured = 0;
   if (req.body.featured = 'true')
    featured = 1;

   let record= { DESCRIPTION: req.body.description, HEIGHT:
req.body.height, LENGTH: req.body.length,  NAME: req.body.name, WEIGHT:
req.body.weight, WIDTH: req.body.width, FEATURED: featured,
AVAILABILITY: req.body.availability, IMAGE: req.body.image, PRICE:
req.body.price};

   dbconn.query('INSERT INTO Product SET ?', record,
function(err,dbRes){
        if (err) {
           dbconn.rollback(function() {
     throw err;
           });
        }

    const tmpSku = dbRes.insertId;
     record = {KEYWORD: req.body.image, SKU: tmpSku};

    dbconn.query('INSERT INTO PRODUCT_KEYWORD SET ?', record,
function(err,dbRes){
           if (err) {
      dbconn.rollback(function() {
        throw err;
      });
           }
     console.log('record insert into PRODUCT_KEYWORD table');
           dbconn.commit(function(err) {
            if (err) {
       dbconn.rollback(function() {
         throw err;
       });
            }
     console.log('inserted into both Product and PRODUCT_KEYWORD tables
in one transcation ');

     const result = {
      sku : tmpSku,
        name : req.body.name,
      description : req.body.description,
      length : req.body.length,
      width : req.body.width,
      height : req.body.height,
      weight : req.body.weight,
      featured : req.body.featured,
      availability : req.body.availability,
      price : req.body.price,
      image : req.body.image
      };

       httpRes.json(result);

           }); //end commit
        }); //end 2nd query
     }); //end 1st query
  });
  // End transaction

      dbconn.release();
 });//end pool.getConnection
});
```

To reduce inventory as part of the checkout process, another method is implemented:

```
//reduce product through post, this is for the checkout process
app.post('/product/reduce', function(req, httpRes) {
 "use strict";
 let sendReply = false;

 pool.getConnection(function(err, conn) {
  for (let i = 0; i < req.body.length; i++) {
   if(!req.body[i].hasOwnProperty('sku') ||
!req.body[i].hasOwnProperty('quantity')) {
    httpRes.statusCode = 400;
    return httpRes.send('Error 400: need to have valid sku and
quantity.');
   }

   const tmpSku = req.body[i]['sku'];
   const tmpQuantity = req.body[i]['quantity'];
   const sqlStr = 'update Product set availability = availability - ' +
tmpQuantity + ' where sku = ' + tmpSku + ' and availability - ' +
tmpQuantity + ' > 0';
   console.log('reduce tmpSku:' + tmpSku);
   console.log('reduce tmpQuantity:' + tmpQuantity);
   console.log('reduce sqlStr: ' + sqlStr);

   conn.query(sqlStr, function(err, result) {
    if(err) throw err;

      if (result.affectedRows > 0) {
     console.log('reduced from Product ' + result.affectedRows + '
rows');
      } else {
     const result = [
       { message : 'Insufficient availability for ' + tmpSku,
       details : null}
     ];
     if (sendReply == false) {
      httpRes.json(result);
      sendReply = true;
     }

      }
   });


  }
```

```
      conn.release();
 });
 return httpRes.send('');;
});
```

To delete a product based on the provided product *SKU*, a method is implemented to respond to *HTTP DELETE* requests. This method also interacts with two different tables and therefore requires a transaction:

```
//delete based on sku #
app.delete('/product/products/:sku', function(req, httpRes) {

 pool.getConnection(function(err, dbconn) {

  // Begin transaction
  dbconn.beginTransaction(function(err) {
     if (err) { throw err; }
     dbconn.query('DELETE FROM PRODUCT_KEYWORD where SKU = ?',
req.params.sku, function(err, result){
        if (err) {
           dbconn.rollback(function() {
     throw err;
           });
        }
   console.log('deleted from PRODUCT_KEYWORD ' + result.affectedRows +
' rows');

   dbconn.query('DELETE FROM Product where SKU = ?', req.params.sku,
function(err, result){
          if (err) {
     dbconn.rollback(function() {
       throw err;
     });
          }
    console.log('deleted from Product ' + result.affectedRows + '
rows');
          dbconn.commit(function(err) {
           if (err) {
      dbconn.rollback(function() {
        throw err;
      });
           }
     console.log('Transaction Complete.');
       httpRes.json('deleted from both Product and PRODUCT_KEYWORD
tables in one transcation');

          }); //end commit
       }); //end 2nd query
    }); //end 1st query
  });
  // End transaction

      dbconn.release();
 });//end pool.getConnection
});
```

Finally, an update method is provided for both full updates through *HTTP PUT* and partial updates through *HTTP PATCH*, based on the *SKU* number. Both methods delegate to a single implementation.

```
//put (update) based on sku #
app.put('/product/products/:sku', function(req, res) {
 updateProduct(req.params.sku, req, res);
});

//patch (update) based on sku #
app.patch('/product/products/:sku', function(req, res) {
 updateProduct(req.params.sku, req, res);
});


//real update function works for both put and patch request
function updateProduct(skuIn, req, httpRes) {
 "use strict";
 let sqlStr = '';
 if (req.body.DESCRIPTION != null){
  sqlStr = 'DESCRIPTION = \'' + req.body.DESCRIPTION + "\'";
 }
 if (req.body.HEIGHT != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'HEIGHT = \'' + req.body.HEIGHT + "\'";
 }
 if (req.body.LENGTH != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'LENGTH = \'' + req.body.LENGTH + "\'";
 }
 if (req.body.NAME != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'NAME = \'' + req.body.NAME + "\'";
 }
 if (req.body.WEIGHT != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'WEIGHT = \'' + req.body.WEIGHT + "\'";
 }
 if (req.body.WIDTH != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'WIDTH = \'' + req.body.WIDTH + "\'";
 }
 if (req.body.FEATURED != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'FEATURED = \'' + req.body.FEATURED + "\'";
 }
 if (req.body.AVAILABILITY != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'AVAILABILITY = \'' + req.body.AVAILABILITY + "\'";
 }
 if (req.body.IMAGE != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'IMAGE = \'' + req.body.IMAGE + "\'";
 }
 if (req.body.PRICE != null){
  if (sqlStr !='') sqlStr = sqlStr + " , ";
  sqlStr = sqlStr + 'PRICE = \'' + req.body.PRICE + "\'";
 }
 sqlStr = 'UPDATE Product SET ' + sqlStr + ' WHERE SKU = ?';
     console.log('!!!!!SQL ready to be executed: ' + sqlStr);


 pool.getConnection(function(err, conn) {
  conn.query(sqlStr, skuIn, function(err, result) {
   if(err) throw err;
   console.log('update Product table' + result.affectedRows + ' rows');
  });
      conn.release();
 });

   httpRes.json('Update Product table');
}
```

CHAPTER 5. CONCLUSION

Linux containers provide numerous advantages and are a natural fit for microservices. Red Hat OpenShift helps govern and simplify various stages of the software lifecycle by leveraging Kubernetes to manage a cluster of containers and expanding on the capabilities of container technology.

This reference architecture demonstrates a heterogeneous environment where Java EE, Ruby and Node.js services interact in a microservice architecture to provide an Red Hat OpenShift application that can be deployed and distributed in a secure environment and on-premise cloud, while benefiting from enterprise support and quality of service.

For a discussion of planning, deployment and operation of an Open Source Platform as a Service, refer to the reference architecture on OpenShift Enterprise 3 Architecture Guide.

APPENDIX A. REVISION HISTORY

Revision	Release Date	Author(s)
1.1	March 2017	Calvin Zhu
1.0	January 2017	Calvin Zhu

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 & Consulting Engineer	Technical Content Review
Christoph Goern	Principal Software Engineer	Technical Content Review
Scott Collier	Manager, Software Engineering & Consulting Engineer	Technical Content Review
Mark Little	Vice President, Software Engineering	Content Review
Rich Sharples	Senior Director, Product Management	Content Review
Lance Ball	Senior Software Engineer	Node.js Code Review

APPENDIX C. REVISION HISTORY

Revision 1.1-0	**March 2017**	**CZ**