Reference Architectures 2017

Enterprise Mobile Applications

Development on Red Hat Mobile Application Platform

Last Updated: 2017-10-24

Reference Architectures 2017 Enterprise Mobile Applications

Development on Red Hat Mobile Application Platform

Jeremy Ary

Babak Mozaffari
refarch-feedback@redhat.com

Legal Notice

Abstract

This reference architecture demonstrates the design, development and deployment of enterprise mobile applications with Red Hat Mobile Application Platform.

Table of Contents

COMMENTS AND FEEDBACK **4**

CHAPTER 1. EXECUTIVE SUMMARY **5**

CHAPTER 2. RED HAT MOBILE APPLICATION PLATFORM **6**
2.1. OVERVIEW 6
2.2. STUDIO 6
2.3. CLIENT APPLICATIONS 6
2.4. CLOUD APPLICATIONS 6
2.5. MOBILE BACKEND-AS-A-SERVICE 6

CHAPTER 3. REFERENCE ARCHITECTURE ENVIRONMENT **7**

CHAPTER 4. CREATING THE ENVIRONMENT **9**
4.1. INSTALLATION 9
4.1.1. Prerequisites 9
4.1.1.1. Overview 9
4.1.1.2. Sizing 9
4.1.1.3. Infrastructure 9
4.1.1.4. Shared Storage 11
4.1.1.5. OpenShift Configuration 11
4.1.1.5.1. Wildcard Certificate 12
4.1.2. Red Hat Mobile Application Platform 12
4.1.2.1. Preparations 12
4.1.2.2. Core Installation 13
4.1.2.2.1. Persistent Volumes 13
4.1.2.2.2. Node Labels 14
4.1.2.2.3. Configuration 14
4.1.2.2.4. Provisioning 14
4.1.2.2.5. Verification 15
4.1.2.3. MBaaS Installation 16
4.1.2.3.1. Persistent Volumes 16
4.1.2.3.2. Node Labels 17
4.1.2.3.3. Configuration 17
4.1.2.3.4. Provisioning 17
4.1.2.3.5. Verification 17
4.1.2.4. Creating an Environment 19
4.2. DEPLOYMENT 22
4.2.1. E-Commerce Services OpenShift Project 22
4.2.1.1. Create Project 22
4.2.1.2. Template Population 22
4.2.1.3. Persistence Data Population 23
4.2.2. E-Commerce Mobile App Platform Project 23
4.2.2.1. Prerequisites 23
4.2.2.2. Parent Project 23
4.2.2.3. MBaaS Health Monitor 24
4.2.2.4. Cloud Application 29
4.2.2.5. Mobile Client Application 30
4.2.2.6. Web Portal Client Application 33

CHAPTER 5. DESIGN AND DEVELOPMENT **34**
5.1. OVERVIEW 34
5.2. HEALTH MONITOR MBAAS SERVICE 34

5.2.1. Template Modifications 34
5.2.2. Cloud Application Integration 36
5.2.3. Client Application Integration 36
5.3. CLOUD APPLICATION 37
5.3.1. Template Scaffolding 37
5.3.2. SDK Configuration 37
5.3.3. cloud.js 37
5.3.4. Cloud Activity Logging 38
5.3.5. Microservice MBaaS Service Call Proxying 39
5.4. CLIENT APPLICATION REQUIREMENTS 39
5.5. CLIENT PLATFORM INTEGRATIONS 40
5.5.1. Data Synchronization 40
5.5.2. Data Caching 41
5.5.3. Form Builder 43
5.5.4. Cloud Storage 45
5.5.5. Build Farms 45
5.6. MOBILE CLIENT APPLICATION 45
5.6.1. Cordova 46
5.6.2. Ionic 46
5.6.3. Template Scaffolding 46
5.6.4. SDK Configuration 46
5.6.5. Cloud Communication 46
5.6.6. Running Locally 47
5.7. WEB PORTAL CLIENT APPLICATION 48
5.7.1. SDK Configuration 49
5.7.1.1. application.js 49
5.7.1.2. fhconfig.js 49
5.7.1.3. Cloud Communication 49
5.7.1.4. Running Locally 49
5.7.1.5. Bootstrap UI Components 49

CHAPTER 6. CONCLUSION **51**

APPENDIX A. AUTHORSHIP HISTORY **52**

APPENDIX B. CONTRIBUTORS **53**

APPENDIX C. ANSIBLE INVENTORY **54**

APPENDIX D. REVISION HISTORY **56**

COMMENTS AND FEEDBACK

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

CHAPTER 1. EXECUTIVE SUMMARY

This reference architecture demonstrates the design, development and deployment of enterprise mobile applications with **`Red Hat Mobile Application Platform`**. It builds on previously published work, and leverages the mobile application platform to add a mobile client to an existing application, while relying on features provided by the platform to ensure security, optimize performance, and accelerate development.

The reference architecture paper documents the steps undertaken to utilize some of the prominent capabilities provided by Red Hat Mobile Application Platform, while the accompanying code and configuration allows the reader to fully understand each step and replicate them at will.

CHAPTER 2. RED HAT MOBILE APPLICATION PLATFORM

2.1. OVERVIEW

Red Hat Mobile Application Platform speeds up the development, integration, deployment, and management of enterprise mobile applications for businesses. The platform offers a suite of features that embrace collaborative app development, centralized control of security and back-end integration, and deployment in cloud, on-premise, and hybrid infrastructures. Enterprise mobile developers and DevOps teams can build mobile apps and services that can be used and reused across multiple organization-wide mobile projects. Developers can use the client-side tools of their choice while still accessing platform functionality for integration, collaboration, deployment and management via platform APIs and SDKs.

2.2. STUDIO

Studio, the web-based interface of the platform, allows for creation, deployment, monitoring, and life-cycle management of all mobile, cloud, and Mobile Backend-as-a-Service (MBaaS) portions of a project. With a variety of starter templates, fully-integrated editor and build interface, the Studio offers simplified launching or integration of mobile projects, featuring enterprise-level capabilities and infrastructure, that seamlessly transition into more complex, large-scale solutions as applications mature.

2.3. CLIENT APPLICATIONS

Mobile, web, and cross-platform client applications can be built with relative ease via usage of native and hybrid mobile SDKs available for iOS, Android, Windows, Xamarin, Cordova, and other HTML5 frameworks. Integrated life-cycle process and build support allow for simplified, consistent artifact creation and distribution.

2.4. CLOUD APPLICATIONS

Cloud applications, written in Node.js, serve as both server-side endpoint and API resource for client applications which interact via RHMAP SDKs. Cloud API options, such as security, session management, data caching and synchronization, to name a few, are integrated at this level.

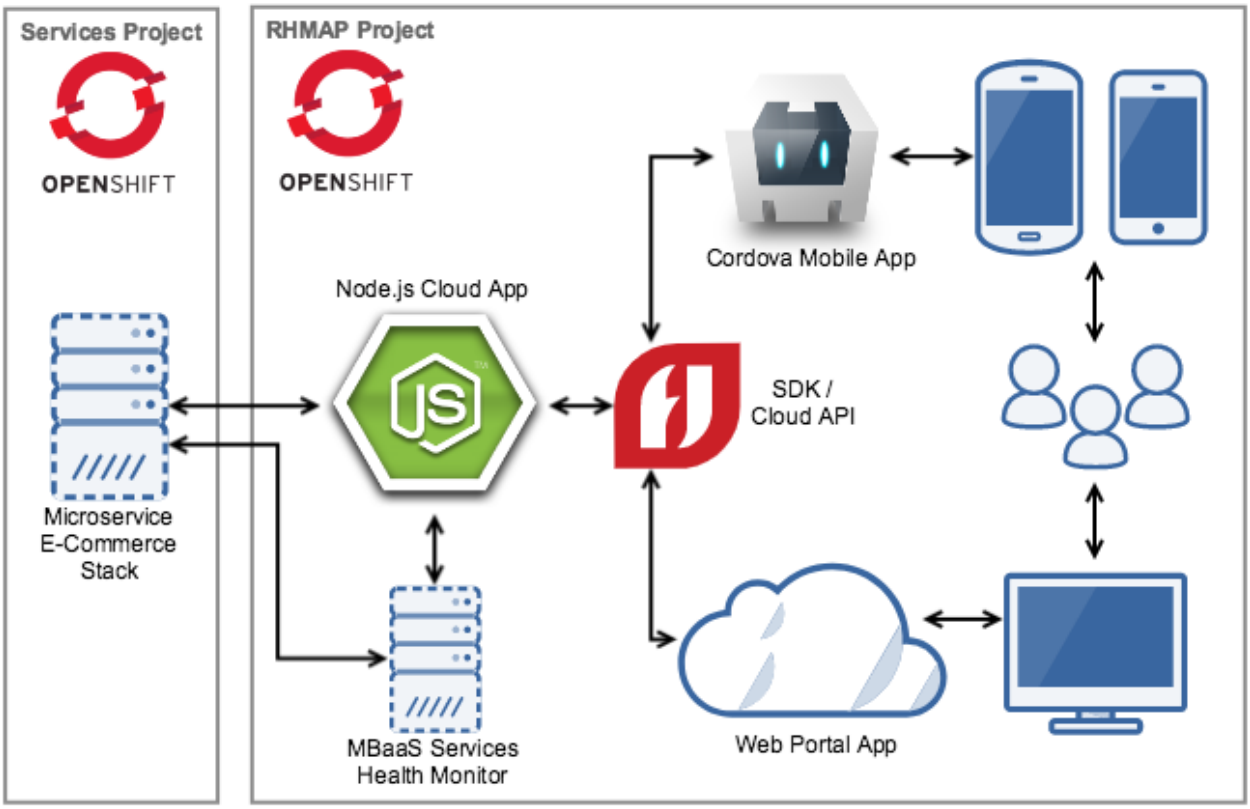
2.5. MOBILE BACKEND-AS-A-SERVICE

Mobile Backend-as-a-Service (MBaaS) hosts a set of services, created from template or scratch, which support and extend cloud application functionality across one or more projects. Examples include infrastructure management tools, security implementations, shared data sources, integration of popular third-party solutions, and more.

CHAPTER 3. REFERENCE ARCHITECTURE ENVIRONMENT

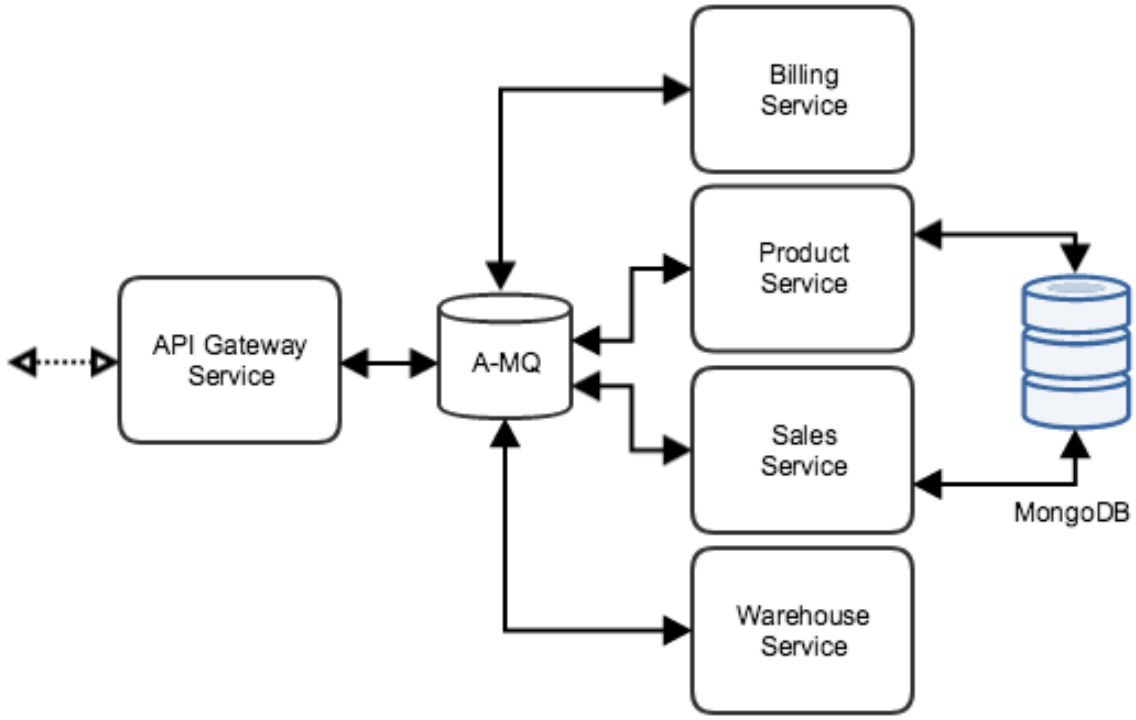
This reference architecture demonstrates an e-commerce system featuring a cross-platform mobile client, web portal, cloud application, MBaaS service, and API connectivity to external microservices hosted as a separate OpenShift Container Platform project. This back-end external microservice architecture, which encompasses representations of the various departmental and back-end functionalities of an e-commerce implementation, was developed as part of a previous project demonstrating Fuse Integration Services on OpenShift Container Platform. The software stack developed herein showcases Red Hat Mobile Application Platform's ability to provide enterprise level client applications and feature sets, essential to large-scale distribution and usability. The overall design of the system is as follows:

Figure 3.1. Project Overview



High-level reference of the "E-Commerce Services" microservices architecture:

Figure 3.2. Services Stack



CHAPTER 4. CREATING THE ENVIRONMENT

4.1. INSTALLATION

4.1.1. Prerequisites

4.1.1.1. Overview

This reference architecture can be deployed in either a production or a trial environment. In both cases, it is assumed that *rhmap-master1* refers to one (or the only) OpenShift master host and that the environment includes six OpenShift schedulable hosts with the host names of *rhmap-node1* to *rhmap-node6*. Production environments would have at least 3 master hosts to provide **High Availability (HA)** resource management.

It is further assumed that OpenShift Container Platform 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*.

4.1.1.2. Sizing

Red Hat provides a sizing tool for Red Hat Mobile Application Platform. For the purpose of installing and configuring an environment, this reference application is assumed to be a *Business to Employee (B2E)* internal application, deployed on an on-premise platform that includes both the *core* and *MBaaS* components. We further assume that it qualifies as a single application for the use of up to 200 employees. Based on these parameters, the sizing tool suggests:

For your B2E subscription for 1 apps and Up to 200 employees, we recommend 3 nodes for your OpenShift master and 0 nodes for your OpenShift infrastructure, 0 nodes for your RHMAP MBaaS and 3 App nodes.

Each VM for those nodes must a minimum configuration of:
vCPUS: 2
RAM(GB): 8
Storage(GB): 16

For full RHMAP installation the Core MAP component for B2E subscription for 1 apps and Up to 200 employees, we recommend 3 nodes.

Each VM for those Core MAP nodes must have a minimum configuration of:
vCPUS: 4
RAM(GB): 8
Storage(GB): 100

4.1.1.3. Infrastructure

Set up six virtual or physical machines to host the nodes of the OpenShift cluster, as suggested by the results of the sizing tool.

In the reference architecture environment, *rhmap-node1*, *rhmap-node2*, and *rhmap-node3* are used for MBaaS. The next three nodes, *rhmap-node4*, *rhmap-node5*, and *rhmap-node6* deploy the core components.

Use *root* or another user with the required privileges to create a *Linux* user that will run the *Ansible* provisioning scripts. For example, to create the user, assign a password, and give it *superuser* privileges:

```
# useradd rhmapAdmin
# passwd rhmapAdmin
# usermod -aG wheel rhmapAdmin
```

Allow members of the *wheel* group to execute privileged actions without typing the password.

```
# visudo
```

Uncomment the line that specifies a *NOPASSWD* policy:

%wheel ALL=(ALL) NOPASSWD: ALL

Repeat this process on all masters and nodes.

Next, generate *SSH* keys for the new user. The keys will be used to allow the user to access all the nodes from the *master*, without having to provide a password. Press enter to accept default values or provide empty values for the prompts:

```
# su - rhmapAdmin
$ ssh-keygen

Generating public/private rsa key pair.
Enter file in which to save the key (/home/rhmapAdmin/.ssh/id_rsa):
Created directory '/home/rhmapAdmin/.ssh'.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:
Your identification has been saved in /home/rhmapAdmin/.ssh/id_rsa.
Your public key has been saved in /home/rhmapAdmin/.ssh/id_rsa.pub.

The key fingerprint is:
d0:d2:d9:ed:31:63:28:d0:c0:4c:91:8b:34:d5:03:b8 rhmapAdmin@rhmap-
master1.xxx.example.com
The key's randomart image is:
+--[ RSA 2048]----+
|     *BB         |
|    + +++o o     |
|   . +o.=.o *    |
|    E .o . o +   |
|        S   .    |
|                 |
|                 |
|                 |
|                 |
+-----------------+
```

Now that the keys are generated, copy the public key to all machines to allow the holder of the private key to log in:

```
$ for host in rhmap-master1 rhmap-node1 rhmap-node2 rhmap-node3 rhmap-
```

```
node4 rhmap-node5 rhmap-node6; \
    do ssh-copy-id -i ~/.ssh/id_rsa.pub $host; \
    done
```

4.1.1.4. Shared Storage

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

Attach 66GB of storage for core and 151GB for MBaaS. Create a volume group for each of them, and logical volumes for each required persistent volume:

```
$ sudo pvcreate /dev/vdc
$ sudo vgcreate rhmapcore /dev/vdc
$ sudo lvcreate -L 5G -n gitlabshell rhmapcore
$ sudo lvcreate -L 5G -n mysql rhmapcore
$ sudo lvcreate -L 5G -n metricslog rhmapcore
$ sudo lvcreate -L 25G -n mongodb rhmapcore
$ sudo lvcreate -l 25G -n fhscm rhmapcore
$ sudo lvcreate -L 1G -n nagios rhmapcore

$ sudo pvcreate /dev/vdd
$ sudo vgcreate mbaas /dev/vdd
$ sudo lvcreate -L 50G -n mongo1 mbaas
$ sudo lvcreate -L 50G -n mongo2 mbaas
$ sudo lvcreate -l 50G -n mongo3 mbaas
$ sudo lvcreate -L 1G -n nagios mbaas
```

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

```
$ sudo systemctl stop nfs
$ for i in rhmapcore/nagios rhmapcore/gitlabshell rhmapcore/mysql
rhmapcore/metricslog rhmapcore/mongodb rhmapcore/fhscm mbaas/nagios
mbaas/mongo1 mbaas/mongo2 mbaas/mongo3;
> do
> sudo mkfs.ext4 /dev/$i;
> sudo mkdir -p /mnt/$i;
> sudo mount /dev/$i /mnt/$i;
> done
```

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

4.1.1.5. OpenShift Configuration

Create an OpenShift user, optionally with the same name, to use for the provisioning. Assuming the use of HTPasswd as the authentication provider:

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

Grant OpenShift admin and cluster admin roles to this user:

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

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

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

Login successful.
```

4.1.1.5.1. Wildcard Certificate

Simple installations of Red Hat OpenShift Container Platform 3 typically include a default HAProxy router service. Use a commercially signed wildcard certificate, or generate a self-signed certificate, and configure a new router to use it.

To generate a self-signed certificate using the certificate included in the OpenShift Container Platform installation:

```
$ CA=/etc/origin/master
$ sudo oadm ca create-server-cert --signer-cert=$CA/ca.crt \
      --signer-key=$CA/ca.key --signer-serial=$CA/ca.serial.txt \
      --hostnames='.rhmap.xxx.example.com' \*
      --cert=cloudapps.crt --key=cloudapps.key
$ sudo cat cloudapps.crt cloudapps.key $CA/ca.crt > cloudapps.router.pem
```

Delete the previously configured router:

```
$ oc delete all -l router=router -n default
$ oc delete secret router-certs -n default
```

Recreate the HAProxy router with the newly generated certificate:

```
$ sudo oadm router --default-cert=cloudapps.router.pem --service-
account=router
```

4.1.2. Red Hat Mobile Application Platform

4.1.2.1. Preparations

Configure and run the provisioning steps for Red Hat Mobile Application Platform from the *rhmap-master1* machine.

Enable the *yum* repository for Red Hat Mobile Application Platform 4.4:

```
$ sudo subscription-manager repos --enable="rhel-7-server-rhmap-4.4-rpms"
```

Install the OpenShift templates for Red Hat Mobile Application Platform, as well as the *Ansible* playbooks and configuration files through its package:

```
$ sudo yum install rhmap-fh-openshift-templates
```

Configure an Ansible inventory file for the reference architecture environment. Several templates are provided as part of the package installed in the previous step. For this environment, use */opt/rhmap/4.4/rhmap-installer/inventories/templates/multi-node-example* as a starting point. The inventory file used for this environment is provided as an appendix to this document.

The *cluster_hostname* variable is set to the OpenShift cluster subdomain. The *domain_name* value is a name that is used to build the address to the studio web console.

The specified user for the *ansible_ssh_user* variable is used to log in to the required machines during installation. The OpenShift user and password are also specified in this file.

To run the provided ansible books, change to the */opt/rhmap/4.4/rhmap-installer/* directory. The *roles* directory is expected in the current directory when running the playbooks. Save the modified host inventory file detailed in the appendix in this directory as *rhmap_hosts*.

Download the required *docker* images to make the platform setup process faster, and avoid timeouts. Run the two following playbooks to download the images:

```
$ ansible-playbook -i rhmap-hosts playbooks/seed-images.yml -e
"project_type=core" -e "rhmap_version=4.4"
$ ansible-playbook -i rhmap-hosts playbooks/seed-images.yml -e
"project_type=mbaas" -e "rhmap_version=4.4"
```

4.1.2.2. Core Installation

4.1.2.2.1. Persistent Volumes

Create an OpenShift persistent volume for each of the mounts designated for the core component. This can be done by first creating a *yaml* or *json* file for each volume, for example:

```
{
  "kind": "PersistentVolume",
  "apiVersion": "v1",
  "metadata": {
    "name": "git-data"
  },
  "spec": {
    "capacity": {
      "storage": "5Gi"
    },
    "accessModes": [
      "ReadWriteOnce"
    ],
    "persistentVolumeReclaimPolicy": "Recycle",
    "nfs": {
      "path": "/mnt/rhmapcore/gitlabshell",
      "server": "10.19.137.71"
    }
  }
}
```

Create a similar file for each logical volume, with the corresponding storage capacity and mount address. The name is arbitrary and the association between the claiming component and the persistent volume is not enforced.

Once created, to create the persistent volume based on a file, simply use the *create* command of the *oc* utility:

```
$ oc create -f git-data-pv.json
```

4.1.2.2.2. Node Labels

Label the nodes intended for provision the *core* components accordingly. These labels can be used by OpenShift commands to filter out the nodes applicable for a given operation.

```
$ for i in {4..6}; do oc label node rhmap-node$i.xxx.example.com
type=core; done;
```

4.1.2.2.3. Configuration

Configure the Red Hat Mobile Application Platform core platform by editing the Ansible file. This file is located at */opt/rhmap/4.4/rhmap-installer/roles/deploy-core/defaults/main.yml*.

When using a self-signed certificate, this file must be edited to use *http* as the git protocol.

git_external_protocol: "http"

Also note that the *Build Farm Configuration* has sample values that do not function. To determine the values for the *builder_android_serviced_host* and *builder_iphone_serviced_host* variables, contact Red Hat Support asking for the RHMAP Build Farm URLs that are appropriate for your region.

builder_android_service_host: "https://androidbuild.feedhenry.com"
builder_iphone_service_host: "https://iosbuild.feedhenry.com"

4.1.2.2.4. Provisioning

With the configuration in place, simply run the playbook to provision the Red Hat Mobile Application Platform core platform:

```
$ ansible-playbook -i ~/setup/rhmap-hosts playbooks/core.yml
```

NOTE

The provisioning script tries to verify the router certificate and match the subject name against the provided cluster name. The *grep* command may not function as expected in some shell environments. The user is given the option to disregard a failed check and continue the installation.

NOTE

The Ansible provisioning script creates a test project with a *mysql* pod to verify that the persistent volume can be used. This project is subsequently deleted and the persistent volume is recycled and made available for the actual pod. However, the recycling of the persistent volumes can take time and if it does not happen fast enough, the pods will have one fewer persistent volumes than expected and the one that does not bind may end up requesting a higher capacity than the previously busy volume is able to provide. The script already includes a 20 second pause in */opt/rhmap/4.4/rhmap-installer/roles/setup/vars/core.yml* but this duration may not be sufficient for some environments, in which case it can be increased in the file.

4.1.2.2.5. Verification

Verify that the RHMAP core has been provisioned successfully, by first listing the pods in the project and ensuring that they are all in the running state:

```
$ oc get pods -n rhmap-core

NAME                    READY     STATUS    RESTARTS   AGE
fh-aaa-1-35cm3          1/1       Running   0          1m
fh-appstore-1-94t68     1/1       Running   0          1m
fh-messaging-1-jtxb5    1/1       Running   0          1m
fh-metrics-1-qvmbp      1/1       Running   0          1m
fh-ngui-1-gkznl         1/1       Running   0          1m
fh-scm-1-hth5b          1/1       Running   0          1m
fh-supercore-1-sl81q    1/1       Running   0          1m
gitlab-shell-1-q90xt    2/2       Running   1          1m
memcached-1-pbsc6       1/1       Running   0          1m
millicore-1-hccdv       3/3       Running   0          1m
mongodb-1-1-c9x66       1/1       Running   0          1m
mysql-1-kv8n4           1/1       Running   0          1m
nagios-1-38pv1          1/1       Running   0          1m
redis-1-8gtcp           1/1       Running   0          1m
ups-1-3t6js             1/1       Running   0          1m
```

For more confidence that all core services are running probably, use the included **`Nagios`** console. Find the URL to the nagios console by querying the created OpenShift route:

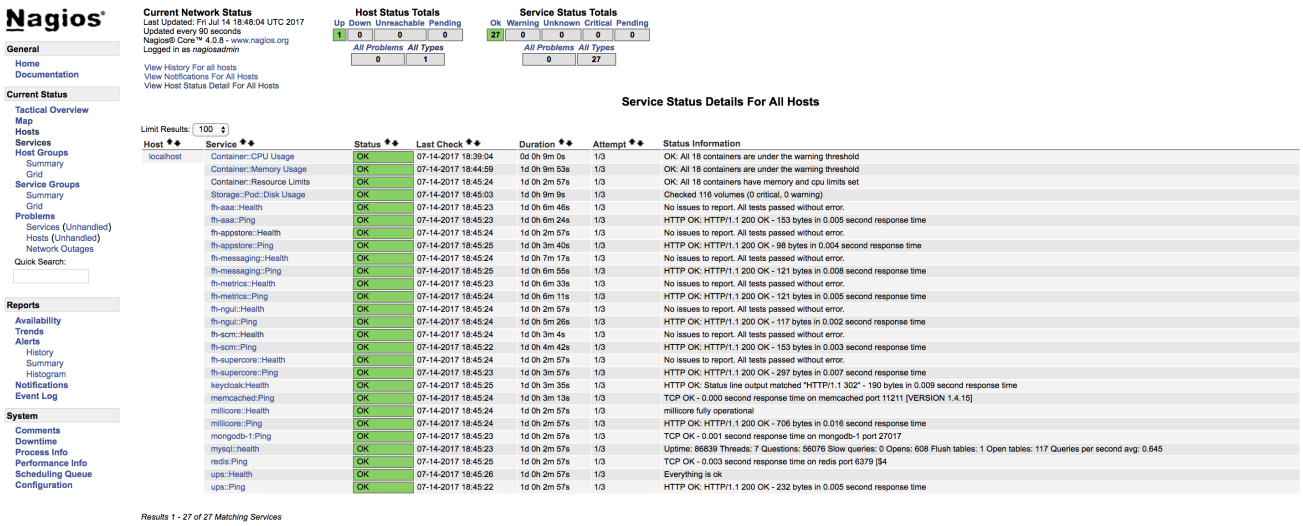
```
$ oc get route nagios -n rhmap-core

NAME      HOST/PORT
PATH      SERVICES   PORT      TERMINATION   WILDCARD
nagios    nagios-rhmap-core.rhmap.xxx.example.com            nagios
<all>     edge/Allow    None
```

Point your browser to this address. The service using basic authentication and while the username for accessing nagios is statically set to *nagiosadmin*, the password is generated by OpenShift during deployment. Query this password from the pod environment:

```
$ oc env dc nagios --list -n rhmap-core | grep NAGIOS

NAGIOS_USER=nagiosadmin
NAGIOS_PASSWORD=7Ppw0t2l78
```

Once logged in, use the left panel menu and click on services. Verify that the status of all services is green and OK.

Figure 4.1. Nagios admin console, RHMAP Core service health



4.1.2.3. MBaaS Installation

4.1.2.3.1. Persistent Volumes

Create an OpenShift persistent volume for each of the mounts designated for the *MBaaS* component. Again, create a *yaml* or *json* file for each volume, for example:

```
{
  "kind": "PersistentVolume",
  "apiVersion": "v1",
  "metadata": {
    "name": "mbaas-mongodb-claim-1"
  },
  "spec": {
    "capacity": {
      "storage": "50Gi"
    },
    "accessModes": [
      "ReadWriteOnce"
    ],
    "persistentVolumeReclaimPolicy": "Recycle",
    "nfs": {
      "path": "/mnt/mbaas/mongo1",
      "server": "10.19.137.71"
    }
  },
  "claimRef": {
    "namespace": "hosted-mbaas",
    "name": "mongodb-claim-1"
  }
}
```

Similar to before, to create the persistent volume based on a file, simply use the *create* command of the *oc* utility:

```
$ oc create -f mbaas-mongo1-pv.json
```

4.1.2.3.2. Node Labels

Label the nodes intended for provision the *MBaaS* components accordingly. These labels can be used by OpenShift commands to filter out the nodes applicable for a given operation. Also label the nodes sequentially as node #1, #2 and #3 for mbaas:

```
$ for i in {1..3}; do oc label node rhmap-node$i.xxx.example.com
type=mbaas; done;
$ for i in {1..3}; do oc label node rhmap-node$i.xxx.example.com
mbaas_id=mbaas$i; done;
```

4.1.2.3.3. Configuration

Configure the mbaas platform by editing the Ansible file. This file is located at */opt/rhmap/4.4/rhmap-installer/roles/deploy-mbaas/defaults/main.yml* and contains *SMTP* and similar configuration.

4.1.2.3.4. Provisioning

With the configuration in place, simply run the playbook to provision the Red Hat Mobile Application Platform mbaas platform:

```
$ ansible-playbook -i ~/setup/rhmap-hosts playbooks/3-node-mbaas.yml
```

Once again, refer to the notes above for some potential provisioning issues.

4.1.2.3.5. Verification

Verify that the RHMAP MBaaS components have been provisioned successfully, by first listing the pods in the project and ensuring that they are all in the running state, except for the *mongodb-initiator*, which only exists to initiate the mongo databases and completes its task after doing so:

```
$ oc get pods -n rhmap-3-node-mbaas

NAME                   READY     STATUS    RESTARTS   AGE
fh-mbaas-1-6f3vq       1/1       Running     4          1m
fh-mbaas-1-gn7b6       1/1       Running     5          1m
fh-mbaas-1-xcr05       1/1       Running     4          1m
fh-messaging-1-654nk   1/1       Running     4          1m
fh-messaging-1-7qtns   1/1       Running     4          1m
fh-messaging-1-805vc   1/1       Running     4          1m
fh-metrics-1-7g7gd     1/1       Running     4          1m
fh-metrics-1-pdz40     1/1       Running     5          1m
fh-metrics-1-w2zs5     1/1       Running     5          1m
fh-statsd-1-hhn6k      1/1       Running     0          1m
mongodb-1-1-cj059      1/1       Running     0          1m
mongodb-2-1-m1fg0      1/1       Running     0          1m
mongodb-3-1-d83dh      1/1       Running     0          1m
mongodb-initiator      0/1       Completed   0          1m
nagios-1-93vp1         1/1       Running     0          1m
```

You can also use nagios again. Find the URL to the nagios console by querying the created OpenShift route:

```
$ oc get route nagios -n rhmap-3-node-mbaas

NAME      HOST/PORT
PATH      SERVICES   PORT      TERMINATION   WILDCARD
nagios    nagios-rhmap-3-node-mbaas.rhmap.xxx.example.com
nagios     <all>     edge/Allow    None
```

The credentials, once again, are available from the pod environment:

```
$ oc env dc nagios --list -n rhmap-3-node-mbaas | grep NAGIOS

NAGIOS_USER=nagiosadmin
NAGIOS_PASSWORD=cYxMr1taIU
```

Similar to the nagios console for RHMAP core services, all 15 services should be in a green OK status.

The much better way of verifying the health of MBaaS services is through a *REST* call:

```
$ curl oc get route mbaas -n rhmap-3-node-mbaas --template "
{{.spec.host}}"/sys/info/health

{
 "status": "ok",
 "summary": "No issues to report. All tests passed without error.",
 "details": [
  {
   "description": "Check fh-statsd running",
   "test_status": "ok",
   "result": {
    "id": "fh-statsd",
    "status": "OK",
    "error": null
   },
   "runtime": 11
  },
  {
   "description": "Check fh-messaging running",
   "test_status": "ok",
   "result": {
    "id": "fh-messaging",
    "status": "OK",
    "error": null
   },
   "runtime": 24
  },
  {
   "description": "Check fh-metrics running",
   "test_status": "ok",
   "result": {
    "id": "fh-metrics",
    "status": "OK",
    "error": null
   },
   "runtime": 27
  },
```

```
  {
   "description": "Check Mongodb connection",
   "test_status": "ok",
   "result": {
    "id": "mongodb",
    "status": "OK",
    "error": null
   },
   "runtime": 407
  }
 ]
}
```

Verify that all 4 services return an OK status.

4.1.2.4. Creating an Environment

Before creating a project and starting with development, an environment and an MBaaS target are required. Discover the web address of the RHMAP studio by querying its associated OpenShift route:

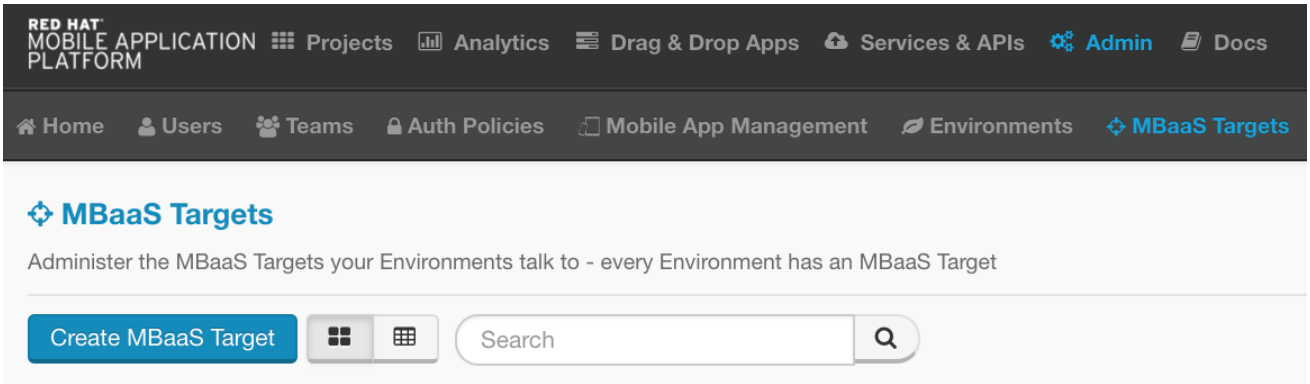
```
$ oc get route rhmap -n rhmap-core
NAME      HOST/PORT                                   PATH
SERVICES      PORT               TERMINATION   WILDCARD
rhmap     app.rhmap.xxx.example.com             rhmap-proxy   server-
http-port   edge/Allow    None
```

The password for the studio is generated by OpenShift during deployment. It can be retrieved from the *millicore* deployment config environment:

```
$ oc env dc millicore --list -n rhmap-core | grep FH_ADMIN
FH_ADMIN_USER_PASSWORD=3QlIf7RBqwmRa4yrH2rNYTKOSC
FH_ADMIN_USER_NAME=rhmap-admin@example.com
```

Use these credentials to log in to the studio. Once signed in, navigate through the *Admin* menu to *MBaaS Targets*:

Figure 4.2. MBaaS Targets



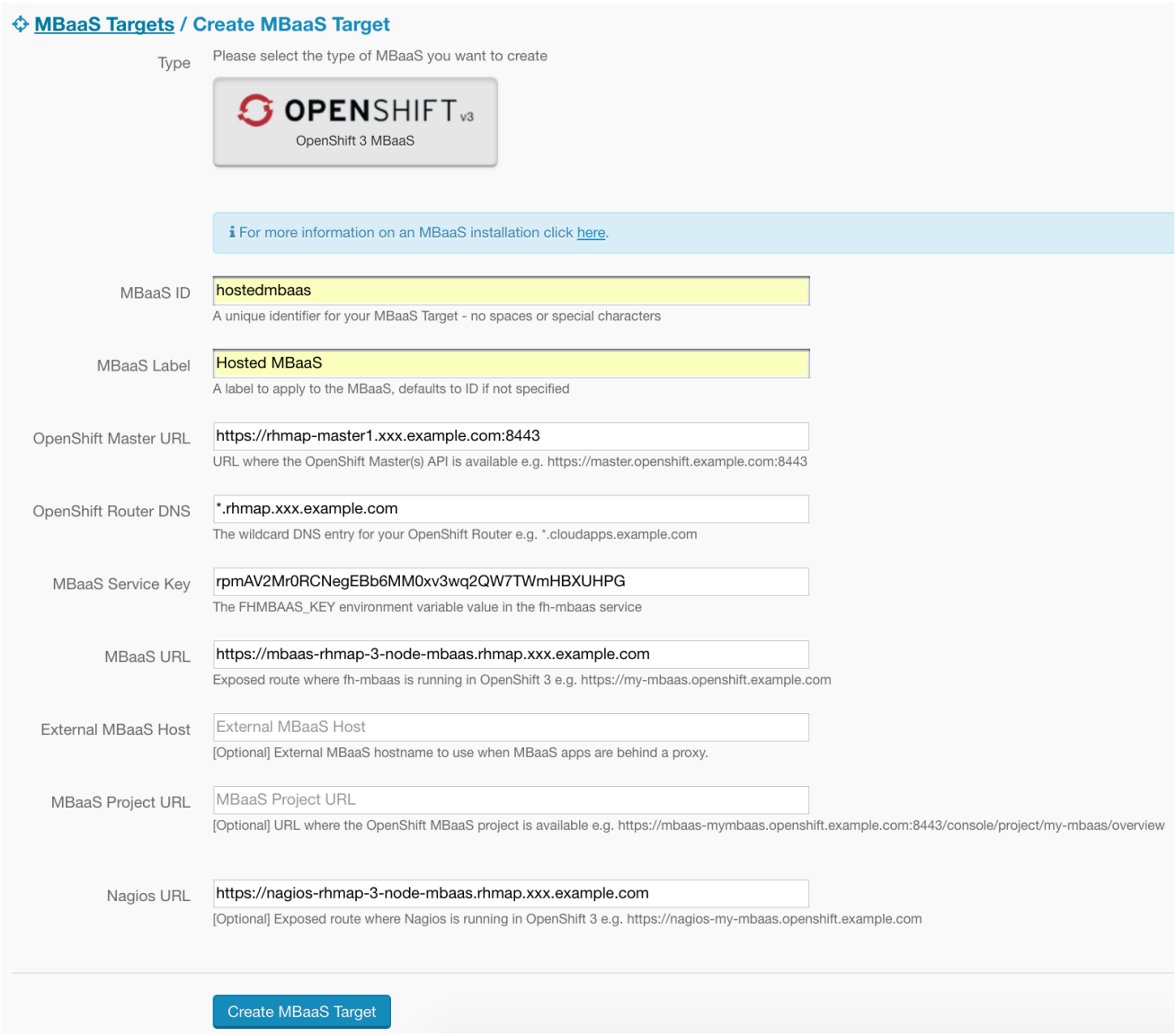
Click the button to create an MBaaS target. This requires providing a number of values, some of which are discretionary names and labels, but others are generated by OpenShift during deployment and should be looked up.

Find the values for *MBaaS Service Key*, *MBaaS URL*, and *Nagios URL*:

```
$ oc env dc fh-mbaas --list -n rhmap-3-node-mbaas | grep FHMBAAS_KEY
FHMBAAS_KEY=rpmAV2Mr0RCNegEBb6MM0xv3wq2QW7TWmHBXUHPG

$ echo "https://"$(oc get route/mbaas -n rhmap-3-node-mbaas -o template --
template {{.spec.host}})
https://mbaas-rhmap-3-node-mbaas.rhmap.xxx.example.com

$ echo "https://"$(oc get route/nagios -n rhmap-3-node-mbaas -o template -
-template {{.spec.host}})
https://nagios-rhmap-3-node-mbaas.rhmap.xxx.example.com
```

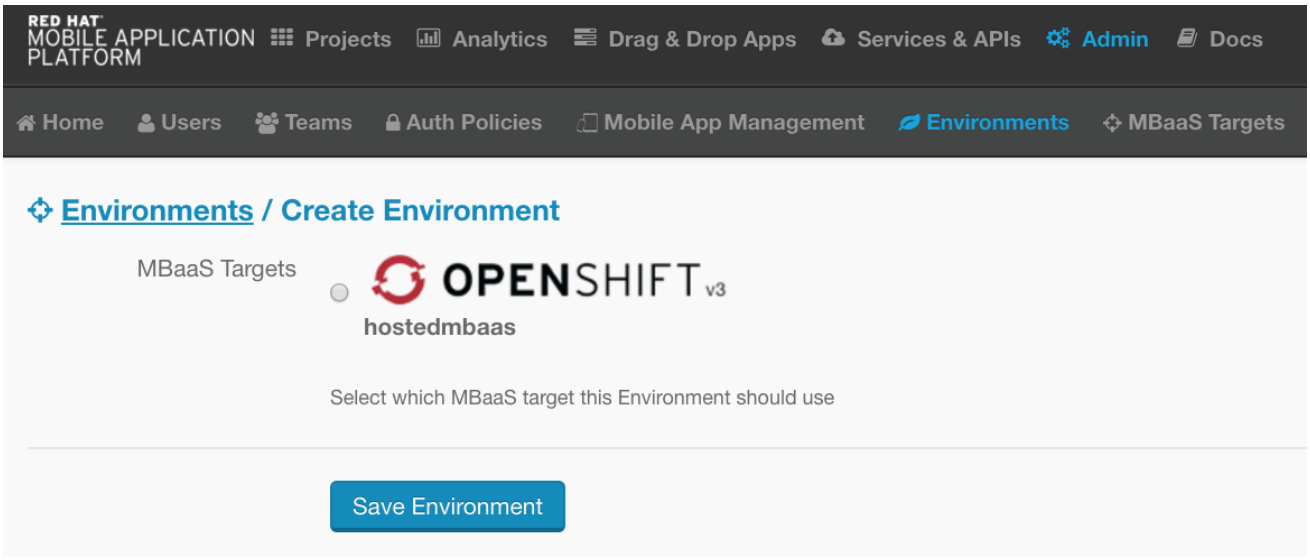
Use the above, and assign other valid values, to fill out the form to create a new MBaaS target:

Figure 4.3. Create MBaaS Target



Once an MBaaS target is successfully created, proceed to create an environment that references it:

Figure 4.4. Environments

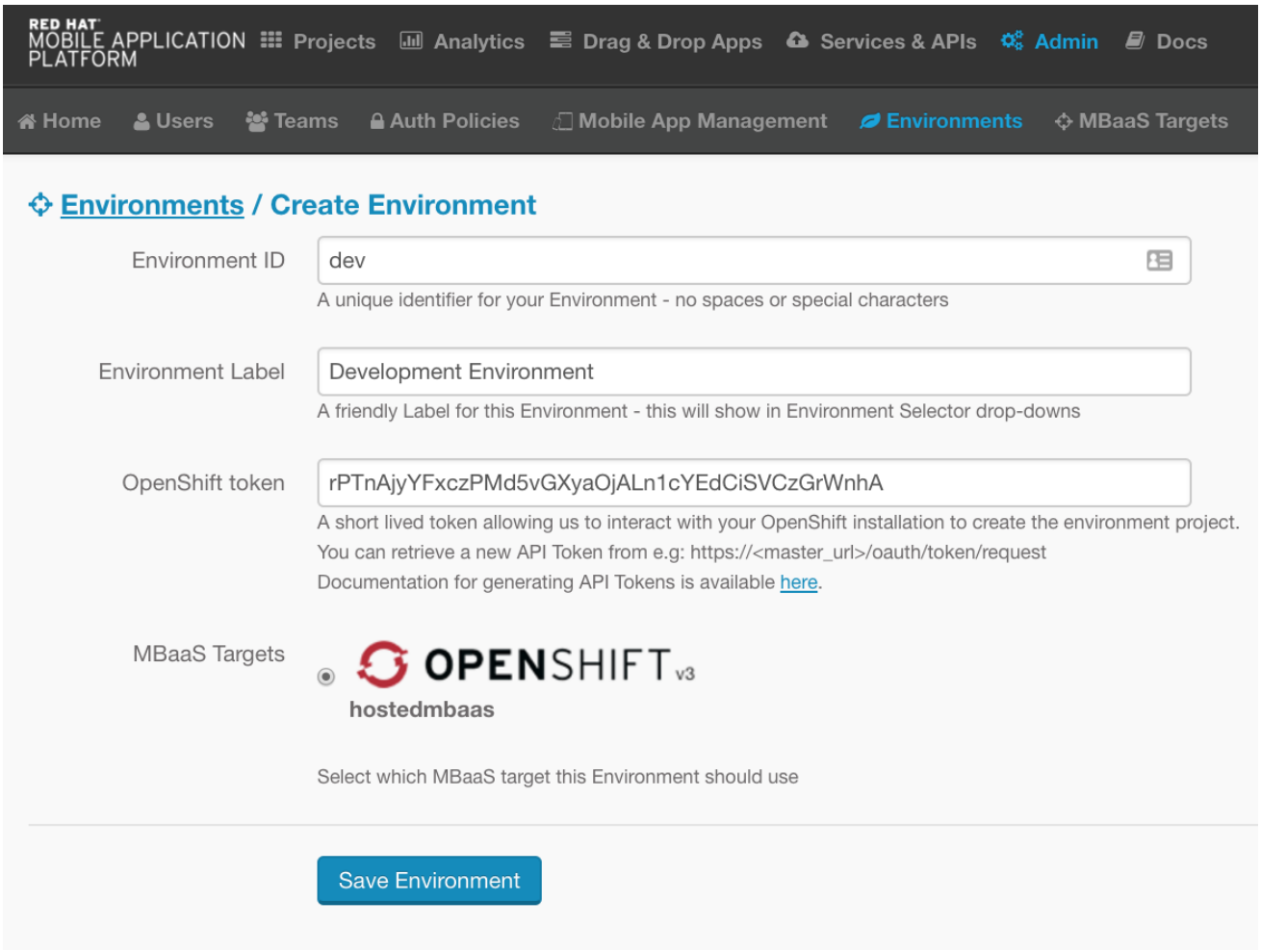


Each environment is associated with an OpenShift project, that must be created through a user's credentials. While logged in to OpenShift as *rhmapAdmin*, use the following command to discover and note the user's security token:

```
$ oc whoami -t
rPTnAjyYFxczPMd5vGXyaOjALn1cYEdCiSVCzGrWnhA
```

Use this token to create the environment:

Figure 4.5. Create Environment



Verify that an OpenShift environment has been created for this environment:

```
$ oc get projects
NAME                 DISPLAY NAME
STATUS
default
Active
kube-system
Active
logging
Active
management-infra
Active
openshift
Active
openshift-infra
Active
rhmap-3-node-mbaas
Active
rhmap-app-dev        RHMAP Environment: app-Development Environment
Active
rhmap-core
Active
```

4.2. DEPLOYMENT

4.2.1. E-Commerce Services OpenShift Project

4.2.1.1. Create Project

To start, a suite of back-end logic microservices will be installed in a secondary OpenShift Container Platform project native to the cluster housing the Mobile Application Platform installation.

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

```
$ oc login -u ocuser
```

Create the new project which will house the microservices suite:

```
$ oc new-project ecom-services --display-name="E-Commerce Services Suite"
--description="Back-end logic microservice suite"
```

4.2.1.2. Template Population

Within the new project, execute the provided YAML template to configure and instantiate the full services suite:

```
$ oc new-app -f https://raw.githubusercontent.com/RHsyseng/FIS2-MSA/ecom-
svcs/project-template.yaml
```

NOTE

Nothing further is required of the user following execution of the template to complete installation. As a whole, necessary builds and deployments can take from 10-20 minutes to complete.

4.2.1.3. Persistence Data Population

Once all services are deployed and running, instantiate the e-commerce data set via a GET request to the gateway service route:

```
$ curl -i http://ecom.rhmap.xxx.example.com/demo/testApi
HTTP/1.1 200 OK
...
```

4.2.2. E-Commerce Mobile App Platform Project

4.2.2.1. Prerequisites

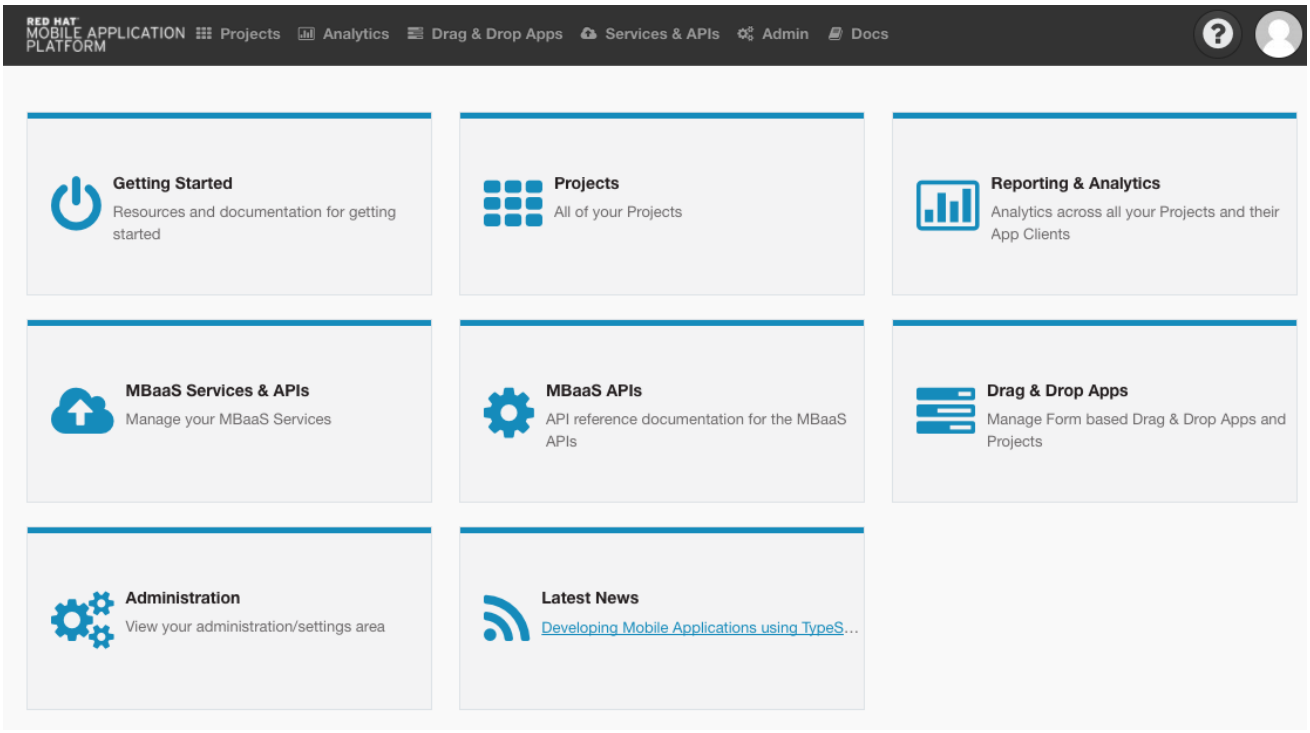
Before beginning, download the following artifact. The archive contains a series of source code zip artifacts, which are used to import the various apps and services. Unzip the top-level artifact, but don't unzip the child artifacts contained within.

```
$ curl -O -L https://github.com/RHsyseng/RHMAP/raw/master/import-
artifacts.zip
$ unzip import-artifacts.zip
```

4.2.2.2. Parent Project

Within App Studio, the Mobile Application Platform web interface, select the Projects widget:

Figure 4.6. App Studio



Press the *New Project* button, then select *Empty Project*. Provide project name *'ecom-refarch'* and then hit *Create*.

Figure 4.7. Empty Project Creation

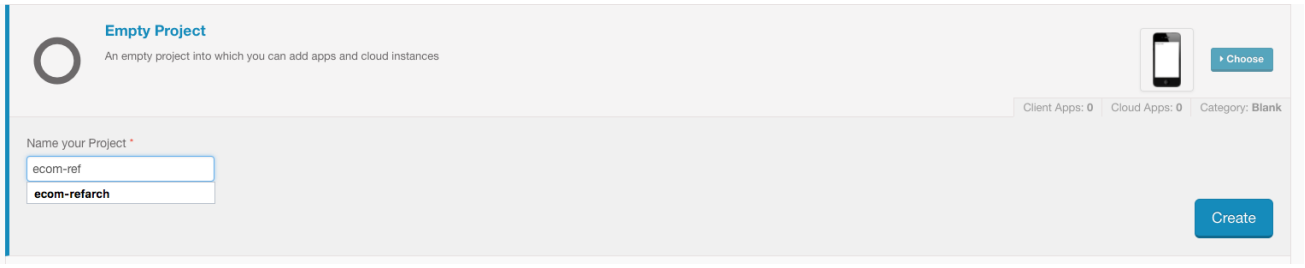
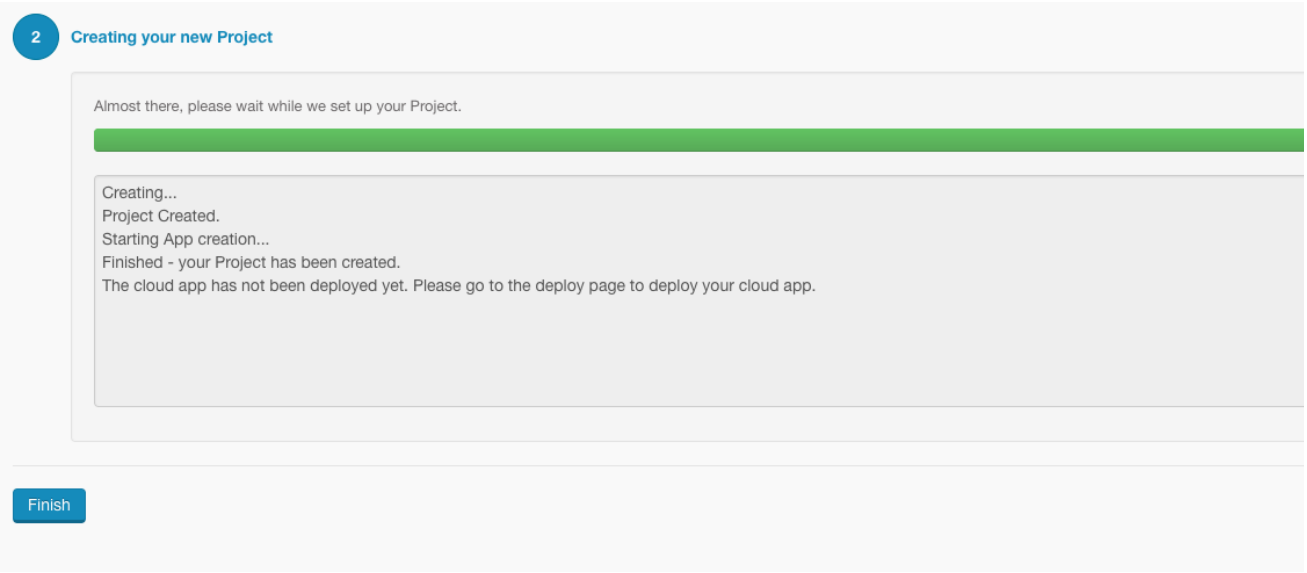
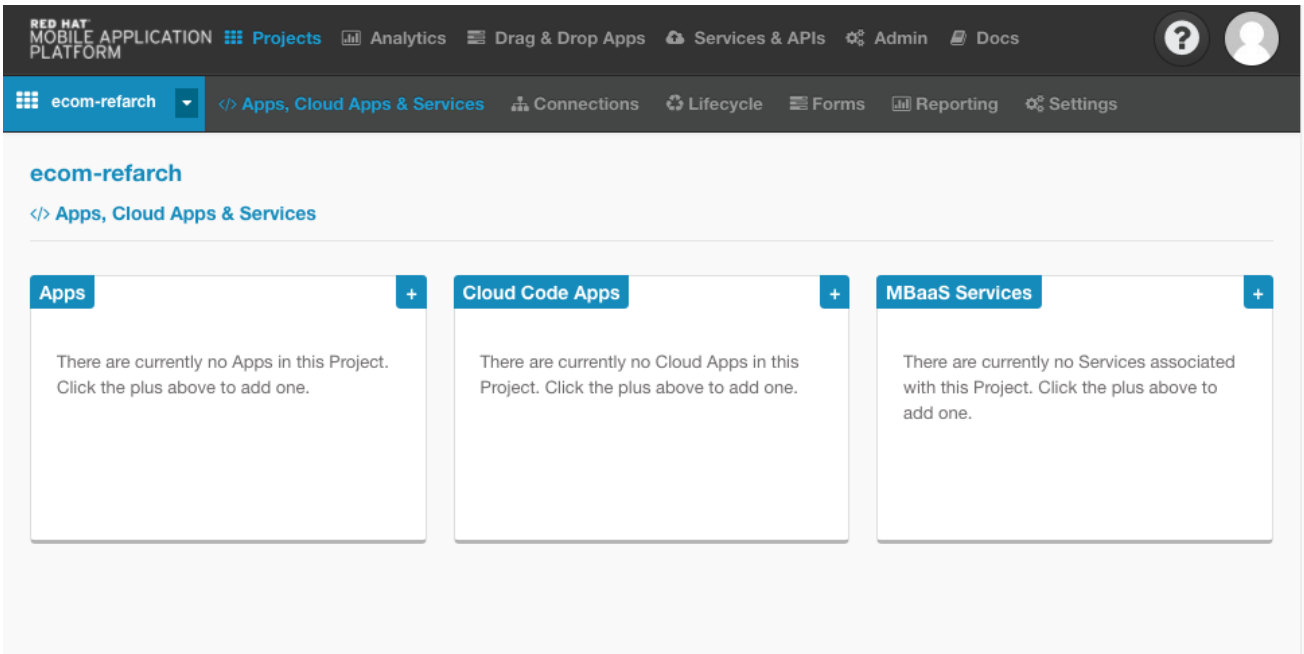


Figure 4.8. Project Creation Complete



Use the *Finish* button after completion to navigate to the project dashboard.

Figure 4.9. Project Dashboard



4.2.2.3. MBaaS Health Monitor

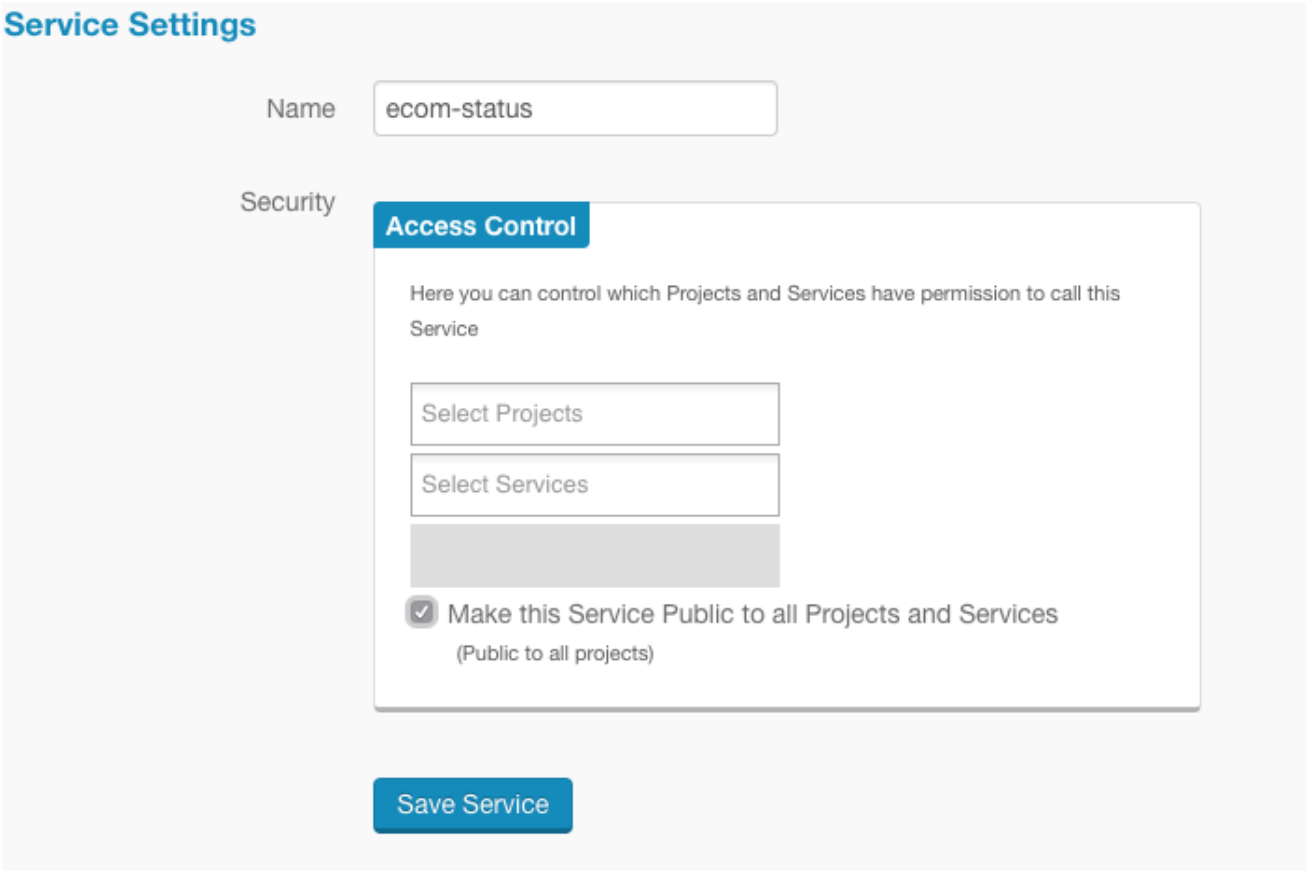
From the top of the screen, select *Services and APIs* to start service creation. Once there, click the *Provision MBaaS Service/API* button to start the creation process.

Next, select *Import Existing Service*, name the service *ecom-svcs-health*, and click *Next*.

For *Import From*, select *Zip File*, provide the *ecom-status.zip* file downloaded in previous steps, then click *Next*, followed by *Finish*.

From the new service app's *Details* page, scroll down and mark *Make this Service Public to all Projects and Services*, then *Save Service*.

Figure 4.10. Marking Service Public



If prompted, confirm pushing of environment variables, then from the left-hand Navigation panel, select *Deploy*.

Use the *Deploy Cloud App* button to instantiate a new build and deployment of the service.

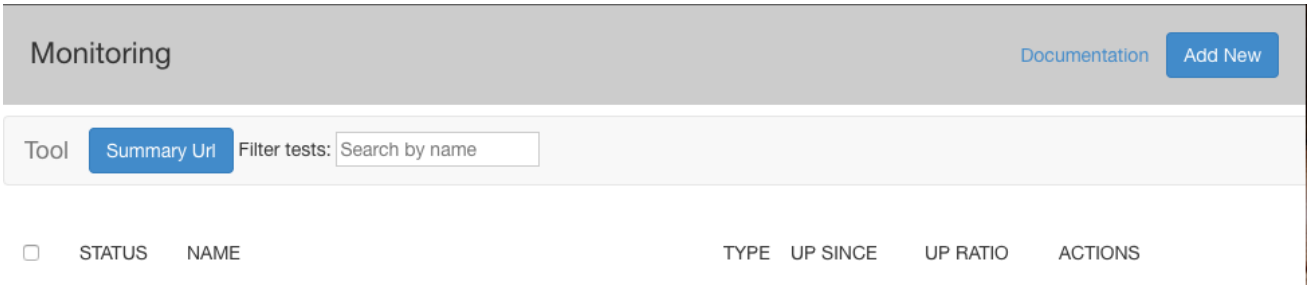
Figure 4.11. Deploy Cloud App

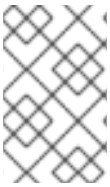
After initiating deployment, use the left-hand navigation to return to *Details*.

The status of the project will change from *Stopped* to *Running* once the triggered build and deployment complete.

Once running, copy the 'Current Host' URL value and open a new tab, append */admin* to the copied value, and navigate to the service page.

Figure 4.12. Health Monitor Service Details



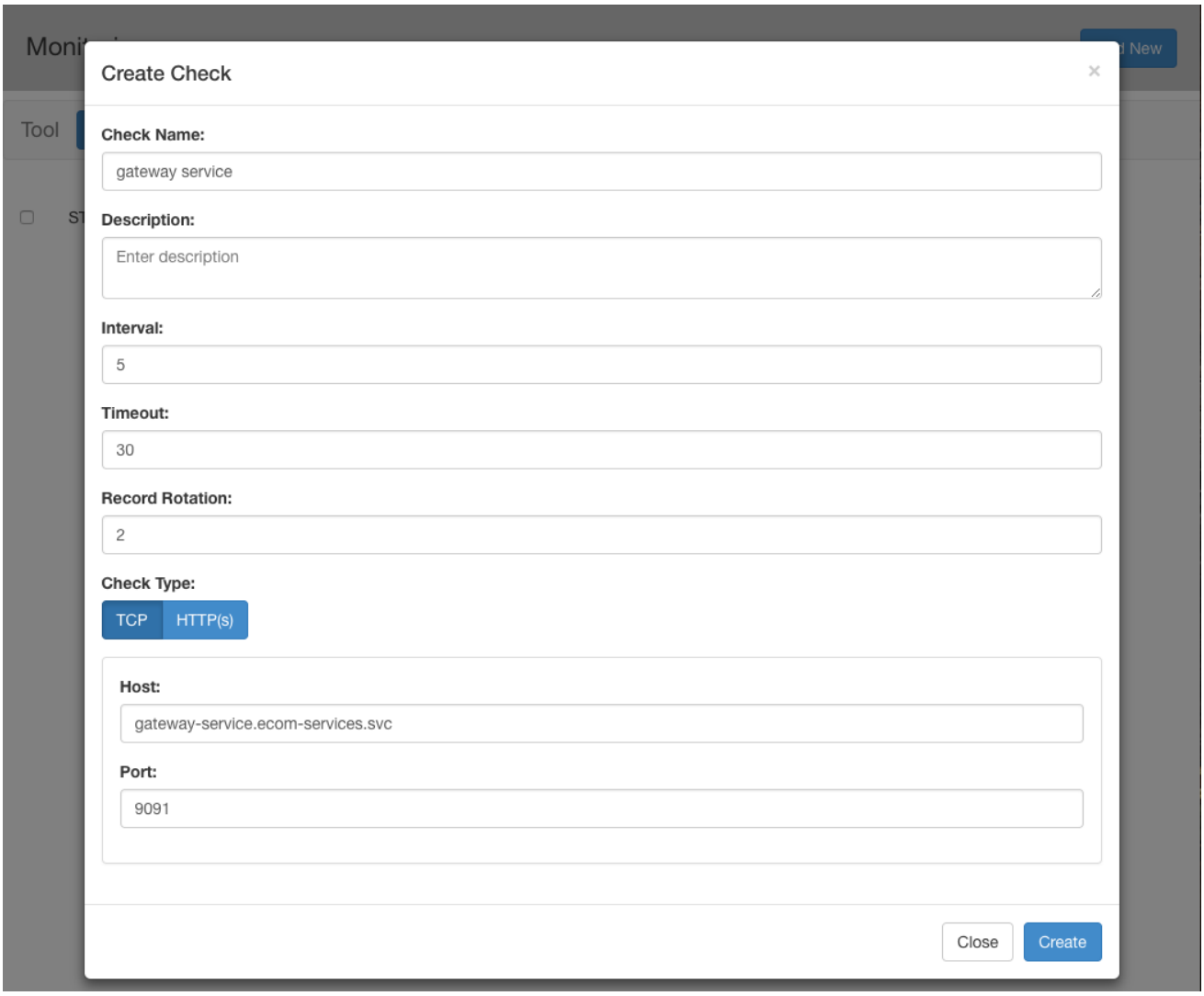


NOTE

If */admin* is not appended to the URL value, the interface will disable the *Add New* button and restrict functionality of existing checks to read-only. To create new checks or kick off existing ones, the admin context is required.

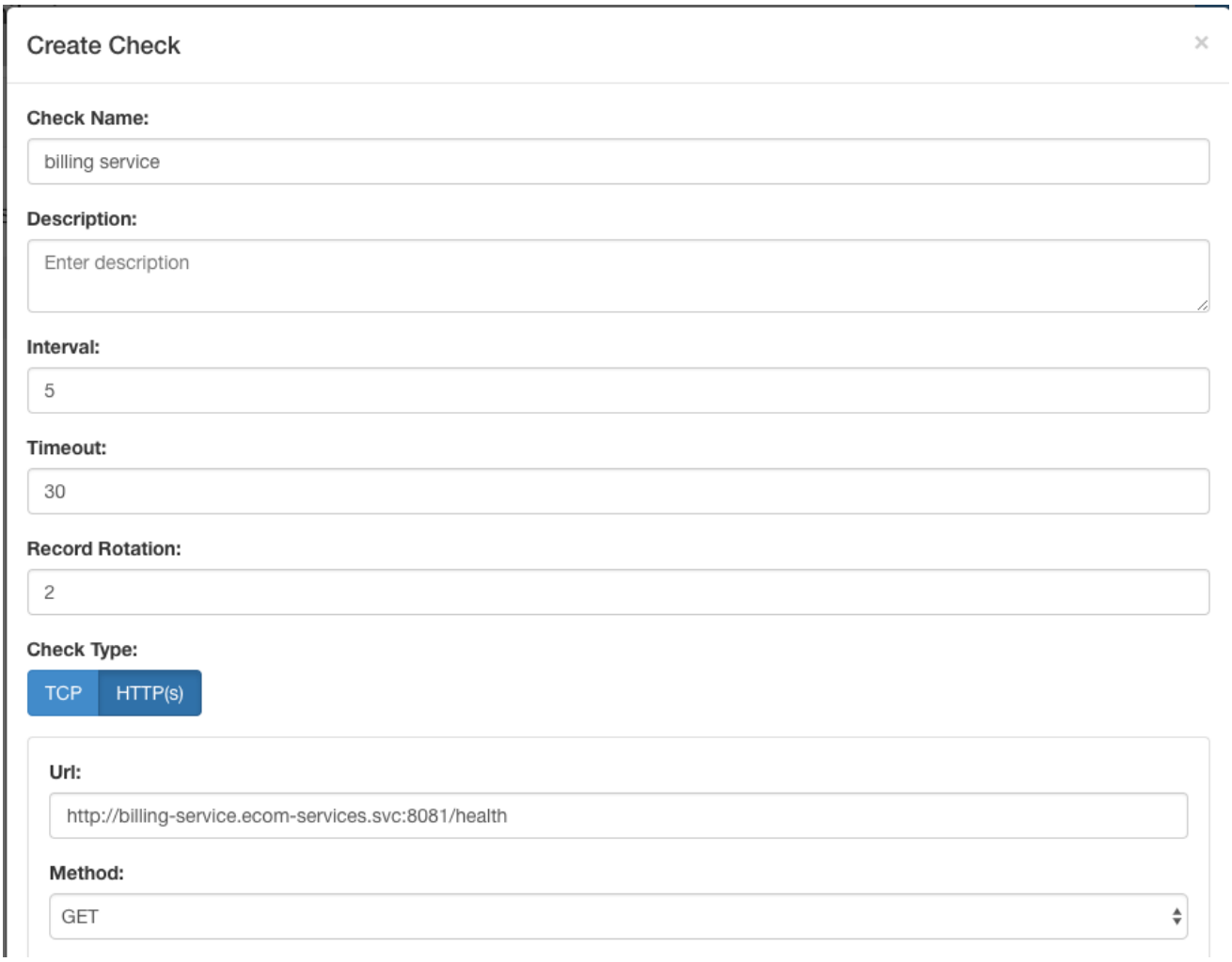
Use the *Add New* button in the upper right-hand corner, configure the modal prompt as shown below, then click *Create*:

Figure 4.13. Gateway Service Check Configuration



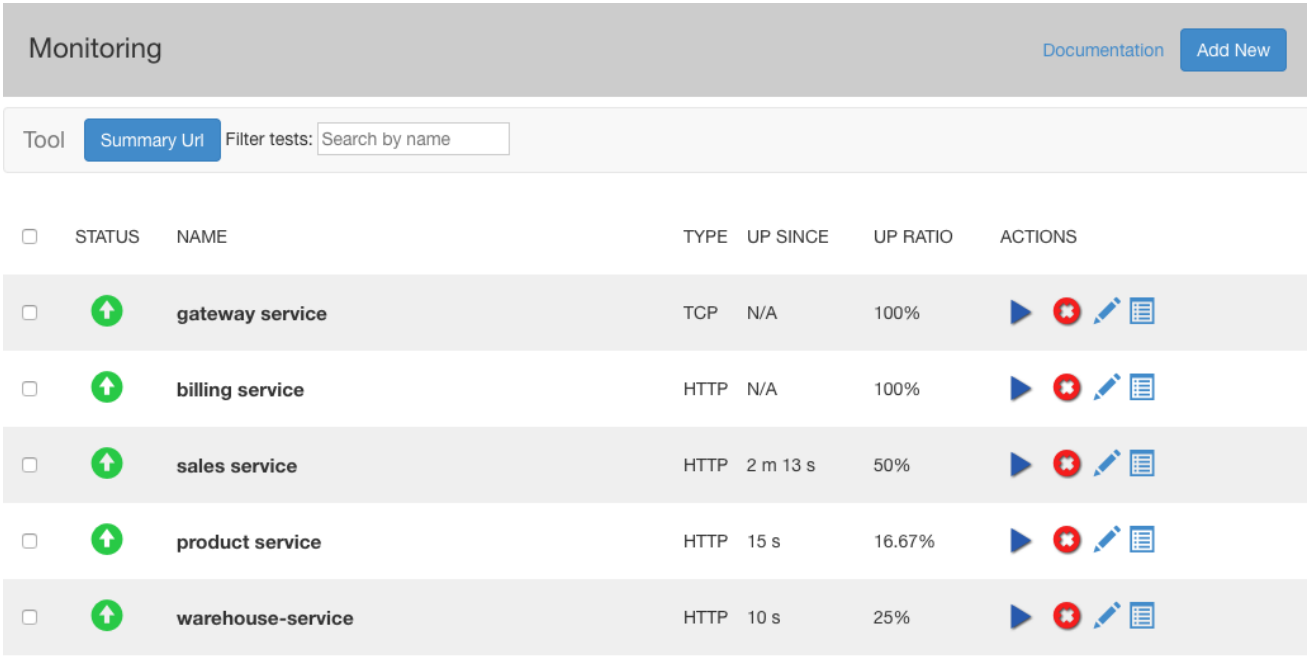
Use the *Add New* button once again to create a check for Billing Service, this time utilizing the *HTTP(s)* protocol option instead of TCP, as shown below:

Figure 4.14. Billing Service Check Configuration



Create 3 additional new services in the same fashion for the *sales-service*, *product-service*, and *warehouse-service*, altering the HTTP(s) URL to match the service destination for each.

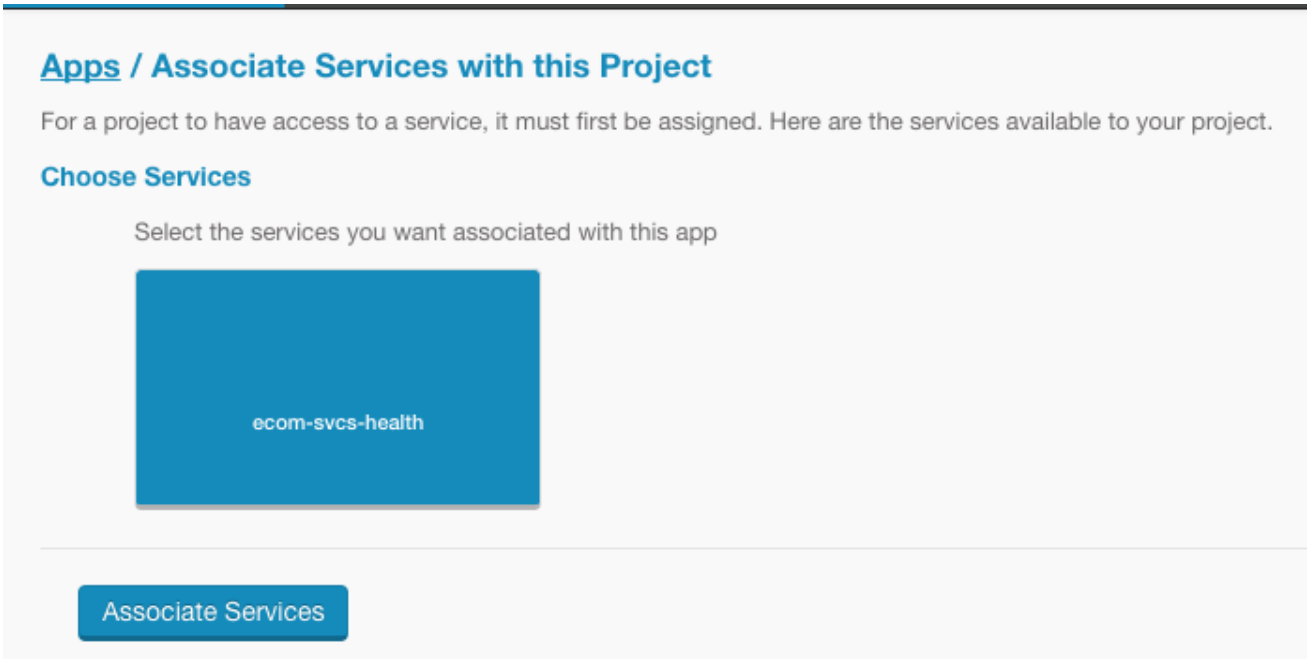
Figure 4.15. Health Monitor Service



At this point, click checkbox next to all 5 services and then click the *Summary Url* button. Copy the URL somewhere close at hand, as it will be required further on.

Lastly, use the *Projects* link in the upper navigation bar to navigate to the *ecom-refarch* project dashboard. Click the plus button in the upper right corner of the *MBaaS Services* widget and click the service to associate it to your project.

Figure 4.16. Associating the Service



NOTE

Association of services is not limited to a single project, but rather allows applications within linked projects to call the service via the *fh.mbaas()* function of the cloud API. The cloud application built below uses the service's exposed URL value rather than a direct API call, thus the MBaaS service does not require association to the project. Since this is atypical of most MBaaS service access, the link is still established as part of the example.

4.2.2.4. Cloud Application

Return to the project dashboard, and use the plus button to add a new *Cloud Code App*.

Just as with the *Health Monitor* service creation, choose to *Import Existing App*, name the app *ecom-cloud*, provide the previously acquired *ecom-cloud.zip* file, and follow the prompts until reaching the application's dashboard.

Using the left-hand navigation, go to *Environment Variables*, then click the *Add Variable* button.

Provide a new variable with the name of *MBAAS_HEALTH_URL*, and use the URL copied from the Monitoring service in the previous step, as the variable value. Click *Push Environment Variables*.

Navigate to *Deploy* and start a new deployment of the cloud app. Monitor the application dashboard *Details* screen for results.

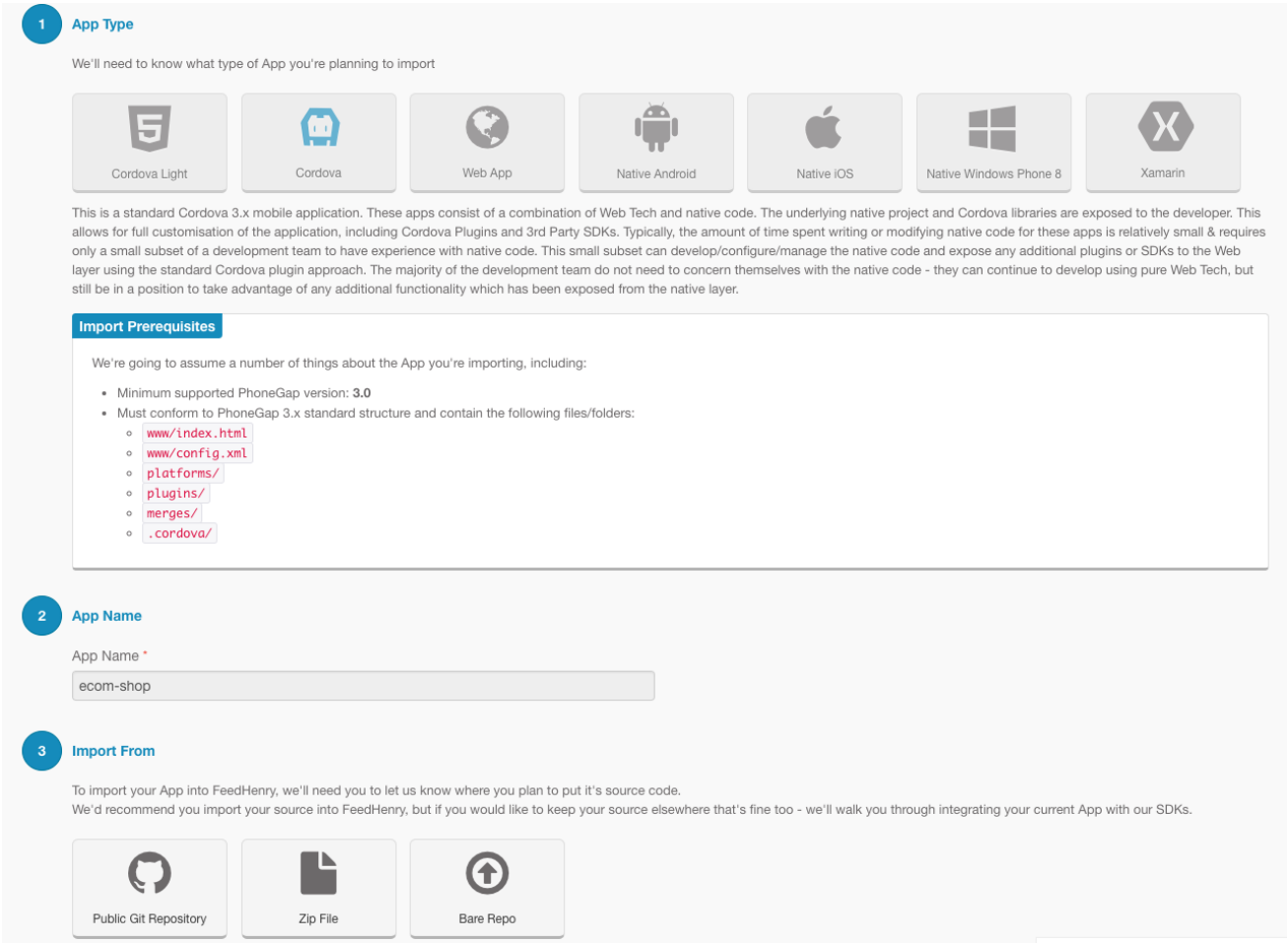
Once deployment is complete, it's possible to directly reach your cloud application via browser at the given *Current Host* address, however, since the cloud application is not intended as a public-facing tool, the page will simply indicate the general availability of the application.

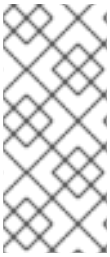
4.2.2.5. Mobile Client Application

Return to the project dashboard and use the plus button on the *Apps* widget.

As before, choose to *Import Existing App* of type *Cordova*, name the application *ecom-shop*, and use the provided artifact with the same name as the import source. The various steps for integration are presented and can be reviewed, but no action is necessary. Follow the prompts until taken to the application *Details* page as before.

Figure 4.17. App Import from Zip



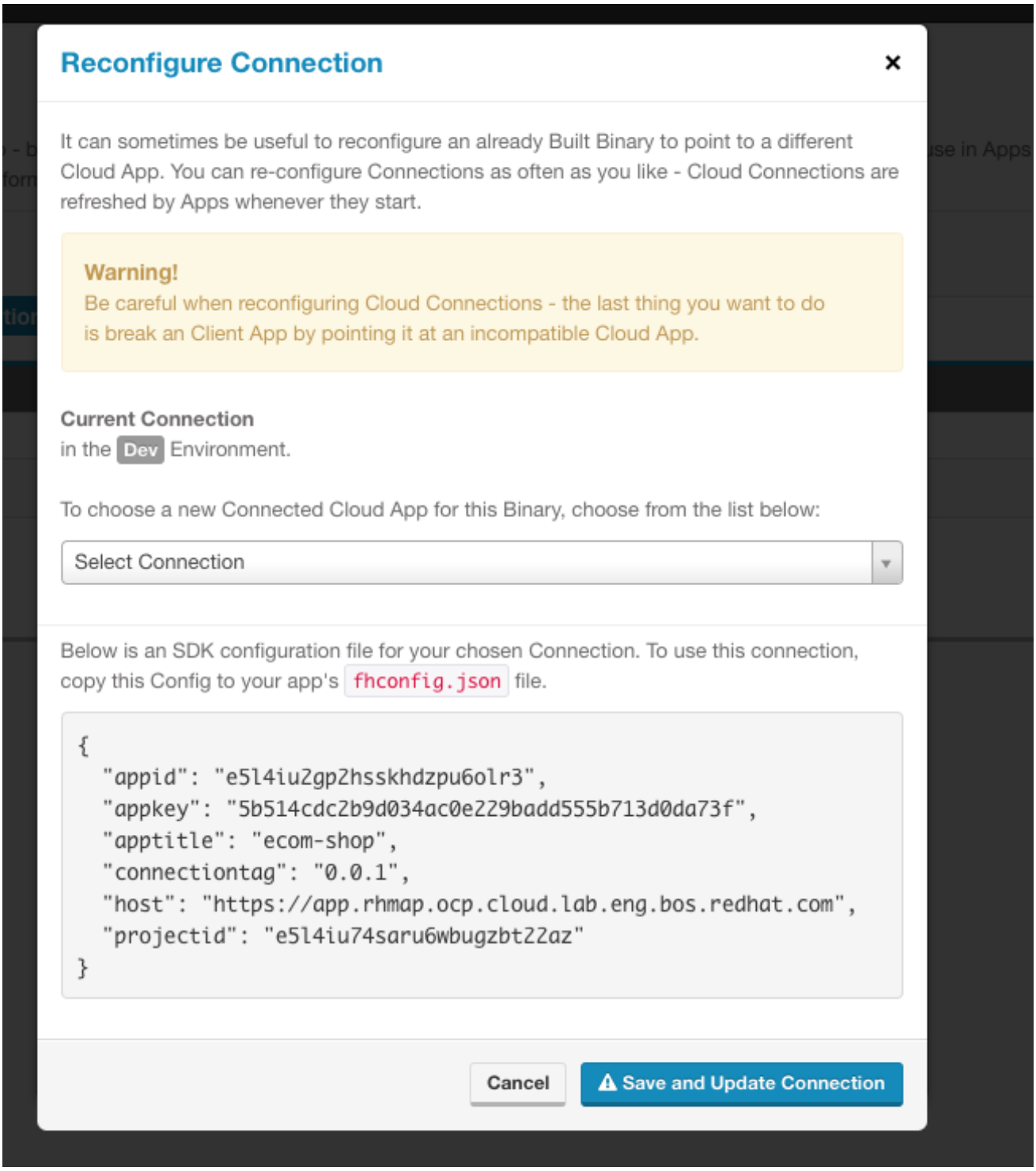


NOTE

When following the prompted steps, the user may encounter a suggested link for integration of the SDK which resolves as a 404 error. At the time of writing, the issue has been road-mapped for correction. For the purpose of this document, the intended reference material is informative, although not essential, as the steps suggested have already been completed prior to assembling the source code artifacts.

From *Details*, select the *Connections* link found in the horizontal navigation bar, then click the *ecom-shop* link inside the *Client App* column. Copy the SDK configuration JSON contents in the bottom box to clipboard, then hit *Cancel* to close the modal window.

Figure 4.18. Connection Details



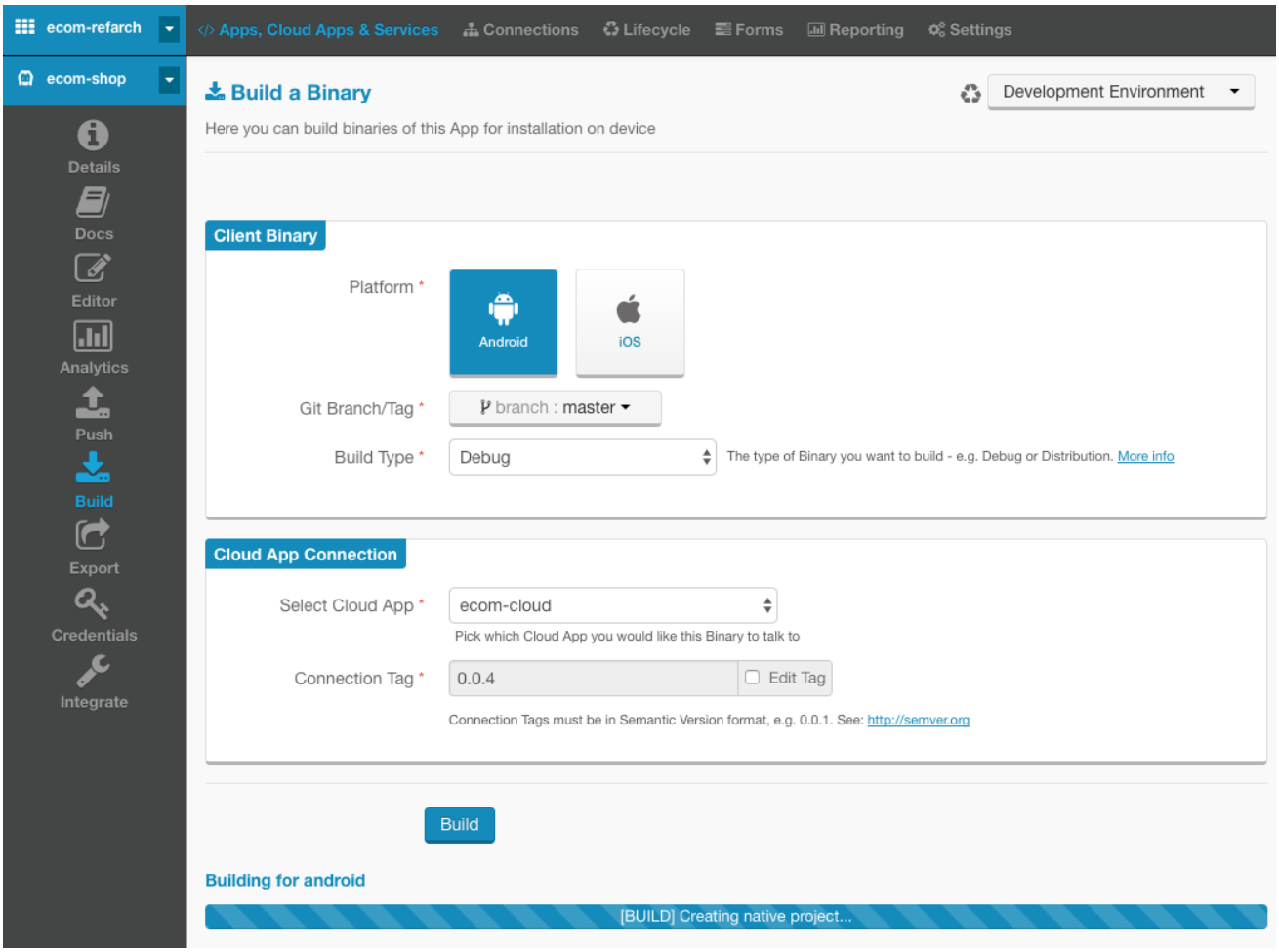
Return to the *ecom-shop* dashboard, navigate to *Editor* on the left-hand side, expand the *www* directory, then select the *fhconfig.json* file. Replace the contents of the file with the clipboard contents attained from the Connections window and then click *File > Save*.

NOTE

By default, RHMAP Studio will complete the above modification to *fhconfig.json* when a build occurs. Details have been included here so that the reader is made aware of the necessary change and source of information required.

Navigate to *Build* from the left-hand bar, select the platform of your choice, and use *Build* to create a native application artifact for the *ecom-shop* project.

Figure 4.19. Build Screen



After the build completes, a modal window presents both QR code and direct download link for fetching of the resulting artifact.

Figure 4.20. Download Artifact

4.2.2.6. Web Portal Client Application

Once more, return to the project dashboard and add a new client application under *Apps*.

As with the *ecom-shop* application, begin the import process for the provided *ecom-portal* artifact file, this time using the *Web App* type instead of *Cordova*. As before, follow the prompts to completion. Lastly, visit *Connections* to capture the *ecom-portal* details, which will differ from those provided for *ecom-shop*, and replace the contents of the *src/fhconfig.json* file with the new connection information.

Once finished, navigate to *Deploy* and start a new deployment for the cloud portal app. Monitor the *Details* page for application state change.

Once complete, the *Current Host* URL can be used to navigate to the portal landing page, showing a list of featured products. Successful loading and visibility of this page indicates that all cloud and client application setup is complete and properly functioning.

CHAPTER 5. DESIGN AND DEVELOPMENT

5.1. OVERVIEW

The source code for the Red Hat Mobile Application Platform E-Commerce example project is made available in a public github repository. This chapter briefly covers each component and its functionality. Note that in the example client applications, the JavaScript SDK is used to demonstrate both Cordova and Web App development. However, Red Hat Mobile Application Platform offers five different SDKs for eight types of mobile application development, reaching well beyond the examples described herein:

- JavaScript SDK (Cordova light, Cordova, Form apps and Web apps)
- iOS SDK (Native iOS apps)
- Android SDK (Native Android apps)
- C#/.NET Native SDK (Xamarin and Native Windows apps)
- Appcelerator SDK (Appcelerator Hybrid apps)

5.2. HEALTH MONITOR MBAAS SERVICE

Given that a significant portion of e-commerce functionality relies on the availability of both product information and transaction processing, an MBaaS service, created from the *Status Monitor Service* template, is made available to clients via the cloud application. The E-Commerce microservices suite features 5 services relevant to the availability of the system from a user's point of view. Each of those services is configured with a health check endpoint which is pinged by the Monitoring service every 5 minutes via the user-configured *Check* tasks performed during initial configuration.

5.2.1. Template Modifications

Status Monitor Service, like most other provided templates, is designed to utilize an internal MongoDB instance for persistence by default. The service application is built on Node.js, much like the cloud and portal applications, and manages dependencies via npm. The database connectivity dependency used within the template is mongoose. Originally built on earlier versions of the Mobile Application Platform, the template requires a few changes to allow proper integration with the newer versions of MongoDB included in more recent OpenShift and Mobile Application Platform releases.

The first change necessitated for compatibility is updating the mongoose dependency. The template initially requests version 3.8.15, which has been changed in the *package.json* file to 4.11.4.

```
"dependencies": {
  ...
  "mongoose": "4.11.4",
  ...
}
```

Second, a slight modification has been made to the application instantiation portion of the application to prevent a race condition from arising. As the template stands at the time of writing, initial database connectivity is attempted before configuration of the application has completed, resulting in errors regarding not yet instantiated or incorrect db *connectionUrl* info. By wrapping the application initialization code in a SDK call which checks and correctly sets the variables needed, the race condition is eliminated.

```
var mbaasApi = require('fh-mbaas-api');
var express = require('express');
var mbaasExpress = mbaasApi.mbaasExpress();
var cors = require('cors');

// handle race condition with OCP 3.X; call for connectionString
// and set to enviro var prior to init of monitor
mbaasApi.db({
    "act": "connectionString"

}, function (err, connectionString) {

    if (err) throw err;

    console.log("connectionString fetched: " + connectionString);

    // rest of app init code here
    ...
});
```

As mentioned prior, most templates will default to using an internal MongoDB setup for persistence. However, it is possible to configure the template to use an external MongoDB instance or cluster by providing a series of environment variables. A quick look at the SDK code responsible for resolving persistence connectivity information yields a list of variables that could be provided to do just that:

```
if (process.env.FH_MONGODB_CONN_URL) {
  return cb(undefined, process.env.FH_MONGODB_CONN_URL);
} else if (process.env.OPENSHIFT_MONGODB_DB_HOST) {
  return mongoConnectionStringOS2(cb);
} else if ("openshift3" === process.env.FH_MBAAS_TYPE) {
  return mongoConnectionStringOS3(cb);
} else if (process.env.FH_USE_LOCAL_DB) {
...
```

Note that the *FH_MONGODB_CONN_URL* can be declared outright to provide the full connectivity string:

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...
[,hostN[:portN]]][/[database][?options]]
```

Backwards compatibility for OpenShift 2.X systems also yields another option where the *OPENSHIFT_MONGODB_DB_HOST* environment variable can be set, as well as a few more variables: *OPENSHIFT_MONGODB_DB_PORT*, *OPENSHIFT_MONGODB_DB_USERNAME*, and *OPENSHIFT_MONGODB_DB_PASSWORD*. *OPENSHIFT_APP_NAME* can also be used to define which database the application should authenticate against and utilize.

```
function mongoConnectionStringOS2(cb) {
  debug('Running in OpenShift 2, constructing db connection string from
additional env vars');
  var connectionString,
    host = process.env.OPENSHIFT_MONGODB_DB_HOST,
    user = process.env.OPENSHIFT_MONGODB_DB_USERNAME,
    pass = process.env.OPENSHIFT_MONGODB_DB_PASSWORD,
    port = process.env.OPENSHIFT_MONGODB_DB_PORT,
    dbname = process.env.OPENSHIFT_APP_NAME;
```

```
  connectionString = mongodbUri.format({
    username: user,
    password: pass,
    hosts: [{
        host: host,
        port: port
    }],
    database: dbname
  });
  process.env.FH_MONGODB_CONN_URL = connectionString;
  return cb(undefined, process.env.FH_MONGODB_CONN_URL);
}
```

If none of the mentioned variables have been added, an OpenShift 3.X system, by default, will have an *FH_MBAAS_TYPE* value set, thus leading the code to resolve and return internal connectivity information.

5.2.2. Cloud Application Integration

The Monitoring service features an aggregated monitoring endpoint which the cloud application utilizes to collectively evaluate the health of the microservices suite. The cloud application is configured to call this endpoint at each client request, record detailed results of the response, and mask that response down for the client to a simple *ok* or *fail*.

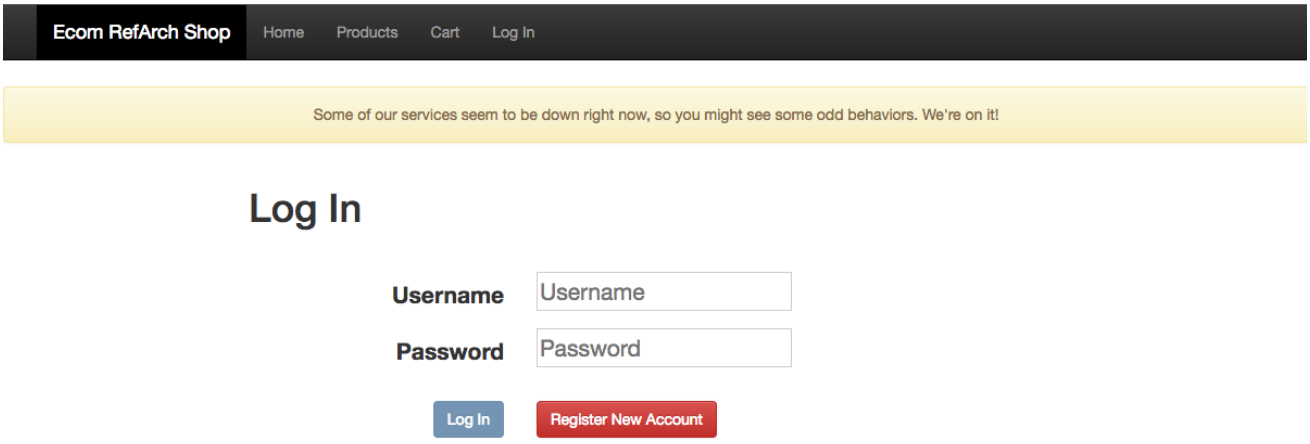
```
request.get(healthUrl, //dynamic url of aggregated health check via enviro
prop
    function(error, response, body) {

        if (error) {
            return callback(error, null);

        } else if (response.statusCode !== 200) {
            return callback("ERROR: bad status code " +
response.statusCode, null);

        } else {
            var health = JSON.parse(body);

            if (health.result === 'ok') {
                return callback(null, '{"result": "ok"}');

            } else {
                console.error('health check detected a problem, status ' +
health.result);
                health.details.forEach(function(serviceEntry) {
                    console.error(serviceEntry.name + ' reported as ' +
serviceEntry.result);
                });
                return callback(null, '{"result": "fail"}');
            }
        }
    }
);
```

5.2.3. Client Application Integration

Client applications use the cloud API at a regular interval to call the endpoint exposed in the cloud application to retrieve a summation of microservice suite health results and notify the user when their experience will be hindered due to a reported *fail* status.

```
$interval(function() {
    AuthService.checkAvailability().then(function(data) {

        $rootScope.availability = (data.status === 'ok');
        console.log("health check..." + data.status + ' response to bool '
+ $rootScope.availability);
    });
}, 5000);
```

Figure 5.1. Client Availability Warning



5.3. CLOUD APPLICATION

In order for client applications to leverage the power of the Mobile Application Platform, a cloud application is set up to mediate requests to and from the E-Commerce microservices gateway and make available platform features such as data persistence, synchronization, caching, and push notifications.

5.3.1. Template Scaffolding

The Mobile Application Platform offers a scaffolding template for kickstarted development of Node.js cloud applications with SDK and other requirements pre-configured. The cloud application imported in the previous chapter was initially built atop the provided template.

5.3.2. SDK Configuration

As with most provided templates, a few assumptions are made about the source code which must be addressed for proper platform integration. In the case of cloud applications, an *application.js* file is assumed which specifies a few basic integrations, such as routing for MBaaS and cloud calls, as well as a health check endpoint & host/port information. A user-defined */cloud* endpoint has been added for routing of custom cloud requests.

5.3.3. cloud.js

The majority of routing logic lies within *cloud.js*, as defined by the express router variable, *cloud*.

```
var cloud = new express.Router();
```

```
cloud.use(cors());

cloud.get('/products', function(req, res) {...};

cloud.post('/customers/authenticate', jsonParser, function(req, res)
{...};

...
```

All requests made from client applications are received here, logged into activity, then forwarded to appropriate services for handling. Once a response is returned from the service, it's then packaged accordingly and returned to the calling client.

```
cloud.get('/products/:sku', function(req, res) {

    activity.record({
        "action": "Product details fetch for sku " + req.params.sku
    }, function(err) {
        if (err) {
            res.statusCode = 500;
            return res.end(util.inspect(err));
        }
        products.get(req.params.sku, function(err, data) {
            if(err) {
                res.statusCode = 500;
                return res.end(util.inspect(err));
            }
            res.send(data);
        });
    });
});
```

5.3.4. Cloud Activity Logging

In most request routings, a call to *activity.record* is made at the start of each process point for received client requests. The *activity* service works with the Mobile Application Platform data caching API to record each request made so that data usage and analysis can be performed.

```
activity.record({
    "action": "action to record"

}, function(err) {...};
```

The express router also allows for fetching and resetting of the activity list via the */activity* context.

```
cloud.post('/activity/list', function(req, res) {

    activity.list(req.body, function(err, data) {
        if (err) {
            res.statusCode = 500;
            return res.end(util.inspect(err));
        }
        res.json(data);
    });
});
```

```
cloud.post('/activity/reset', function(req, res) {

        activity.reset(function(err, data) {
            if (err) {
                res.statusCode = 500;
                return res.end(util.inspect(err));
            }
            res.json(data);
        });
    });
```

5.3.5. Microservice MBaaS Service Call Proxying

A majority of endpoints defined within the express router configuration forward client requests to the API Gateway entry point of the e-commerce microservices suite. A few additional routes handle internal operations such as administration of activity logging. The express router is capable of handling the GET, POST, PUT, PATCH, and DELETE commands used within the microservice suite, as well as a few others not in use. It also handles the passing of URL parameters to the processing function. Requests carrying a JSON payload utilize a JSON parser, *bodyParser*, to properly translate the contents of the request for service consumption.

Example usage of inline parameters and JSON parsing:

```
cloud.post('/customers/:userId/orders', jsonParser, function(req, res) {

    activity.record({
        "action": "Save new user order"

    }, function(err) {
        if (err) {
            res.statusCode = 500;
            return res.end(util.inspect(err));
        }
        sales.saveOrder(req.params.userId, req.body, function(err, data) {
            if(err) {
                res.statusCode = 500;
                return res.end(util.inspect(err));
            }
            res.send(data);
        });
    });
});
```

5.4. CLIENT APPLICATION REQUIREMENTS

Both mobile and portal client applications were developed with a unified set of functional feature requirements in mind:

- Highlight featured product recommendations
- List of all available products, including product details
- Add/Remove/Checkout Cart functionality

- New user registration
- User authentication
- Session synchronization
- Automatic session timeout
- Persistent, synchronized list of a user's 'favorites' across clients
- Payment processing
- Order details and processing status
- Notification of back-end service availability

5.5. CLIENT PLATFORM INTEGRATIONS

A number of features available in the Mobile Application Platform are used via cloud/client cooperation to satisfy some of the listed requirements. For a full list of available Client API functionality and features, please refer to the Red Hat Mobile Application Platform 4.4 Client API documentation.

5.5.1. Data Synchronization

```
$fh.sync.init(dataset_id, options, callback)
```

The cloud sync framework requires handler functions defined at the client application level responsible for providing access to the back end data & management of data collisions. Each unique dataset being synchronized is identified by a *dataset_id* specified as the first parameter when calling any function of the sync framework. The platform also provides accessibility of sync events at the cloud level via request interceptors. Implementation-wise, both client applications use data sync to maintain and expose a persistent list of "favorite" products for each authenticated user.

Client application sync initialization and usage, *SyncService.js*:

```
init: function () {

    var deferred = $q.defer();

    console.log('initializing sync service');
    $fh.sync.init({
        "do_console_log": true,
        "storage_strategy": "dom"
    });

    ...

    $fh.sync.manage(datasetId);
    $fh.sync.notify(function (notification) {
        if ('sync_complete' === notification.code) {
            $fh.sync.doList(datasetId, success, fail);
        }
        else if ('local_update_applied' === notification.code) {
            $fh.sync.doList(datasetId, success, fail);
        }
```

```
        else if ('remote_update_failed' === notification.code) {
            var errorMsg = notification.message ? notification.message.msg
? notification.message.msg : undefined : undefined;
            fail(errorMsg);
        }
    });
    return deferred.promise;
  },

  getList: function() {
    return promiseWrap(function (success, fail) {
        $fh.sync.doList(datasetId, function (r) {
            success(unwrapList(r));
        }, fail);
    });
  },

  update: function (item) {
      return promiseWrap(function (success, fail) {
          $fh.sync.doUpdate(datasetId, item.id, item.data, success, fail);
      });
  },

  ...
```

Client application SyncService usage, *HomeController.js*:

```
function addToFavorites(product, $event) {

    $event.stopPropagation();
    SyncService.save(product);
    product.addedToFavorites = true;
    $timeout(function () {
        product.addedToFavorites = false;
    }, 700);
}
instance.addToFavorites = addToFavorites;
```

Cloud application Sync request interceptor, *sync.js*:

```
var globalRequestInterceptor = function(dataset_id, params, cb) {

    console.log('sync intercept for dataset ' + dataset_id + ' with params
' + params);
    return cb(null);
};

fh.sync.globalInterceptRequest(globalRequestInterceptor);
```

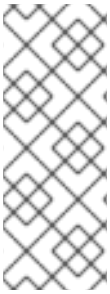
5.5.2. Data Caching

In the example client applications, data caching is used for recording the previously mentioned cloud activity list and tracking of user sessions. Caching functionality is mostly wrapped within calls made to the cloud application. For example, when a user is authenticated in the cloud via a call from a client

application, a token is created and cached, which allows tracking/usage of the session from any client application, and automatic expiration after a configurable amount of time.

Cloud token management, *sales.js*:

```
exports.setCacheKey = function (cacheKey) {
    key = cacheKey;
};

exports.checkToken = function (userId, callback) {
    fh.cache({
        act: "load",
        key: key
    }, function (err, res) {
        if (err) {
            console.error("error checking token ", err);
            return callback(err, null);
        }
        var tokenList = JSON.parse(res);
        var tokenFound = (tokenList || []).indexOf(userId) > -1;
        console.log("token " + userId + " - " + tokenFound);
        return callback(null, {
            'tokenFound': tokenFound
        });
    });
};

exports.setToken = function (userId, callback) {
    fh.cache({
        act: "load",
        key: key
    }, function (err, res) {
        if (err) {
            console.log("error1 " + err);
            return callback(err, null);
        }
        var tokenList = JSON.parse(res);
        ...
    });
};

exports.removeToken = function (userId, callback) {
    fh.cache({
        act: "load",
        key: key
    }, function (err, res) {
        if (err) {
            return callback(err, null);
        }
        var tokenList = JSON.parse(res);
        ...
    });
};
```



NOTE

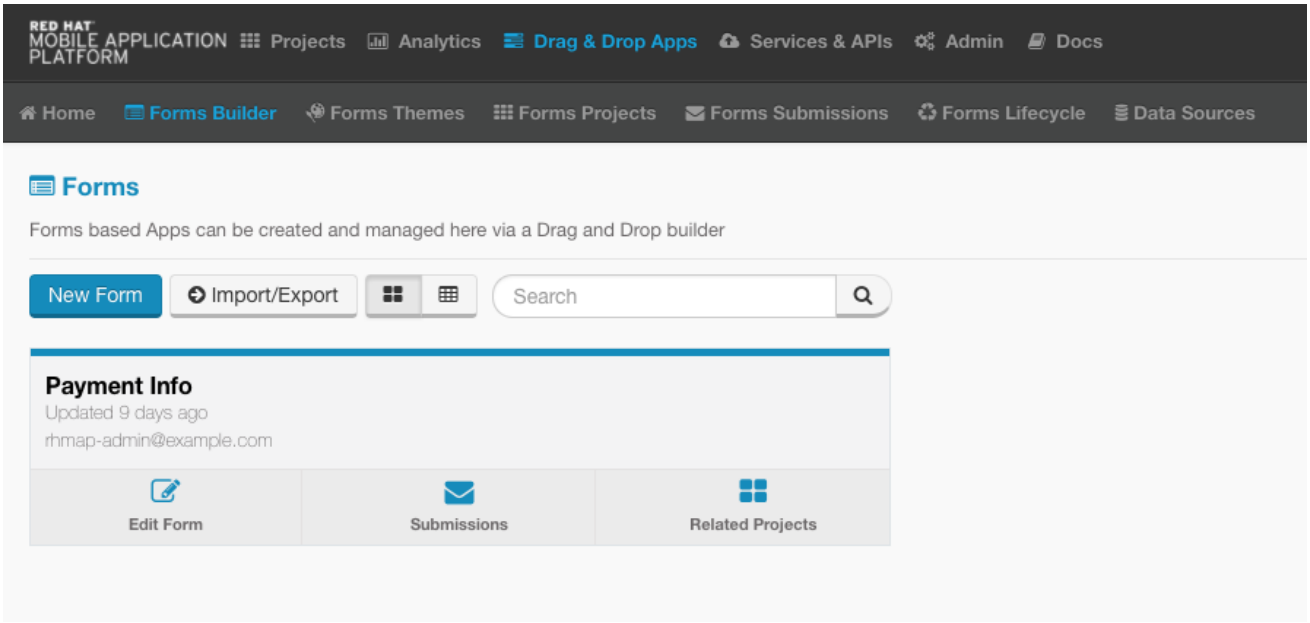
While the example cloud application builds upon the more rudimentary *$fh.cache()* functionality for demonstrating tracking of user sessions, the Cloud API provides a more direct solution - *$fh.session()*. This tool natively offers most of the functionality that was manually built around *$fh.cache()* for token tracking demonstration. A collective session id, data object, and expiration time are cached as a part of the session object, which is available via set, get, and remove endpoints.

5.5.3. Form Builder

The Forms Builder allows mobile forms to be quickly and easily created using drag and drop components. While it's possible to build entire applications around this functionality, integration of a single example form for collection of payment information could better handle form-specific validation and third-party payment system integration. For the purposes of this project, a payment information form was generated to showcase the feature, but not integrated into the client applications:

Similarly to MBaaS Services and APIs, *Drag and Drop Apps* are a sharable component, and thus housed in their own dashboard within the Mobile Application Platform.

Figure 5.2. Drag and Drop Apps Dashboard



Seen here, drag and drop components allow for easy definition of form content and configuration of each field:

Figure 5.3. Constructing the Form

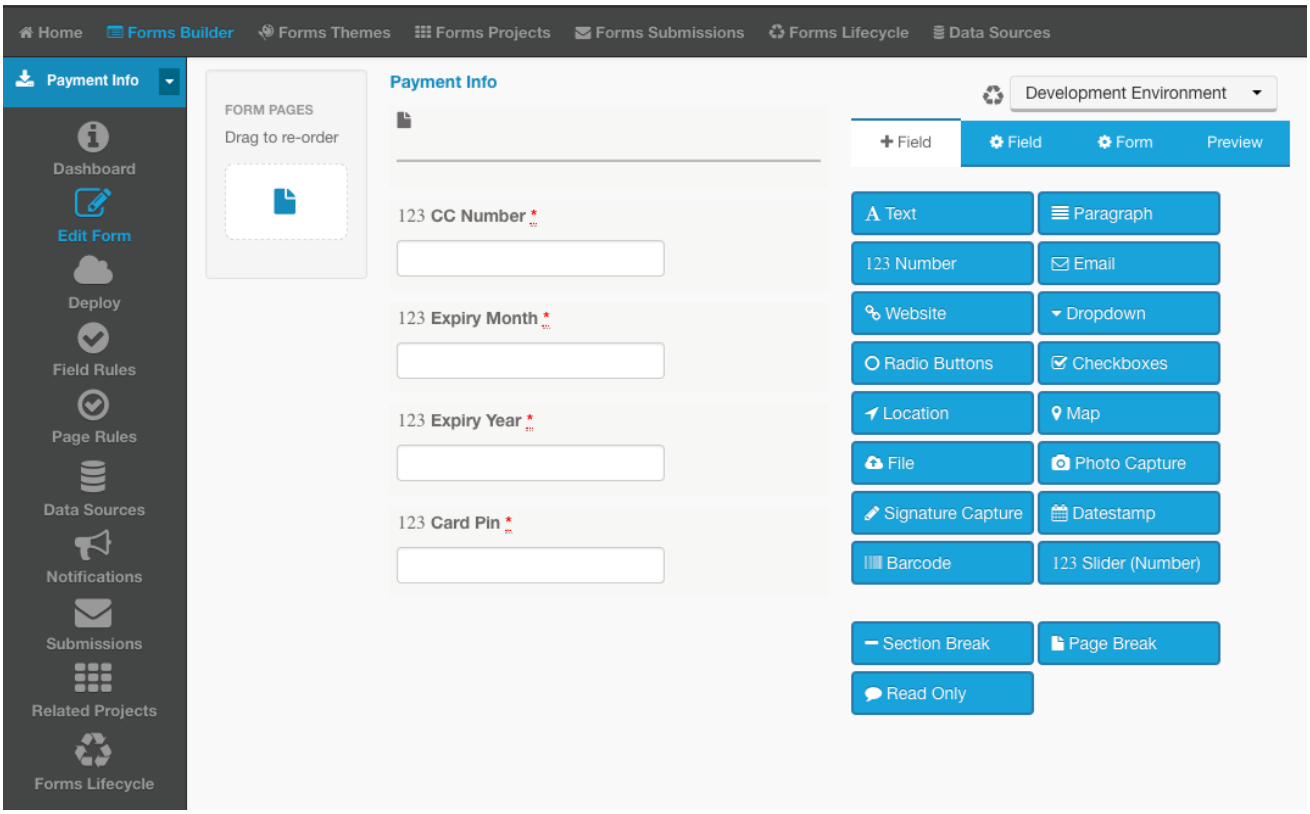
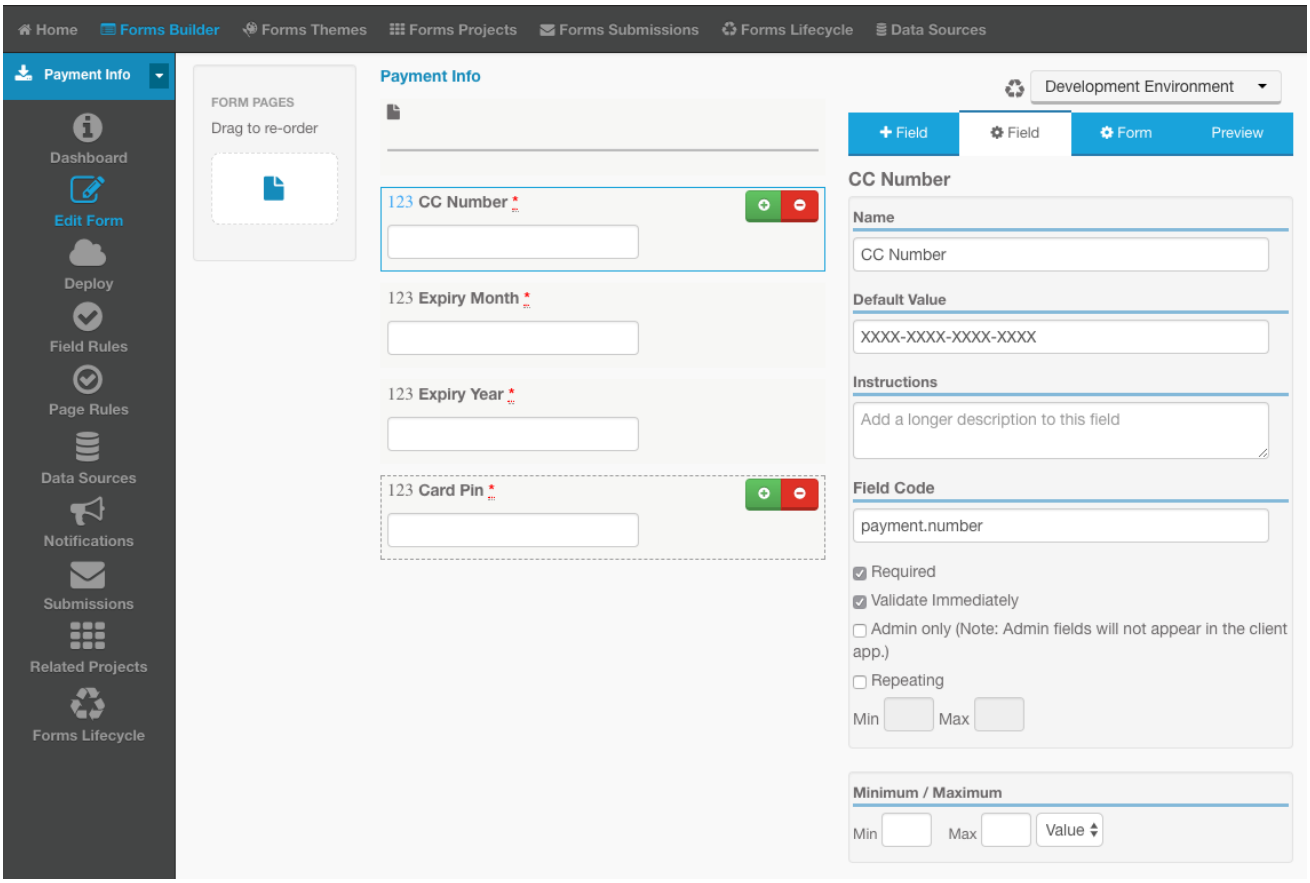
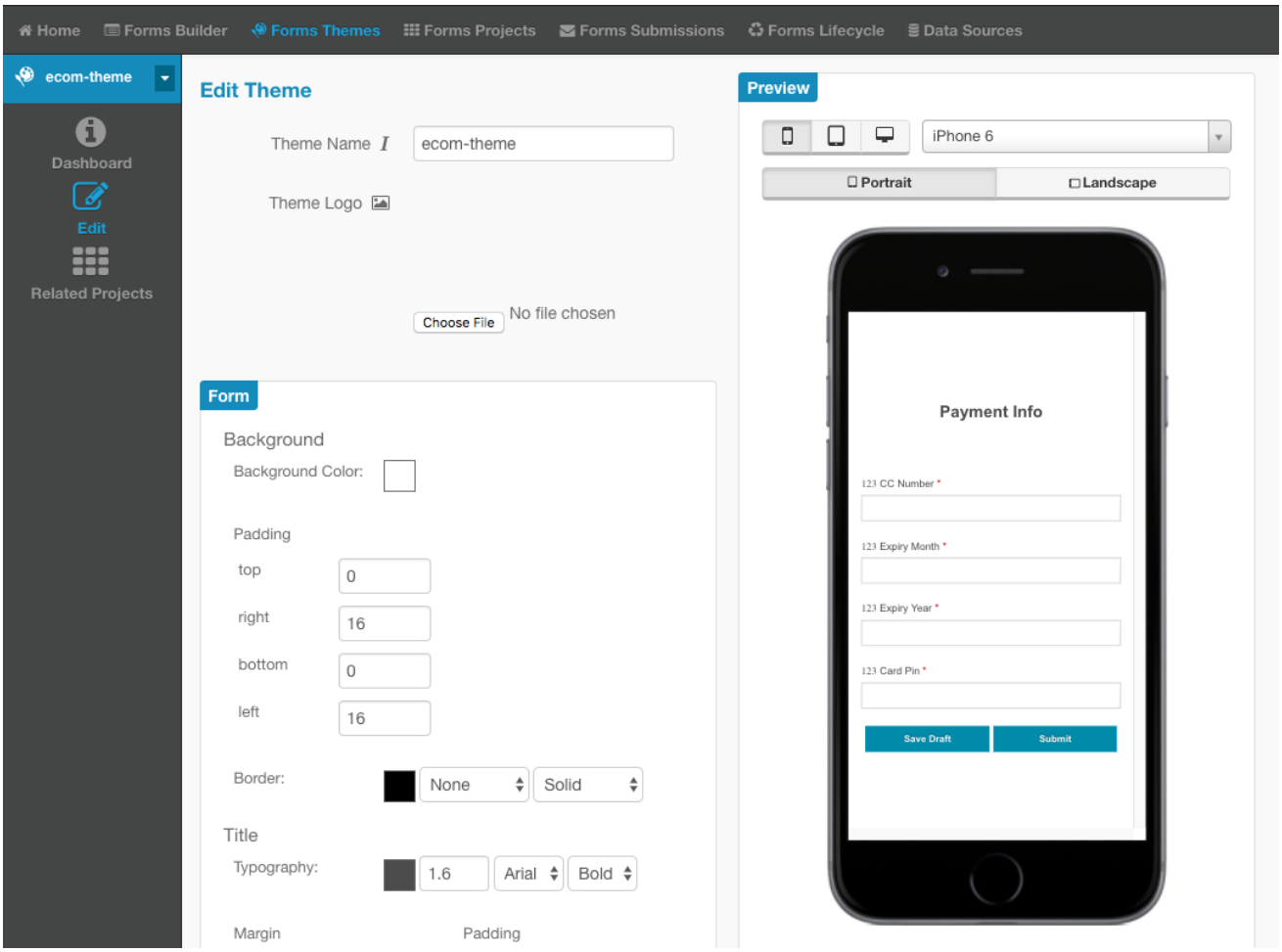


Figure 5.4. Field Configuration



After construction and configuration, form themes can be created to allow for seamless visual integration into existing apps or complete customization of application appearance:

Figure 5.5. Form Theming



Once developed, forms can further be enhanced via available platform tooling to include field-specific and page-specific rule sets, data source coupling, broadcast of change notifications, submission routing, and more. For more information on using Form Builder, refer to the Red Hat Mobile Application Platform 4.4 Drag and Drop Apps Guide.

5.5.4. Cloud Storage

```
$fh.db(options, callback(err, res))
```

The *$fh.db* call provides access to hosted data storage, which supports CRUDL (create, read, update, delete, list) and index operations. It also supports deleteall, which deletes all the records in the specified entity.

5.5.5. Build Farms

When using the *Build* context within the Cordova/Ionic client application, the resulting native mobile artifacts are built via the Mobile Application Platform's Build Farm. This tool enables automation of the build process, record keeping of previous builds, and creation of deployable artifacts for various mobile platforms without requiring the associated infrastructure and tools (e.g. building iOS binaries from a Linux OS).

5.6. MOBILE CLIENT APPLICATION

A mobile application has been implemented to showcase usage of the provided SDK to communicate with the cloud. The cloud API aids to abstract sharable functions and data sets to the cloud layer so that any number of other client applications or instances can share and collaborate with one another.

5.6.1. Cordova

In order to accommodate a variety of target platforms while preserving offline capabilities and access to device APIs, the example mobile application uses the Apache Cordova starter template provided within the Mobile Application Platform. Cordova, as one of several cross-platform frameworks supported, features HTML/CSS/JS mobile development via Node.js application. The framework is open source and widely available via npm. For more information on Cordova, visit the official documentation.

5.6.2. Ionic

The Ionic Framework, built atop Cordova, further simplifies the process of developing cross-platform applications by simplifying access to dozens of native device features via plugin support and themed interface components for rapid UI and feature development.

5.6.3. Template Scaffolding

The Mobile Application Platform offers a scaffolding template to kickstart development of Cordova and/or Ionic applications with SDK and other requirements pre-configured. Such a template was used initially for starting the example *ecom-shop* source code used for import.

5.6.4. SDK Configuration

As previously mentioned in the context of importing into the App Studio, a few assumptions are made about the configuration of the client. Contents of the *fhconfig.json* file are copied directly from the parameters given in the *Connection* component of the project, using the following structure:

```
{
  "appid": "<app_id>",
  "appkey": "<app_key>",
  "apptitle": "<app_title>",
  "connectiontag": "<connection-tag>",
  "host": "<app_host>",
  "projectid": "<project_id>"
}
```

5.6.5. Cloud Communication

Communications with the cloud app from within the client are performed via SDK. Such calls are wrapped inside services that utilize *$fh.cloud(...)* to form and issue the request:

```
var register = function(user) {

   var defer = $q.defer();
   user.id = null;
   var params = {
       path: '/cloud/customers',
       data: user,
       method: "POST",
       contentType: "application/json",
       timeout: 15000
```

```
    };
    $fh.cloud(params, defer.resolve, defer.reject);
    return defer.promise;
};
```

5.6.6. Running Locally

Users can run an HTML5 presentation of the mobile client locally by issuing the *npm install* command within the home directory of the project, followed by *grunt serve*:

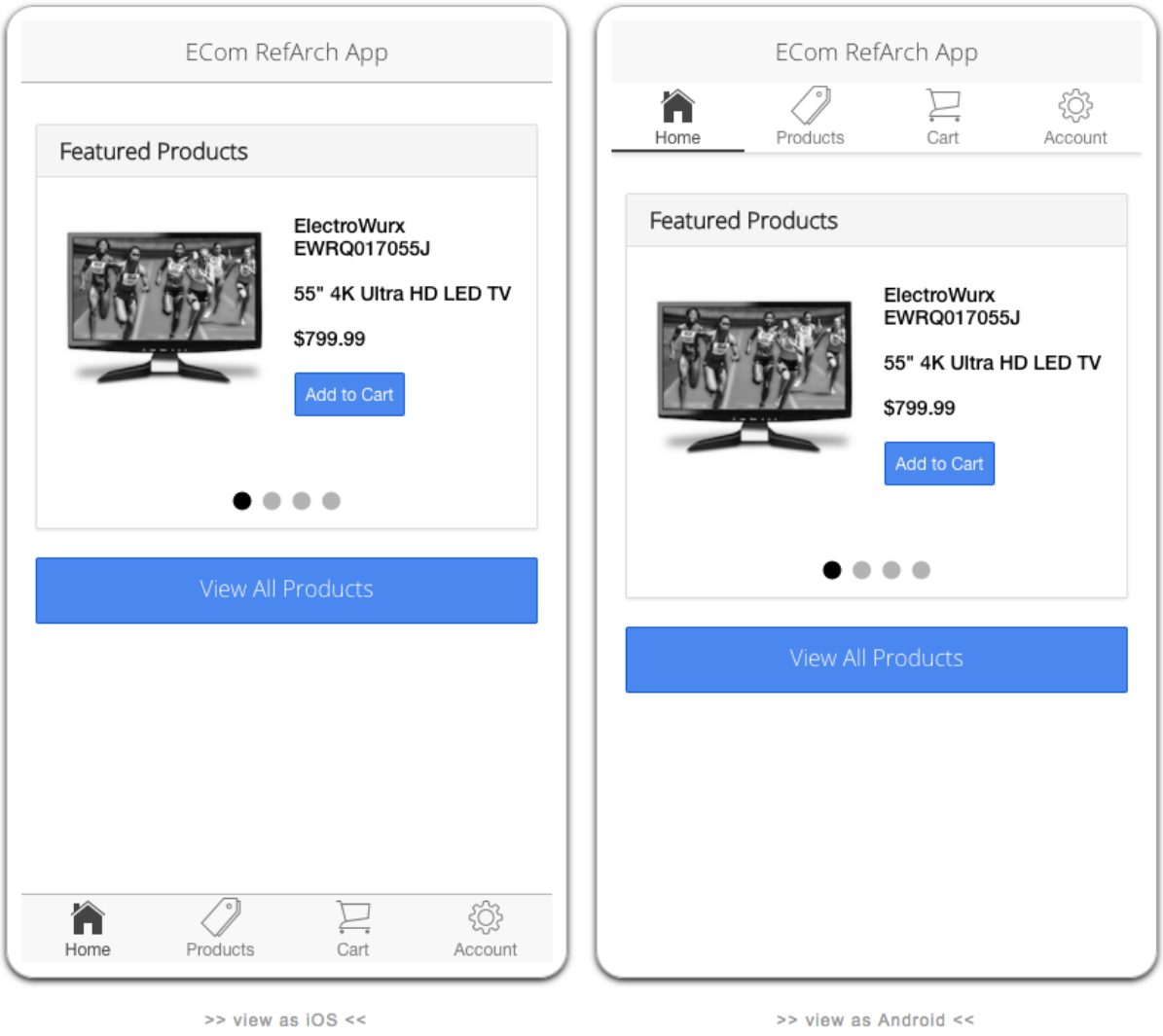
NOTE

The Grunt task runner tool is assumed to be installed locally prior to execution of the following steps. Further information regarding installation can be found at the GruntJS home page.

```
$ grunt serve
Running "serve" task

Running "browserify:www/main.js" (browserify) task
>> Bundle www/main.js created.

Running "clean:server" (clean) task

Running "connect:livereload" (connect) task
Started connect web server on http://localhost:9002

Running "watch" task
Waiting...
```

Once running, a browser tab or window will launch for the application. The user can also visit *lab.html* to see both iOS and Android renderings of the application:

Figure 5.6. Platform Preview



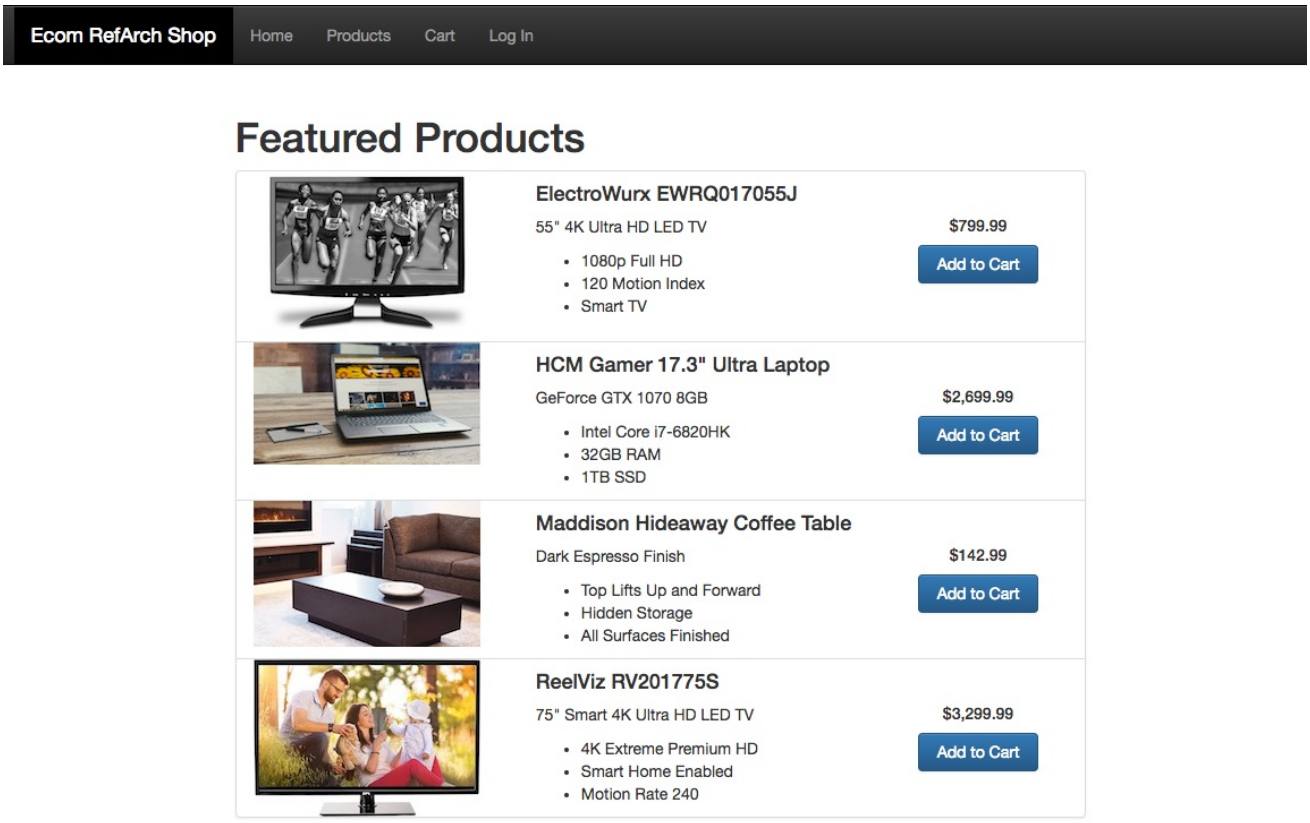
5.7. WEB PORTAL CLIENT APPLICATION

A complementary HTML5 web portal client application has been built atop Node.js using Angular and Bootstrap to demonstrate that cloud functionality of the Mobile Application Platform can also be made available via a singular cross-platform solution via native browser. The portal also demonstrates the capabilities of shared data usage and synchronization between both native mobile applications and web-based client applications.

Several similarities exist between the mobile and portal applications as they serve a similar end goal, demonstrating two different approaches to utilizing a Node.js and Angular-based application to serve as a cross-platform client solution. That being said, the focus of the portal application is primarily demonstration of desktop browser functionality and interactivity with the Mobile Application Platform. Further refined responsive design of the provided code could ultimately bridge the portal to support both mobile and desktop browser resolutions, however, focus within the example application was placed on demonstrating usage of larger displays to simplify navigation and functionality on the desktop.

While ultimately similar to the mobile app in function, some differences in form are readily apparent; product details and order details no longer require their own context and the featured list on the landing page now showcases several products at once with a greater level of detail.

Figure 5.7. Web Portal Application



5.7.1. SDK Configuration

SDK configuration of the client portal application is completed as follows:

5.7.1.1. application.js

Like the cloud app, the portal app is written on Node.js. Therefore, a similar initial configuration is required. Unlike the cloud app, Angular will be handling the custom routing responsibilities of the portal, so the *application.js* file is left intact to define only those configurations assumed by the Mobile Application Platform.

5.7.1.2. fhconfig.js

Just as with the mobile client application, contents of the *fhconfig.json* file are copied directly from the parameters given in the *Connection* component of the project.

5.7.1.3. Cloud Communication

The portal app, via the same SDK configuration, utilizes various *$fh.cloud(…)* calls to communicate with the cloud application.

5.7.1.4. Running Locally

The portal app can be ran locally by issuing the *npm install* followed by *grunt serve* commands.

5.7.1.5. Bootstrap UI Components

Where the Cordova-based mobile application used Ionic interface components to build the UI, the goal of the portal application was to build a solution not entirely tied to mobile platforms. Thus, Bootstrap was used alongside Angular to build a more browser-friendly solution that better utilizes large display space.

CHAPTER 6. CONCLUSION

This document has demonstrated the design, development and deployment of enterprise mobile applications and supporting services with **`Red Hat Mobile Application Platform`**.

New mobile and portal client applications, each hosting various new user-centric functionalities, were built to interact with and supplement a pre-existing e-commerce microservices back-end stack. By leveraging the Mobile Application Platform's Cloud API and SDK suite, features such as data caching, data synchronization, optimized performance, status monitoring, native device binary builds, and device-specific API interactivity were introduced to the software stack. These features boost the available feature set for users, and also harness the reliability of OpenShift Container Platform to ensure enterprise-level performance, management, and availability.

APPENDIX A. AUTHORSHIP HISTORY

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

APPENDIX B. CONTRIBUTORS

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

Contributor	Title	Contribution
John Frizelle	Consulting Software Engineer	Technical Assistance
Wojciech Trocki	Senior Software Engineer	Technical Assistance
Craig Brookes	Senior Software Engineer	Technical Assistance
Michael Burkman	Senior Consultant	Technical Assistance
Martin Murphy	Senior Software Engineer	Technical Assistance
Evan Shortiss	Senior Software Engineer	Technical Assistance, Content Review

APPENDIX C. ANSIBLE INVENTORY

```
[Nodes:children]
master
core
mbaas

[Nodes:vars]
ansible_ssh_user=rhmapAdmin
ansible_sudo=true

# Valid targets enterprise or dedicated
target="enterprise"

# Cluster hostname
cluster_hostname="rhmap.xxx.example.com"

# Customer subdomain name
domain_name="app"

# OpenShift Credentials
oc_user=rhmapAdmin
oc_password=PASSWORD
kubeconfig="/home/rhmapAdmin/.kube/config"

# Add values here if a HTTP proxy server is to be used
proxy_host=""
proxy_port=""
proxy_user=""
proxy_pass=""
# For proxy_url the syntax should be: http://<proxy-host>:<proxy-port> or
http://proxy_user:proxy_pass@<proxy-host>:<proxy-port>
proxy_url=""

# URL to be checked to test outbound HTTP connection. Must be whitelisted
if using a HTTP Proxy server
url_to_check="https://www.npmjs.com"

# Gluster config - Set if using Gluster FS for storage
# gluster_storage=true
# gluster_metadata_name=gluster
# gluster_endpoints=["10.10.0.55", "10.10.0.56"]

# In a mulit-master configuration, only one master node is required
[master]
rhmap-master1.xxx.example.com

[mbaas]
rhmap-node1.xxx.example.com
rhmap-node2.xxx.example.com
rhmap-node3.xxx.example.com

[core]
rhmap-node4.xxx.example.com
rhmap-node5.xxx.example.com
rhmap-node6.xxx.example.com
```

APPENDIX D. REVISION HISTORY

Revision 1.0-0	**August 2017**	**JA**