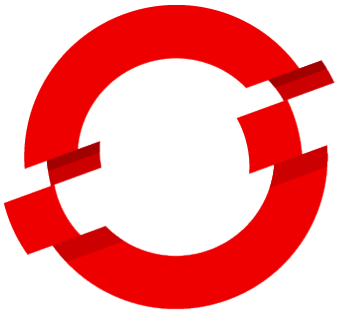


Red Hat OpenShift

OpenShift 4.3 DevEx Advanced Workshop

Jaroslaw Stakun

Version 1.0.0, April 21, 2020

Table of Contents

Introduction

This advanced workshop is designed to introduce the attendee to the additional and exciting developers features above and beyond the basic functionality offered by OpenShift.

This includes an introduction to the ODO command line for developers, an introduction to Tekton, the native Kubernetes pipeline functionality, an introduction to the concepts and implementation of serverless functionality via Knative, Camel-K and eventing.

Attendee details

Name:	
User ID (userX):	

What is Openshift

Red Hat® OpenShift® is a hybrid cloud, enterprise, secure Kubernetes application platform.

OpenShift is an Enterprise strength, secure implementation of the Kubernetes container orchestration project with additional tooling designed to make the lives of Developer and Administrators as easy as possible.

Links

- https://www.openshift.com/learn/what-is-openshift
- https://en.wikipedia.org/wiki/OpenShift
- https://www.openshift.com/products/container-platform
- https://bit.ly/2VH1jXw (the Web Console URL for the Workshop)
- https://api.cluster-cee-3fb4.cee-3fb4.example.opentlc.com:6443 (The Web Terminal URL for the workshop)

Contributors

Red Hat UK DSA Team:

- Phil Prosser
- Mark Roberts
- *'Uther'* Lawson
- Jonny Browning

Original asciidoc documentation for DevEx3.x by Phillip Kruger, Red Hat CEMEA

Workshop Pre-requisites

Setting up the UI and Terminal

The workshop was designed and tested on the Chrome browser and it is advised to avoid issues that this browser be used whenever possible

Open the browser and navigate to the console url https://bit.ly/2VH1jXw

Logon using the user provided by the course administrator (userx where x is a unique number for your session)

When logged on you will be presented with a screen listing the projects, of which there are currently none

Hit *Create Project*

For *Name* enter sandboxx, where x is your unique number

For the duration of the advanced workshop you will be working in this project

Display Name and *Description* are optional labels

Once the project has been created (you will be given a dashboard) change the mode of the UI from Administrator to Developer by clicking on the top left of the UI where it says *Administrator* and selecting *Developer*

On the Topology page it will state *No workloads found* - Click on *Container Image*

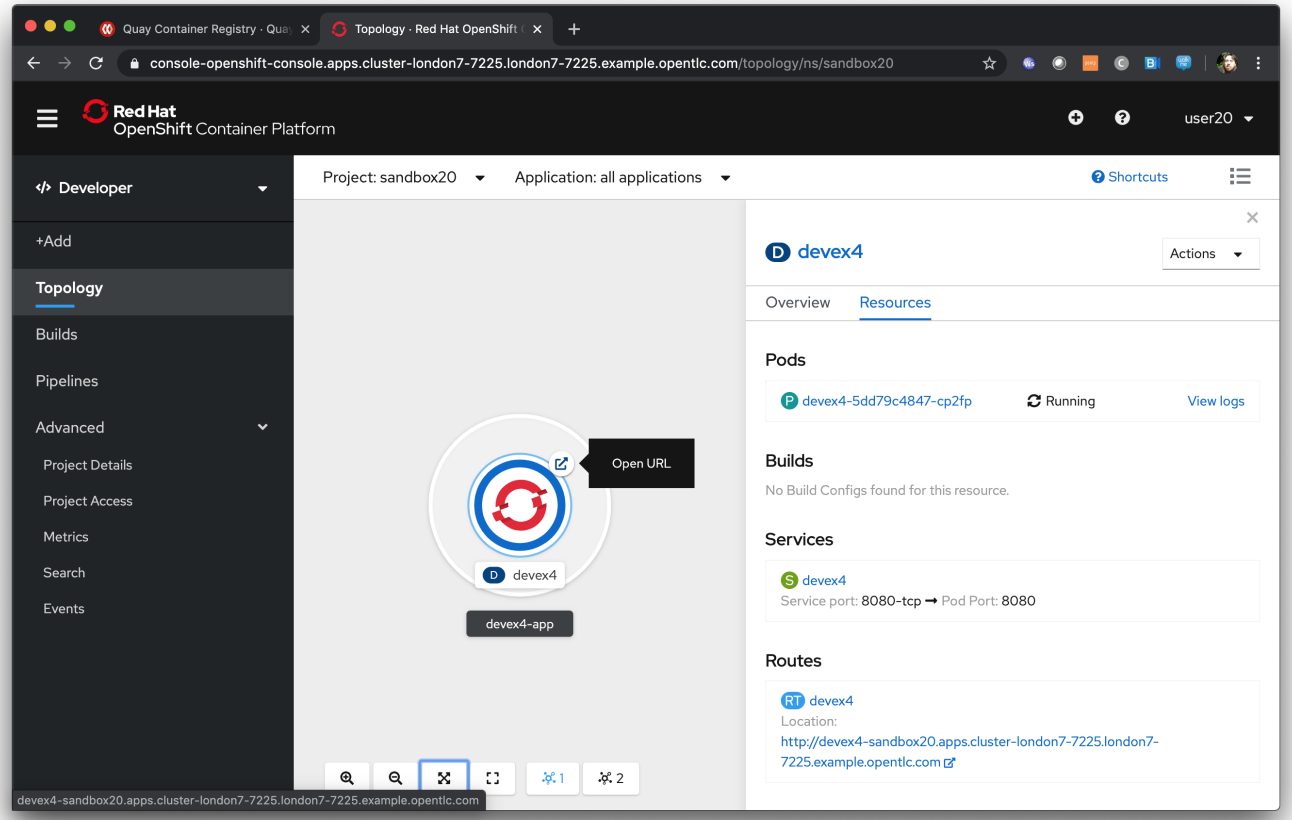
In *Image Name* enter *quay.io/ilawson/devex4*

Click on the search icon to the right of the text box

Leave everything else as default and scroll down to the bottom of the page. Hit *Create*

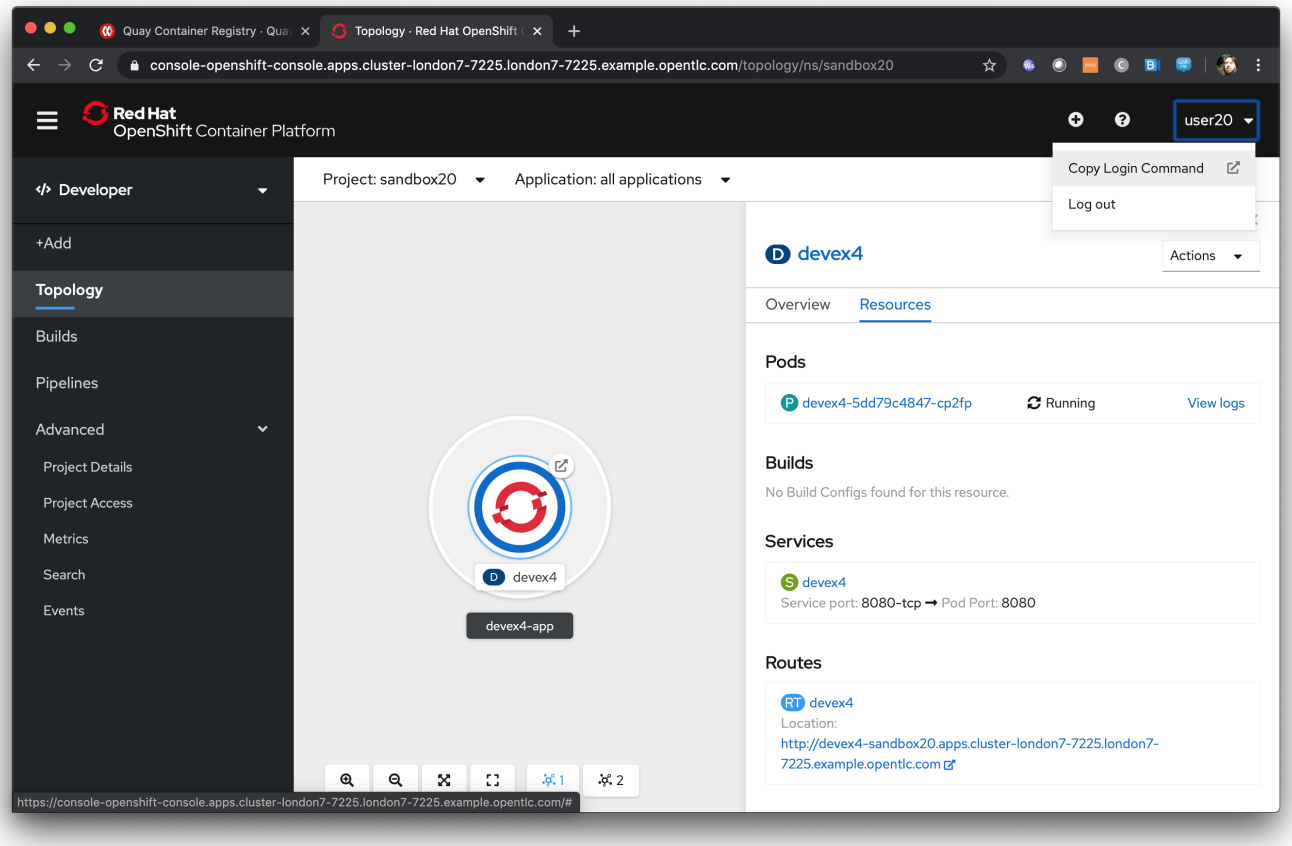
Wait for the ring around the application icon to change to dark blue - this indicates the application has been started

In the topology page click on the *Show URL* icon as shown below



This will open another tab with a command line in it - this is your terminal for the duration of the advanced workshop

Switch back to the UI and click on the userx displayed at the top right and select *Copy Login Command* as shown below



In the new tab that appears login with your userx (unique number instead of x) and password *openshift*

Click on *Display Token*

Copy the command given for *Log in with this token* - this may require using the browser *copy* command after highlighting the command

Close this tab and switch to the terminal tab - if you have closed the terminal tab go back to the UI and select the *Show URL* from the topology view in the Developer UI

Paste and execute the command

Press *y* to use insecure connections

The terminal should now be logged on - to check it try

```
oc whoami
oc version
```

The terminal should display your user for the first command and the client and Kubernetes versions for the second command

Finally, for use in the examples we will require some source material from the git repo for this workshop. To set this up type the following:

```
cd /workspace
git clone https://github.com/utherp0/workshop4
ls -alR
```

These files are the source of the Workshop and all the required local material for the course.

OpenShift DO

Author: Mark Roberts (feedback to mroberts@redhat.com)

Introduction

An overview of odo

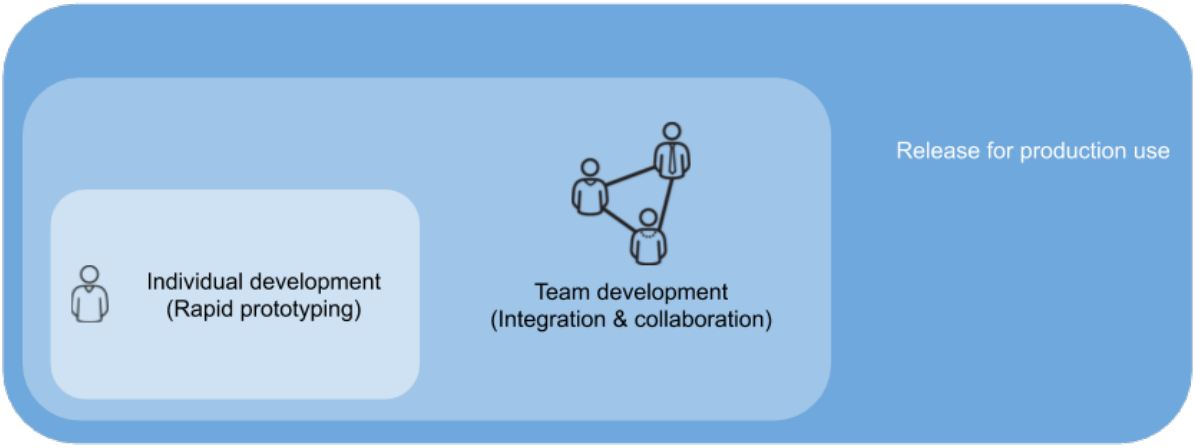
The OpenShift command line interface *oc* is a general purpose interface with a vast set of commands for both development and administrative purposes. The odo command is a more developer centric command line interface for users who simply want to build, deploy and run applications on OpenShift. There is definitely a place for both tools within the kitbag of an OpenShift user, and for fast iterations of edit - run - edit - run the odo interface is perfect.

The approach described in the application basics chapter used a GIT repository as the source of the content to be built and deployed. This is a excellent approach when multiple users are working together in a complex application performing frequent integration of source code in GIT and validating their combined efforts as the project progresses. Avoiding such frequent integration activity will make the code of each developer diverge storing up integration headaches in the future.

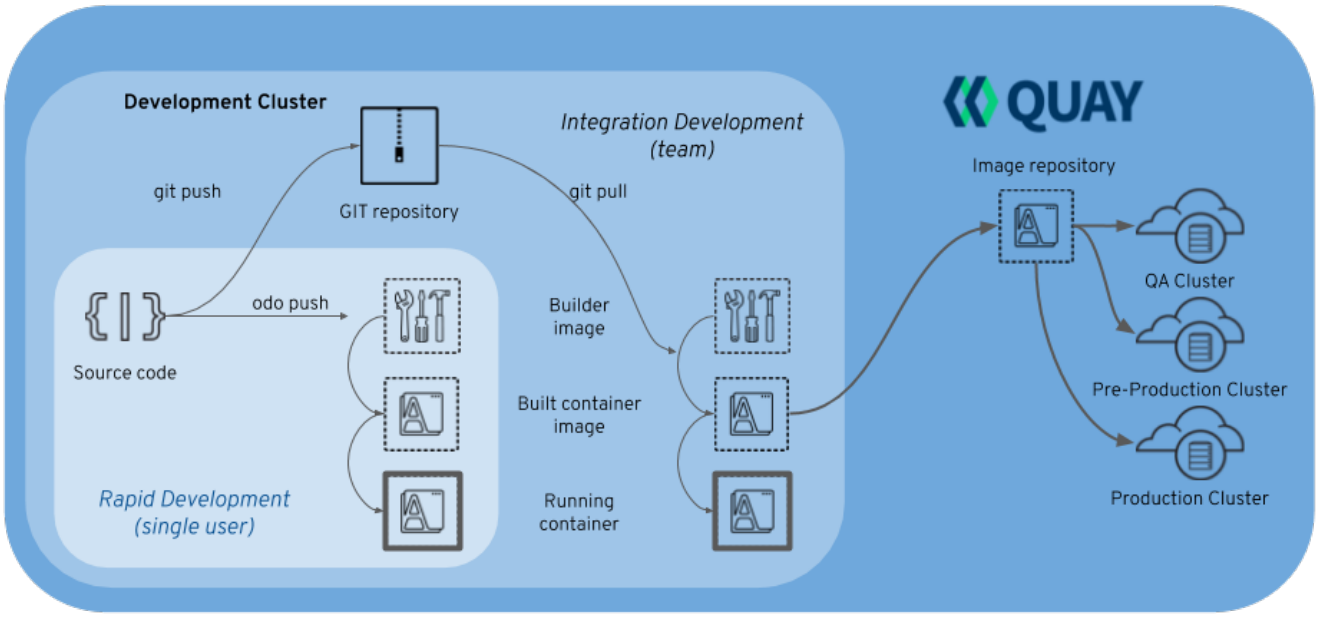
The ability of the OpenShift *Source-2-image* capability to identify the code within the GIT repository (from the options of Node.js, Java, Ruby, Perl, PHP, and Python), select a builder image, build the code and deliver a new container with the running application is a huge efficiency advantage for teams. However there are scenarios in which a single developer simply wants to get their code running as quickly as possible and which is where odo can be advantageous.

The principle of the odo command is that it can very quickly create a new project and use a *GIT like* syntax to enable a developer to push the code into OpenShift. The receiving OpenShift cluster will then decide how to build the code, perform the build and deliver a running container. When the developer wants to make changes they simply push their code and the build and deploy operations are repeated. The first push process may take around a minute, but after that each iteration will take only a few seconds.

The diagram below shows how the odo command can fit into a development cycle. The outline diagram shows that the odo command is appropriate for single developers working with a degree of isolation, the team development involves multiple developers integrating their code through a GIT repository and the final phase involves the use of multiple clusters for the deployment of applications through to production.



In more detail the diagram below shows how odo and oc / source-2-image work together to create a scaled development experience. In the single user phase the odo command is used to push the application code to a container in OpenShift. In the light blue box the source code is pushed to OpenShift in which the language specific builder image is used to create a new image containing the built code. This image is started to create a running container. When the developer needs to share the code with the work of others GIT is used as a vehicle for integration and to remove any code conflicts. This is the medium blue box in which the code is pushed to GIT and the OpenShift source-2-image capability is used to build a new running container. In the final phase shown in the darker blue box, the built container image can then be pushed to the Quay image repository to be stored securely ready for deployment to further environments such as pre-production and production.



Installing

The odo command line interface is available for download from here : https://github.com/openshift/odo

If you haven't already please follow the instructions in the pre-req chapter which explains the creation and setting up of the terminal application. For this chapter you will be using that application which has the odo command pre-installed for you

Switch to the terminal application tab of the browser as dsecribed in the pre-req chapter

To use the command simply type *odo* and you will see help regarding the objects and to get help on a specific object use *odo* <*object*> *--help*.

odo command set

The odo command has the following subcommands :

Commands:

```
app         Perform application operations (delete, describe, list)
catalog     Catalog related operations (describe, list, search)
component   Manage components (create, delete, describe, link, list, log, push, unlink,
update, watch)
config      Change or view configuration (set, unset, view)
debug       Debug commands (port-forward)
preference  Modifies preference settings (set, unset, view)
project     Perform project operations (create, delete, get, list, set)
service     Perform service catalog operations (create, delete, list)
storage     Perform storage operations (create, delete, list)
url         Expose component to the outside world (create, delete, list)
```

Utility Commands:

```
login       Login to cluster
logout      Log out of the current OpenShift session
utils       Utilities for terminal commands and modifying odo configurations (terminal)
version     Print the client version information
```

Component Shortcuts:

```
create      Create a new component
delete      Delete component
describe    Describe component
link        Link component to a service or component
list        List all components in the current application
log         Retrieve the log for the given component
push        Push source code to a component
unlink      Unlink component to a service or component
update      Update the source code path of a component
watch       Watch for changes, update component on change
```

The most common are used within this workshop but feel free to experiment with any of the above commands.

Logging in

At this point if you are using the terminal as explained in the pre-req chapter you will already be logged into the system. This can be checked by typing the following:

```
oc whoami
odo project list
```

This should result in your user name being displayed followed by the sandboxX project being shown by the result to the odo command

If you do need to login to odo outside of this workshop use the *odo login* command with appropriate parameters

The odo command will evolve over time and it performs a check to see if you have the latest version each time it is used. If there is an update available you will be given a prompt to update from a given URL

To see which version of odo you are using enter the command:

```
odo version
```

Examining source assets

The pipeline assets are located in a GIT repository that has been preinstalled in the terminal. Using your terminal window check the assets using the commands below

```
cd /workspace/workshop4/attendee/slave

ls -al
```

You will see that the directory only has the source file slave.js and the package file called package.json.

Create, push source & run cycle

Create a new project using the odo command replacing X with your user number below

```
odo create nodejs node-app-slave
```

TIP:The syntax of the above command is *odo create* <*component-type*> <*component-name*> *--project* <*project-for-the-component*>

The result of running this command is simply the creation of a .odo directory containing a config.yaml file. The file contains the desired state for the application in OpenShift and is only committed to OpenShift and acted upon by OpenShift when the user issues the command *odo push*. Examine the config.yaml file with the command below

```
---
cat .odo/config.yaml
---
```

Create a route for the application by using the command below

```
odo url create node-app-slave
```

Examine the contents of the .odo/config.yaml file again and you will see that new content has been added

```
cat .odo/config.yaml
```

Now push the configuration to OpenShift by using the command below

```
odo push
```

The output from the above command is shown below

```
Validation
 ✓  Checking component [113ms]

Configuration changes
 ✓  Initializing component
 ✓  Creating component [348ms]

Applying URL changes
 ✓  URL node-app-slave: http://node-app-slave-app-master-slave-odo.apps.cluster-london-a29c.london-a29
c.example.opentlc.com created

Pushing to component node-app-slave of type local
 ✓  Checking files for pushing [568214ns]
 ✓  Waiting for component to start [1m]
 ✓  Syncing files to the component [15s]
 ✓  Building component [22s]
 ✓  Changes successfully pushed to component
```

The application has started up and will be running at the URL indicated in the output above. Copy the URL from your command window and paste it into a new browser tab. You should see an output similar to that shown below

```
Hello - this is the simple slave REST interface v1.0
```

Now make a small change to the comment in the source code of the slave.js file to change the line shown below

```
response.send('Hello - this is the simple slave REST interface' + versionIdentifier);
```

Change the response to the following

```
response.send('Hello - MODIFIED and pushed with ODO' + versionIdentifier);
```

Now use odo to push the changed source to OpenShift

odo push

The code still needed to be pushed to the component, but the final stage of building the component is much faster. Refresh the browser window showing the application output and you will see your code change. The edit - push - test cycle is as simple as that

odo watch

The odo process also has a *watch* facility that allows you to force odo to constantly watch for source code changes and push them immediately. Open another instance of the terminal application by pointing a new tab in the browser to the route to the terminal application.

In the new terminal tab enter

```
cd /workspace/workshop4/attendee/slave
odo watch
```

The command window should report : *Waiting for something to change in <current-working-directory>*

Switch back to your other terminal window and make another change to the source code, similar to the change above. After saving the edit switch to the terminal window in which you typed *odo watch* and observe that a new push of the code to OpenShift has taken place

The window with the watch command running will report:

```
File <path-to-source>/slave.js changed
Pushing files...
 ✓  Waiting for component to start [73ms]
 ✓  Syncing files to the component [11s]
 ✓  Building component [4s]
```

Refresh the browser widow showing the application output and you will see your code change

odo is clearly a very fast way to go from code to running your application without having to install

tools and frameworks on your laptop

Finally, lets clean up the project by typing

```
odo delete node-app-slave
```

This will delete the application from the project

Camel K on Openshift

Author: Phil Prosser (feedback to pprosser@redhat.com)

Introduction

Apache Camel K is a lightweight integration framework built from Apache Camel that runs natively on Kubernetes and is specifically designed for serverless and microservice architectures.

Users of Camel K can instantly run integration code written in Camel DSL on their preferred cloud (Kubernetes or OpenShift).

The purpose of this lab is to show you how easy it is to build, deploy, and delete Camel K integations using very simple examples. It is not the goal of this lab to demonstrate the integration capabilities of Camel K.

We are going to use a new project to create our Camel K integrations

This workshop requires you to be logged on to both the OpenShift console and the terminal as described in the pre-requisites. If you do not have both tabs logged in please follow the instructions before continuing

Check the project is ready to use

In the browser based terminal window, check you are still logged on and using the correct project by typing:

```
oc whoami
oc project
```

If the response from the commands indicates that you are not userX (where X is your assigned user number) and not using the project sandboxX please repeat the commands in the pre-requisites.

Camel K and the Operator Lifecycle Manager

Camel K and the Operator Lifecycle Manager

Camel K uses the Operator Lifecycle manager, this means that new Custom Resource Definitions (CRDs) will be added to Openshift. These CRDs will extend the Openshift data model and allow Camel K to be managed using the standard 'oc' command. Installing operators requires a higher cluster privilage. All the required operators for the workshop should have been pre-installed for you

The examples required for this workshop have been pre-created as part of the terminal. In the terminal type:

```
cd /workspace/examples
ls
```

These are the example files we will be using.

To allow a developer to easily interact with an Openshift cluster, Camel K has it's own command line interface. The cli is called *kamel* and is preinstalled in the terminal app

Before we can create an integration, we need to add a Camel K *IntegrationPlatform*.

In the terminal window, type

```
cd /workspace/workshop4/camelfiles/camelkplatform/
oc apply -f integrationplatform.yaml
```

now type

```
oc get integrationplatform
```

The output should look similar to below

```
NAME      PHASE
camel-k   Ready
```

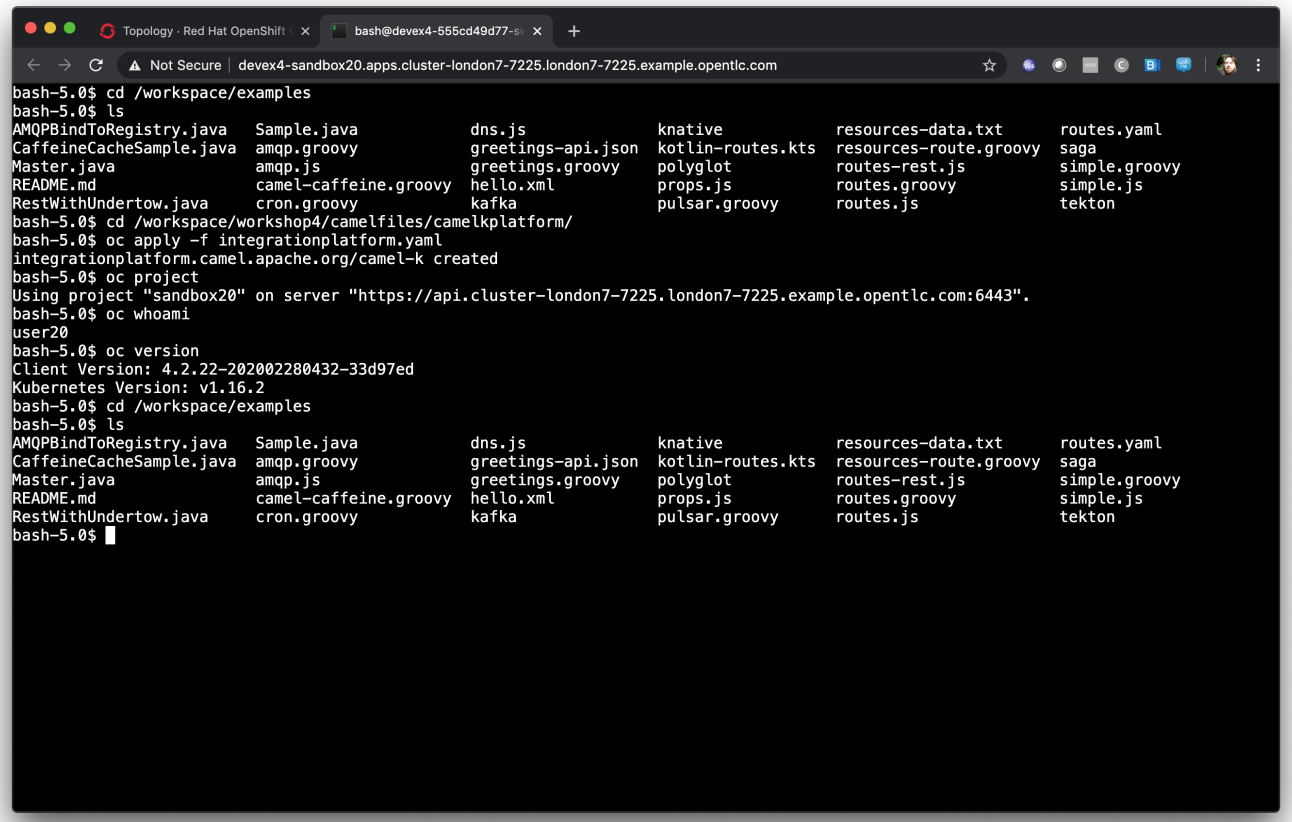
Once the "Phase" says "Ready", you can continue

You should now have all the pieces you need to start creating and deploying lightweight Camel Integrations to Openshift.

Now enter the following commands:

```
oc project
oc whoami
oc version
cd /workspace/examples
ls
```

You should see an output similar to the one shown below:



Deploy a Camel K Integration

Firstly, let's start by deploying a simple integration, type

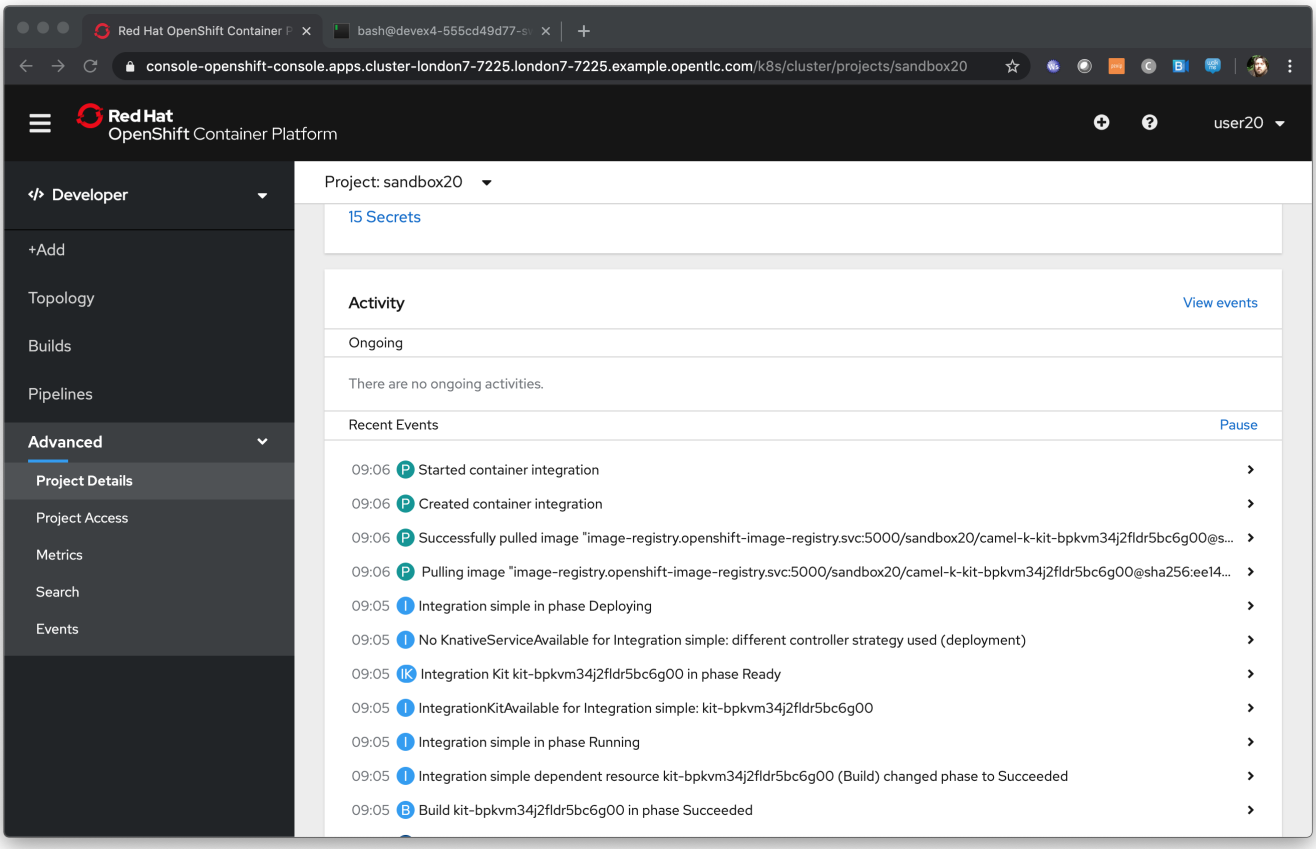
```
kamel run simple.groovy
```

The first time you deploy an Integration, it will take a few minutes. The operator manages all dependencies required by the Integration and downloads these from the Red Hat repository on demand. Once downloaded, the operator caches dependancies therefore subsequent deployments are significantly quicker. If you want to see what's happen, in the terminal window type *oc get pods*, you will see that there are builds running

To see what is going on go to the console UI in the Browser, make sure you are switched to *Developer view*

Click on *Advanced/Project Details*

If your screen is wide enough the Activity pane will appear on the right hand side. If not, scroll down to the *Activity* pane - this shows the events and actions currently occuring within the project. It will look similar to this:



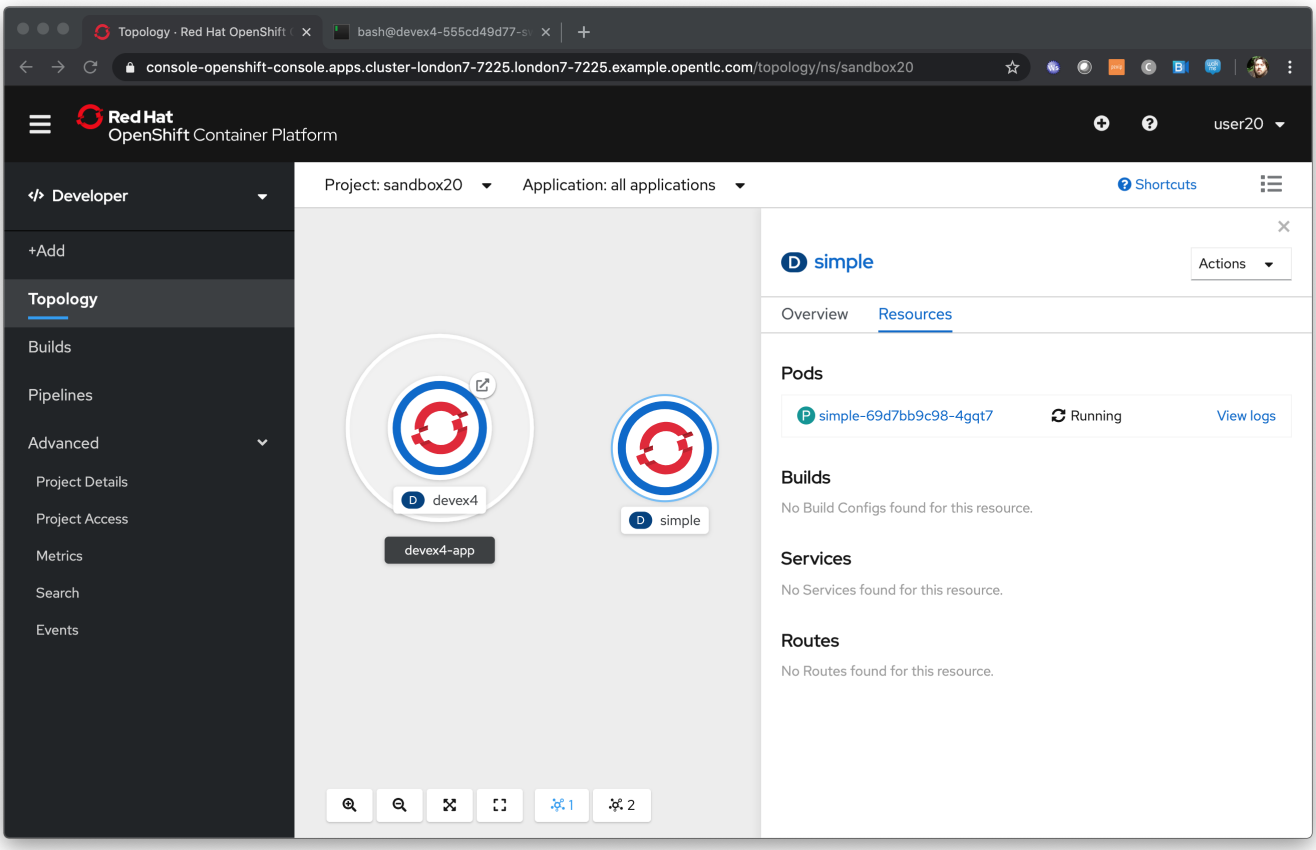
Switch back to the project Topology view by clicking on *Topology*

Once the deployment is complete, you will see *Simple* deployed on the topology view

Click on *Simple* (this is your integration), and you can see what's going on.

Once the pod has a dark blue ring around it, it is running, as per the screenshot below. The devex4

application is your terminal application and can be ignored for the duration of this chapter

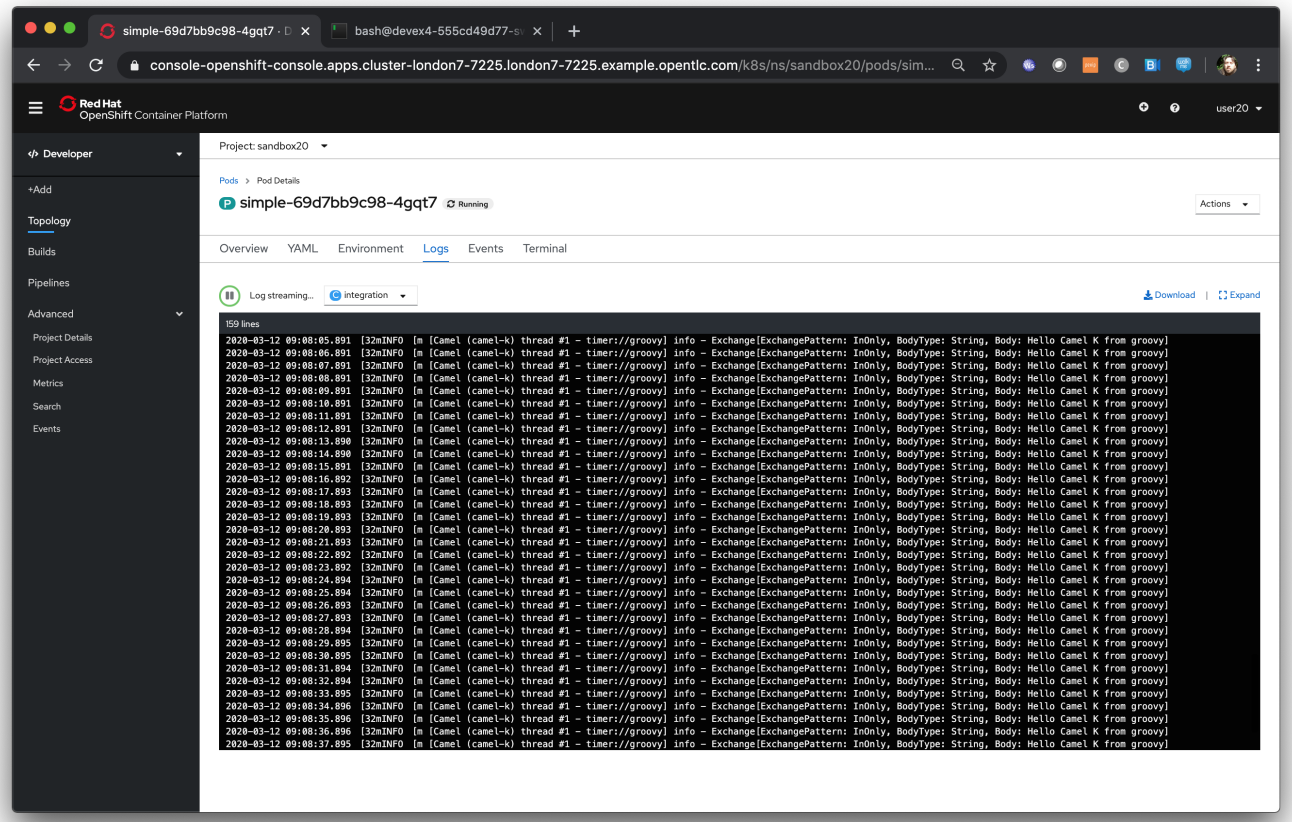


If you haven't already, click *inside the circle* to open the overview window

Click on *Resources*

There will be one Pod running with a name similar to simple-xxxxxxx-yyyyy (randomly generated). Next to it will be an indicator that it is running. Next to that is a clickable point labelled *View Logs*. Click on this.

The output of the log should look as shown below:



The integration is a simple timer that triggers every 1 second and writes to the log file.

In the Terminal Browser window type

```
oc get integrations
```

You should now see an integration called *simple* in the list similar to this:

```
NAME     PHASE     KIT                          REPLICAS
simple   Running   kit-bpj4ns3tvn0va7c7gs9g   1
```

In the Terminal browser window type

```
oc describe integration simple
```

If you scan to the top of the output you will see some code in the *from* component that represents the integration's behaviour

We will now make a change to the integration

In the browser terminal window

```
vi simple.groovy
```

You will see the text - *Hello Camel K from ${routeId}* in the code definition of the integration

Change the text to the following by pressing [ESC] then I to insert:

```
'Hello Camel K from ${routeId}. Added some more text'
```

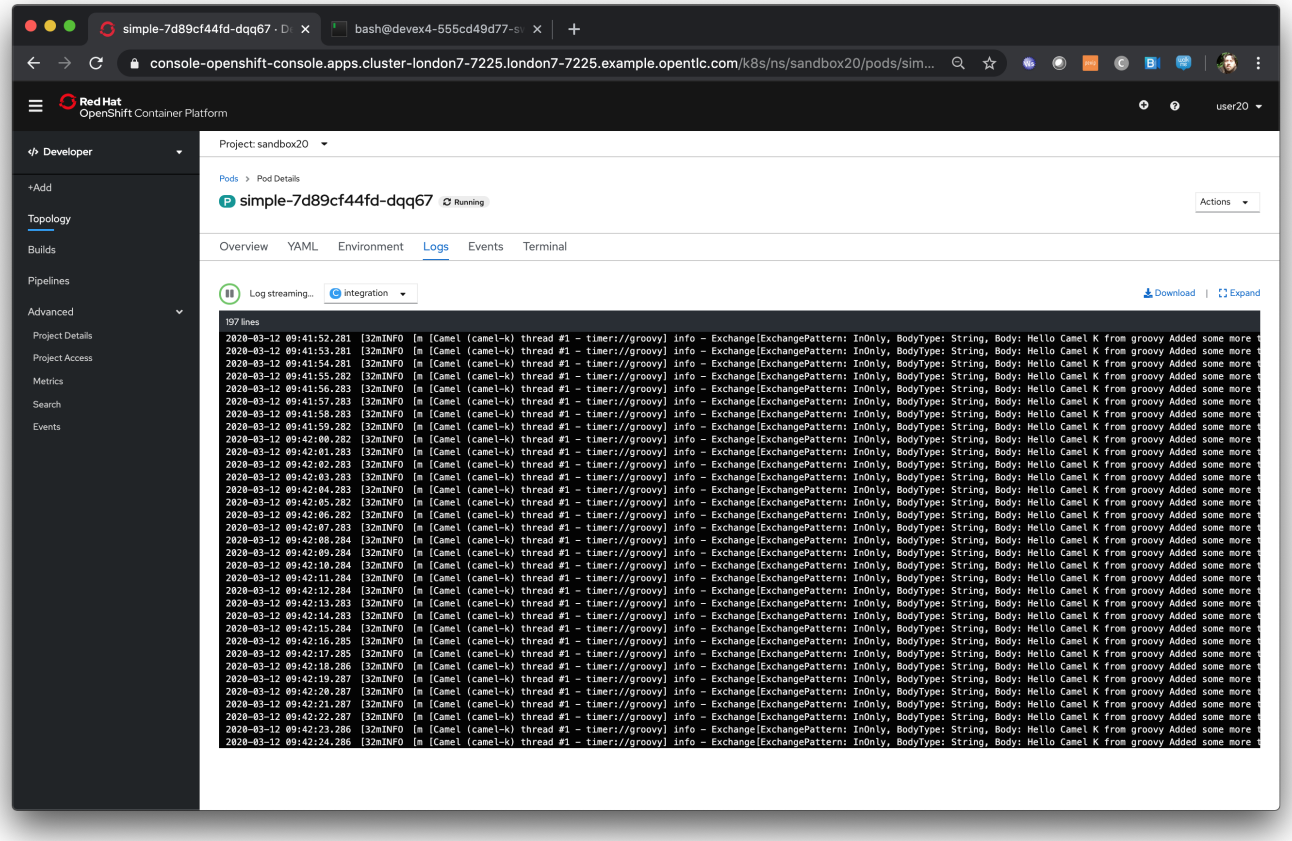
To save the change now hit [ESC]:wq[RETURN]

Now, you need to deploy this integration to Openshift again to test by typing:

```
kamel run simple.groovy
```

If you are quick enough (you need to be really quick) switch back to the OpenShift console and hit *Topology*, you'll see the integration doing a re-deployment

Look at the log file again (as above) to make sure the change has been deployed



Deploy Camel K in Developer mode

While the process of redeploying is simple, it isn't very developer friendly. The *kamel* cli has a developer friendly "hot deploy" mode that makes this experience much better

First Delete the integration.

There are 2 ways you can do this in the Terminal Browser window (your choice). Either use the "kamel" cli:

```
kamel delete simple
```

Or use the Openshift cli:

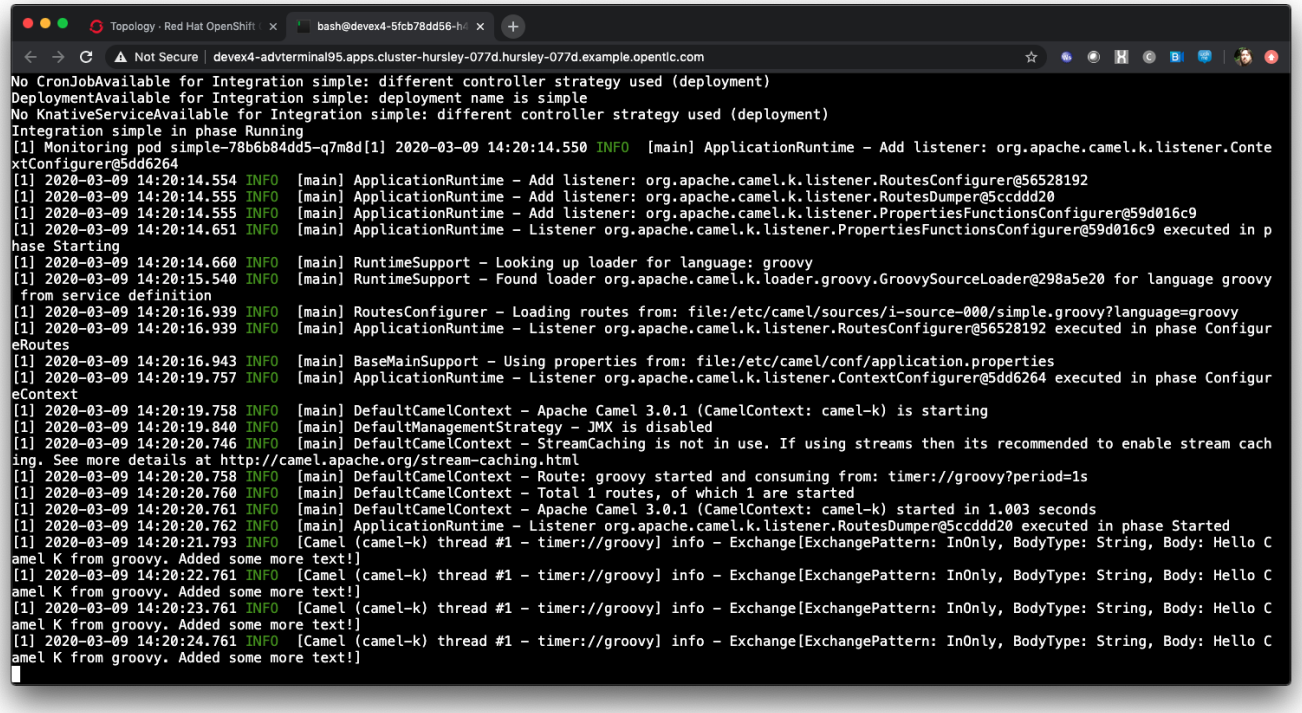
```
oc delete integration simple
```

This is the great thing about CRDs, you can use the normal Openshift cli to managed the custom data model (integrations in this case)

To deploy the integration in developer mode, type:

```
kamel run simple.groovy --dev
```

You will see the deployment phases logged on the screen, followed by the log outputting automatically from the integration pod, useful for a developer to see what's going on. The output should look similar to the screenshot below



For the next exercise, you will need 2 terminal windows. Go to the OpenShift Console, which should be on the Developer view. Click on Topology if the Topology window is not currently in focus. Click on the URL icon at the top right of the *Devex4* application as you did to originally open the terminal. This will open another Terminal for you to use.

In the first terminal tab, which will be the one furthest to the right, you will notice that the terminal window is outputting the log of the active and running integration

In the **new** terminal now type:

```
cd /workspace/examples
```

Make another change to the text in “simple.groovy” by following the same instructions above - make sure the text outputted is different and that you save it as described above

Once you have saved the changes, go back to the browser terminal tab outputting the log

Switch to the original output tab. The integration will shutdown and restart with the new code and new text

You should see that the changes have been automatically applied to the running integration, without the need to redeploy

Go back to the browser terminal that’s outputting the log, press ‘ctrl c’

Look at the Topology view in the Openshift console(or *oc get integrations* in the terminal)

The integration should have been deleted and no Pods should now be running (other than the Terminal pod), just like a developer would see by pressing *ctrl c* on a Java application running on their laptop

Close down one of the terminal window tabs so you only have one terminal and the OpenShift console

If you followed the lab, the Integration should be gone, however, lets make sure

clean up the project

In the terminal window, type

```
kamel get
```

If there is no Integration runninng then proceed to the next lab of your choice

If an Integration is running, then please delete it by typing

```
kamel delete simple
```

OpenShift Pipelines - Tekton

Author: Mark Roberts (feedback to mroberts@redhat.com)

Introduction

For many years teams have created automation pipelines to build, unit test, analyse and deploy applications in a variety of tool-chains. Some such tool-chains (Jenkins for one) pre-date the popularisation of containers and are now being questioned for their fit for the delivery of cloud native applications. While we are at pains not to criticise Jenkins (we are big users and fans) there is scope to look at alternative approaches for cloud native applications.*

OpenShift Pipelines are based on the Tekton open source project and they provide a Kubernetes style, resource based approach to the construction of pipelines. Tasks are defined as small units of operations (typically 10 to 15 lines of YAML) and such tasks are grouped together into taskruns or pipelines for execution.

A new command line exists for the execution of Tekton commands and a graphical user interface can be added to the cluster to provide a simple way to run tasks and to observe running or completed tasks.

The Tekton project can be found here : https://tekton.dev/

Further documentation on the specifics of Tekton can be found on the github site here : https://github.com/tektoncd/pipeline

Tasks

Download pipeline assets

As part of the pre-requisites you should have cloned the required git repo and created the base project for all the chapters - if you haven't please go back to the pre-requisites and follow the instructions

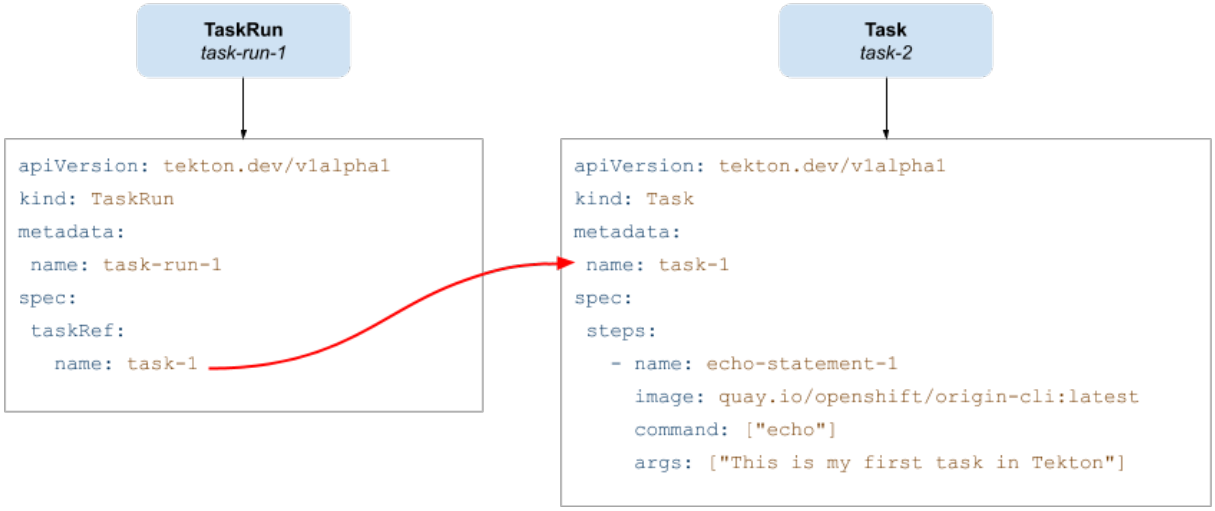
Check the required assets for the workshop are available in your terminal image using:

```
cd /workspace/workshop4/attendee/tekton/
ls -al
```

You should now see a list of yaml files that are used during the remainder of this chapter.

Simple task creation and execution

In the simplest format a task can be executed by a taskrun object. The diagram below shows a simple taskrun calling a single task



Tekton command line

A new command line utility is used to manage the Tekton command line. The commands cover the various different pipeline objects such as task, taskrun, pipeline, pipelinerun, resources, cluster tasks and conditions

The Tekton command line is downloaded from here : https://github.com/tektoncd/cli

The pre-installed terminal application created in the pre-requisites already has the command line installed

To use the command simply type *tkn* and you will see help regarding the objects and to get help on a specific object use *tkn <object> --help*

To execute the above pipeline use the following commands to create the pipeline objects. Note that when creating the taskrun objects the associated task(s) will run immediately.

```
oc create -f task-1.yaml

oc create -f taskrun-1.yaml

tkn taskrun ls
```

The response from the last command will display a line similar to the following:

```
NAME          STARTED          DURATION      STATUS
task-run-1    9 seconds ago    ---           Running(Pending)
```

Repeat the final command a few times until you see it change to be similar to the following:

```
NAME          STARTED           DURATION      STATUS
task-run-1    3 minutes ago    21 seconds    Succeeded
```

The result of running the task can then be viewed by executing the following command:

```
tkn taskrun logs task-run-1
```

As of Tekton 0.8.0 there is an outstanding bug involving logging that may cause extraneous log messages to appear. Please ignore these, they are fixed at the next release of Tekton

```
[echo-statement-1] This is my first task in Tekton

[echo-statement-2] ------------------------------------------------------------
[echo-statement-2]   - This is a multi-line comment
[echo-statement-2]   - This is useful as a separator but each line has
[echo-statement-2]   - the title repeated next to it using different colours
[echo-statement-2]   - which helps with the identification of different tasks.
[echo-statement-2]  ------------------------------------------------------------
[echo-statement-2]
```

Note that the command response above uses different colours for each block of command result titles in the [ ] brackets. This helps to differential between the response for each step.

Pipelines

Pipelines are used to manage the execution of a series of tasks within a pipeline. The example pipeline below is used to execute the Tekton tasks : task-1, task-2 and task-3. Remember that each Tekton task can have multiple steps within it.

```
apiVersion: tekton.dev/v1alpha1
kind: Pipeline
metadata:
  name: pipeline-1
spec:
  tasks:
  - name: task-1
    taskRef:
      name: task-1
  - name: task-2
    taskref:
      name: task-2
  - name: task-3
    taskref:
      name: task-3
```

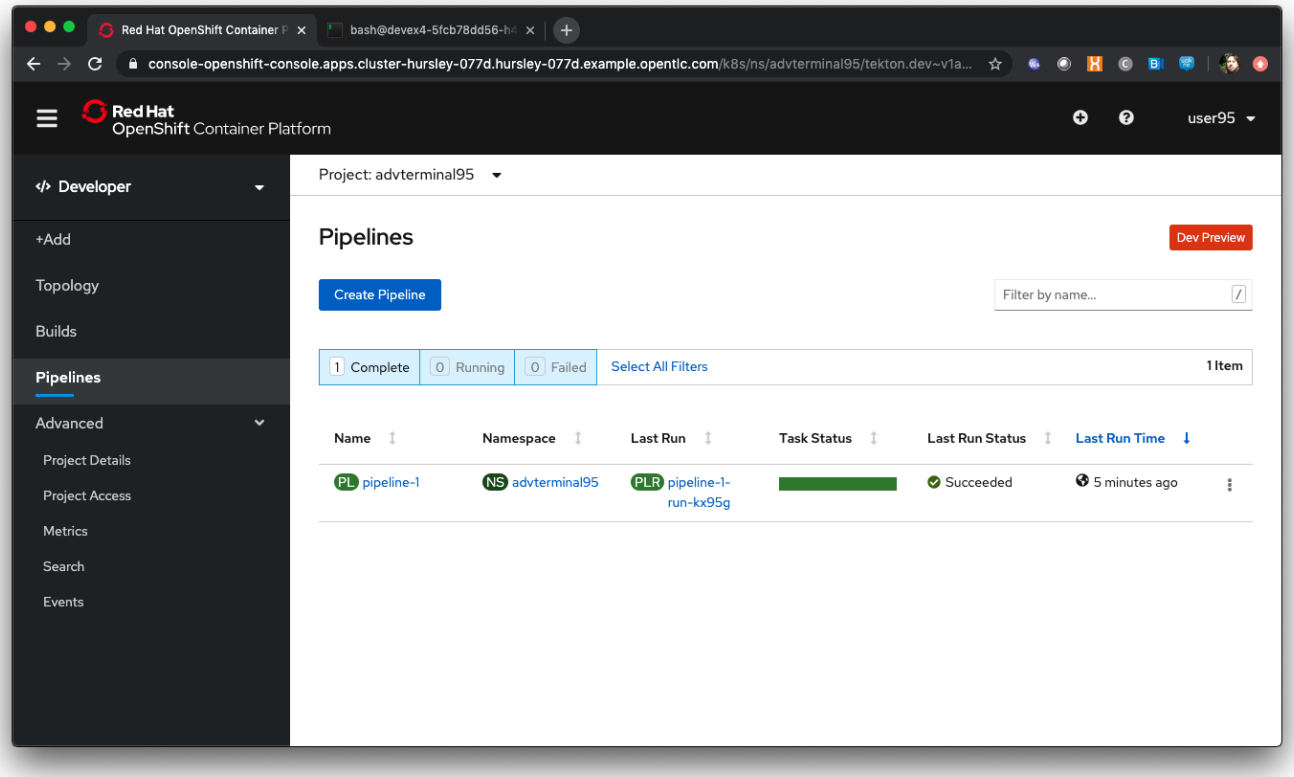
Before executing this pipeline create the required additional resources with the commands :

```
oc create -f task-2.yaml

oc create -f task-3.yaml

oc create -f pipeline-1.yaml
```

Execute the pipeline with the following Tekton task

```
tkn pipeline start pipeline-1
```

Once run switch back to the OpenShift console. If you select the Developer view you will find there is a menu option on the right hand side labelled *Pipelines*. Select this option and the screen should look like this:



Once the pipeline is indicated to have finished, go back to the terminal and enter the command suggested by the response to the tkn command - it will look similar to this:

```
tkn pipelinerun logs pipeline-1-run-kx95g -f
```

Note that the -n parameter is optional, it states the namespace to look in for the pipelinerun and we are running in a single namespace

Enter the command as provided by the tkn command and the response should look something like this:

Again, note there is a bug with Tekton 0.8.0 that may inject extraneous log errors, please ignore these.

```
Pipelinerun started: pipeline-1-run-ffxsk
Showing logs...
[task-2 : what-directory] /workspace

[task-2 : describe-command] ------------------------------------------------------------
[task-2 : describe-command]   - Openshift oc command line example
[task-2 : describe-command]  ------------------------------------------------------------
[task-2 : describe-command]

[task-2 : oc-version] Client Version: unknown
[task-2 : oc-version] Kubernetes Version: v1.14.6+76aeb0c

[task-3 : echo-statement-3] echo - statement 3
[task-1 : echo-statement-1] This is my first task in Tekton


[task-3 : echo-statement-4] echo - statement 4

[task-1 : echo-statement-2] ------------------------------------------------------------
[task-1 : echo-statement-2]   - This is a multi-line comment
[task-1 : echo-statement-2]   - This is useful as a separator but each line has
[task-1 : echo-statement-2]   - the title repeated next to it using different colours
[task-1 : echo-statement-2]   - which helps with the identification of different tasks.
[task-1 : echo-statement-2]  ------------------------------------------------------------
```

There may be an issue in the order of the execution above. The order of the pipeline expected is different to the order observed:

```
   Expected              Actual
task 1 - step 1       task 2 - step 1
task 1 - step 2       task 2 - step 2
task 2 - step 1       task 2 - step 3
task 2 - step 2       task 3 - step 1
task 2 - step 3       task 1 - step 1
task 3 - step 1       task 3 - step 2
task 3 - step 2       task 1 - step 2
```

In some pipelines the order of execution may not matter but if it does the order can be managed by the addition of the *runAfter* directive to a specific task as shown in the update to the pipeline-1 pipeline shown below:

```
apiVersion: tekton.dev/v1alpha1
kind: Pipeline
metadata:
  name: pipeline-1
spec:
  tasks:
  - name: task-1
    taskRef:
      name: task-1
  - name: task-2
    taskref:
      name: task-2
    runAfter:
    - task-1
  - name: task-3
    taskref:
      name: task-3
    runAfter:
    - task-2
```

Make the above changes to the pipeline-1.yaml file by using vi:

```
vi pipeline-1.yaml
```

Press [ESC] then i to edit/insert, make the changes to the file, then press [ESC] and type :wq[RETURN] to save the changes

Now replace the existing pipeline using the following commands:

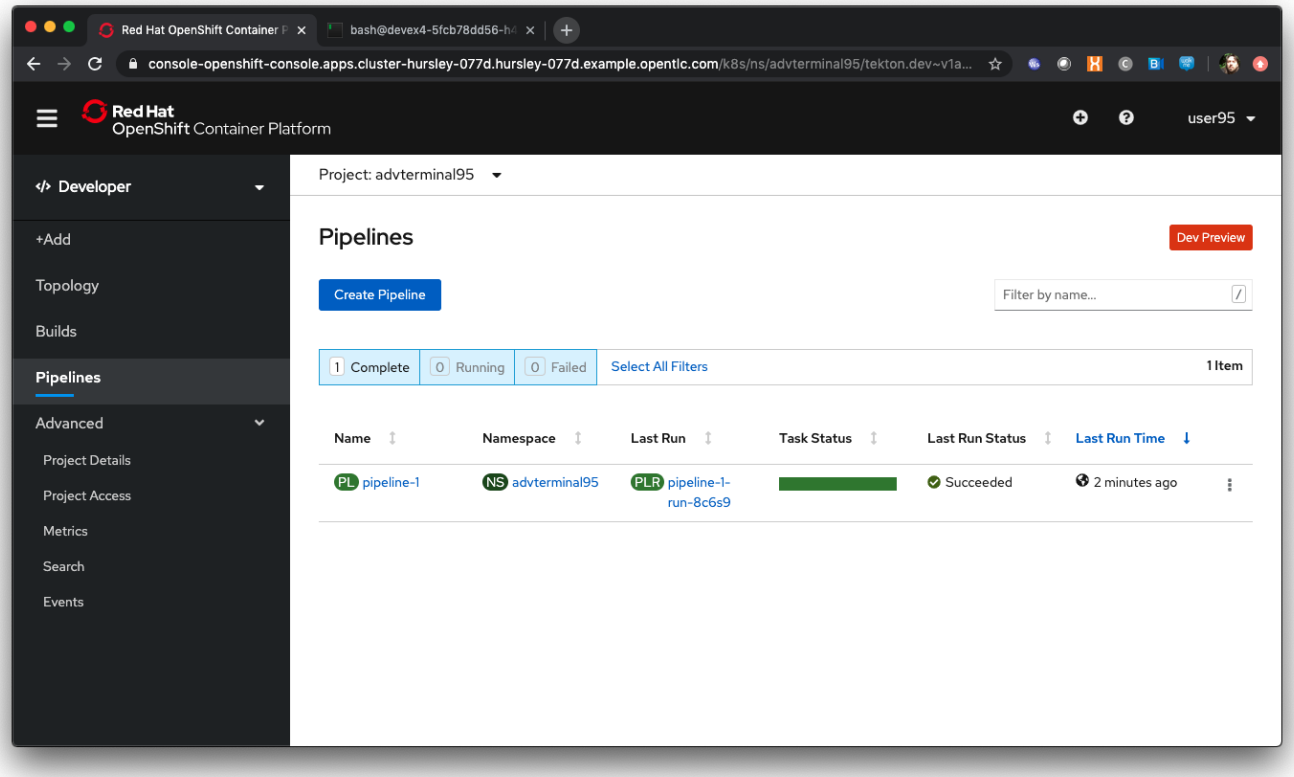
```
oc delete pipeline pipeline-1

oc create -f pipeline-1.yaml

tkn pipeline start pipeline-1
```

As soon as you enter the last command switch back to the console and watch the pipeline complete, note the synchronous order of the steps.

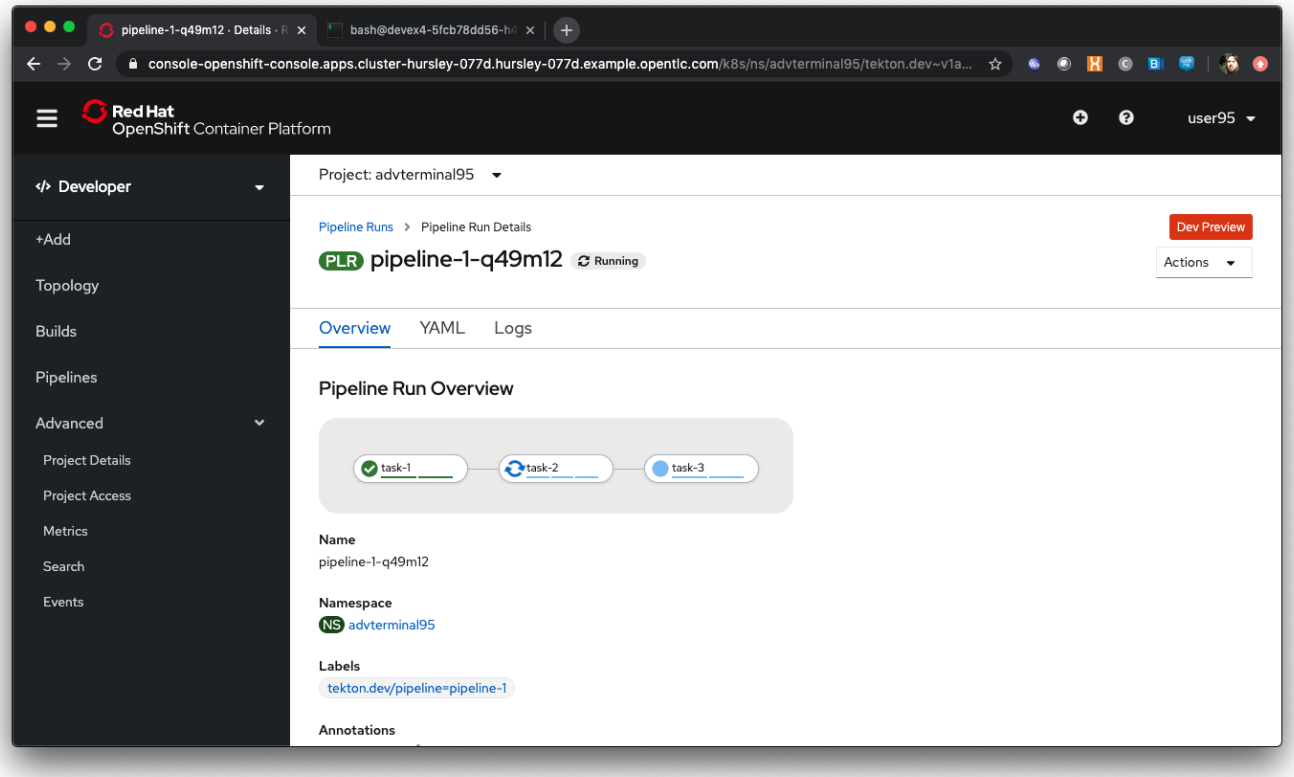
Viewing pipelines through the Web UI

In the OpenShift console you will see the pipeline recently created and it will show a green bar to the right indicating the previous successful execution of the pipeline, as shown below. Note that the green bar will display dark blue sections for running tasks, light blue sections for pending tasks, green for

completed and red for failed.



From the three dot menu on the right hand side it is possible to start a run of the pipeline. Do this now and watch as the screen changes to show the details of the pipeline run as shown below:



Each block can be clicked on to show the details of the steps within the task. Experiment with the different screens to look at the details of the running or completed tasks.

Task inputs

There will be scenarios where it is necessary to provide specific parameters to a pipeline process and the underlying tasks that the pipeline call.

There are two mechanisms for getting specific values into tasks :

- parameters - used to provide specific values to tasks at runtime. If a parameter is declared it must either have a default value defined within the task or it must have a value supplied from a calling taskrun or pipeline run.
- pipeline resources - a reference to a defined resource object that can be accessed by a Tekton pipeline. If a resource is referenced by a task then the resource must exist unless it has been defined as an optional resource in the task definition.

Pipeline Resource Types

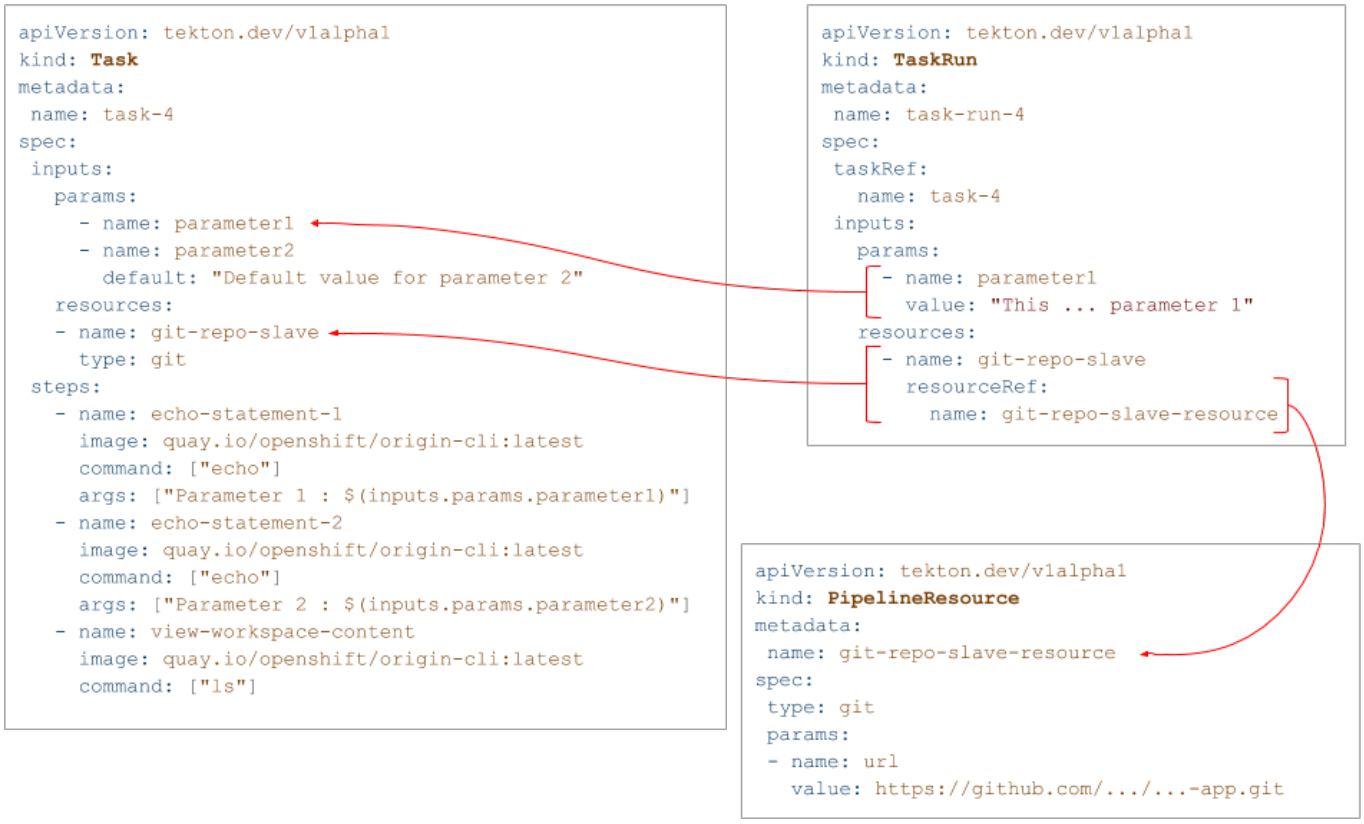
The following pipeline resource types exist :

- Git Resource - The git resource identifies a git repository, that contains the source code to be built by the pipeline. The resource can point to a specific branch or commit and can extract content from a specific directory.
- Pull Request - Can be used as an input resource to identify specific meta data about a pull request. if used as an output a pull request can be updated with changes made during the pipeline process.
- Image - An image to be created as part of the pipeline process.
- Cluster Resource - A different cluster to the cluster on which the pipeline is running. This can be used to deploy content to an alternative cluster as part of a deployment pipeline process.
- Storage Resource - Blob storage that contains either an object or directory.
- Cloud Event Resource - A cloud event that is sent to a target URI upon completion of a TaskRun.

Further details on the options for all of the above resources is included here : https://github.com/tektoncd/pipeline/blob/master/docs/resources.md

Task input example

The task defined in task-4.yaml uses both parameters and pipeline resources to get information into the task. This allows a generic task to be written with specific values supplied to it from the taskrun. The Taskrun object acts as a *value provider* giving specific values for parameters and referencing specific pipeline resources. The following diagram shows the relationship between the three specific objects.



As shown above the task has place-holders for two parameters. The first parameter has a value defined within the taskrun. The second parameter has a default value so it is not essential to provide a value for it in the taskrun. Both parameters are referenced from the steps of the task using the notation $(inputs.params.<parameter-name>).

The task also defines a resource object called git-repo-slave of type git. Within the taskrun an input resource object is defined with the same name (git-repo-slave) referring to a pipeline resource object called git-repo-slave-resource. A pipeline resource object is created from the yaml file git-resources.yaml which makes a reference to the actual git repository.

To create the resource object go back to the terminal app and execute the following command :

```
oc create -f git-resources.yaml
```

To view the resources in the project use the command:

```
tkn resources list
```

The response will be :

```
NAME                      TYPE   DETAILS
git-repo-slave-resource   git    url: https://github.com/marrober/slave-node-app.git
```

The use of pipeline resource objects for git repositories and created images (as output resources) helps teams to create generic build, test and deploy pipelines that can be reused across multiple projects where the projects simply define the custom pipeline resource objects that are specific to their project or environment.

Workspaces and Volumes

Workspaces allow you to organise the content used by tasks and the assets that are produced by tasks. This can be useful to add structure to the content during large complex pipelines.

Workspaces are storage structures within the pod that runs the containers of the pipeline and workspaces are scoped at the task level. Separate steps within a task can see the same workspace.

Volumes are similar to workspaces except for the fact that they are backed by persistent volumes. This ensures that content written to the volume is accessible by steps from multiple tasks, allowing for a greater separation of steps into different tasks. For example a generic build task could be used to create an executable, writing the deliverable to a volume. A separate testing task could then be invoked by a pipeline to perform tests against the newly created deliverable. Accessing the file via a volume will work for the two separate tasks.

Task 5 has steps for creating files in the workspace and in a volume, followed by steps to display the files in the workspace and the volume which work fine. Task 6 only has tasks for attempting to display the content of the workspace and the volume. Since the workspace in task 6 is a different workspace to that used in task 5 there is no content to display. The volume however shows the file written in the step in task 5. Tasks 5 and 6 are orchestrated by the pipeline called pipeline-5.

Create the persistent volume claim to use in this exercise with the command:

```
oc create -f persistentvolumeclaim.yaml
```

Create tasks 5 and 6:

```
oc create -f task-5.yaml

oc create -f task-6.yaml
```

Create the pipeline task:

```
oc create -f pipeline-5.yaml
```

The persistent volume will show that it is in an pending state after creation as no resource has attempted to consume it. After the task has been executed look again at the persistent volume and it will show that it is bound.

To see the state of the pvc enter the following:

```
oc get pvc
```

Before executing the task the state of the pvc should be as follows

```
NAME                    STATUS        VOLUME                                     CAPACITY
ACCESS MODES   STORAGECLASS   AGE
tekton-task-cache-pvc   Pending
gp2            4s
```

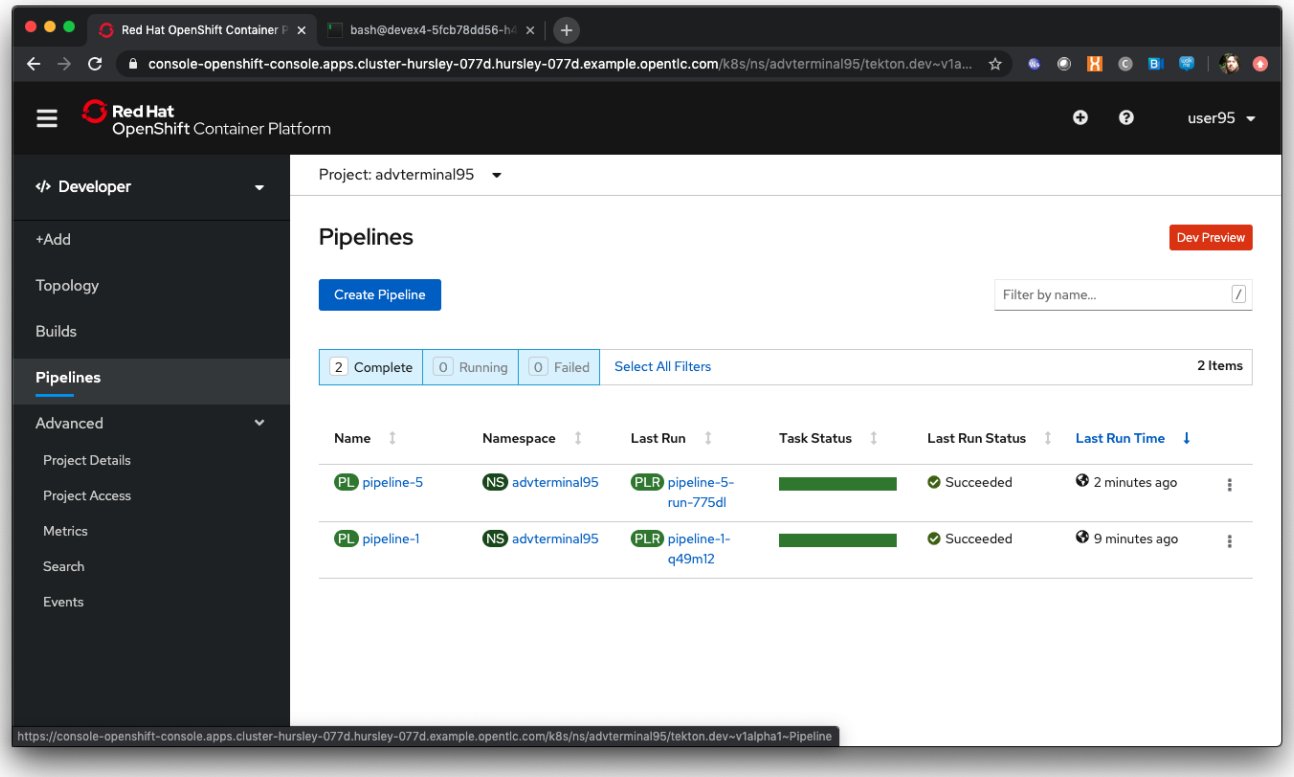
Once the pipeline has completed (you will run it after this) the pvc will indicate itself as bound - try it after the pipeline has completed

```
NAME                    STATUS        VOLUME                                     CAPACITY
ACCESS MODES   STORAGECLASS   AGE
tekton-task-cache-pvc   Bound         pvc-1d894a93-2646-11ea-9f45-0a9970779e5c   1Gi
RWO            gp2            2m2s
```

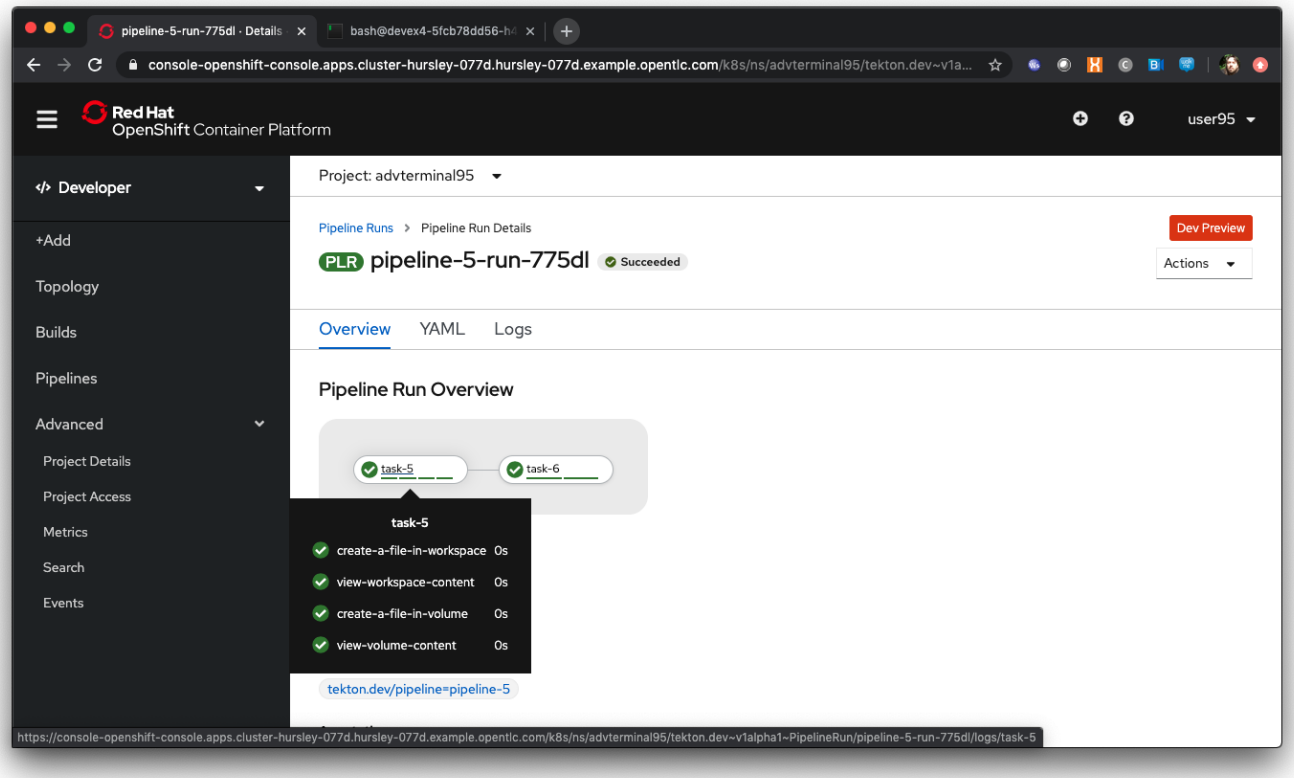
Execute the pipeline using the following command in the terminal

```
tkn pipeline start pipeline-5
```

Now switch to the OpenShift console. Select the Pipelines entry on the left side of the Developer panel.



You can click on the pipeline-run (labelled pipeline-5-run-XXXXX) and examine the logs for each of the tasks.



When you are satisfied that the pipeline has executed as expected, perform the check on the pvc in the terminal window

Camel K and Openshift Serverless Eventing

Author: Phil Prosser (feedback to pprosser@redhat.com)

Introduction

OpenShift Serverless

Openshift Serverless, based on Knative is the serverless technology that was introduced in Openshift 4.2. Openshift Serverless enables Pods running on Openshift to be scaled to 0 therefore taking zero processing power. Only when called, Openshift Serverless will scale the Pod up on demand before processing the request. Openshift Serverless also has the ability to autoscale based on load before eventually scaling back to zero when no requests are being received.

Openshift Serverless supports "Serving" and "Eventing"

At the time of writing, Serving is in technology preview, and Eventing is in Developer Preview

"Serving" enables request/response workloads, and Eventing enables asynchronous event based workloads using cloudevents.

Eventing has become an important part of a Microservice architecture. It enables services to notify other services of change (typically, change in state) in a loosely coupled manner. To enable this, Knative Eventing uses a publish and subscribe architecture. This enables the source service to publish events without having the knowledge of who maybe wanting to consume the events. It allows any number of sink services to subscribe to the event and act upon it.

In this lab, we are going to look at eventing, and how easy it is to integration with Camel K.

Check the project is ready to use

In the browser based terminal window, check you are still logged on and using the correct project by typing:

```
oc whoami
oc project
```

If the response from the commands indicates that you are not userX (where X is your assigned user number) and not using the project sandboxX please repeat the commands in the pre-requisites.

Whilst Openshift Serverless has it's own cli (kn), the purpose of this lab is to show the integration of

Camel K into Openshift Serverless and how easy this is to use.

Openshift Serverless and the Operator Lifecycle Manager

Openshift Serverless and the Operator Lifecycle Manager

Openshift Serverless uses the Operator Lifecycle manager, this means that its operator and Custom Resource Definitions (CRDs) will be added to Openshift via "OLM". Once created, the new CRDs will extend the Openshift data model allowing Openshift Serverless to be managed using the standard ‘oc’ command. Installing operators requires a higher cluster privilege so the presenter will have already set these up for you.

Creating the pre-requisites for the chapter

Before we can create an integration, we need to check that the camel-k integration added in a previous chapter is still active

In the terminal window, type

```
cd /workspace/workshop4/camelfiles/camelkplatform
oc apply -f integrationplatform.yaml
```

This will either create the integration platform or, if it is still active, indicate it is unchanged

now type

```
oc get integrationplatform
```

The output should look similar to below

```
NAME      PHASE
camel-k   Ready
```

Once the "Phase" says "Ready", you can continue

Create Knative Messaging Channel

Openshift Serverless uses Knative eventing, Knative eventing is a loosely couple asynchronous architecture allowing event producers to send an event to one or more event consumers. Event

Consumers can be scaled to 0 when no events are folowing through the system.

By default, Knative uses an in memory messaging channel. In this lab we will configure 2 of these channels to use with Camel K

Options are also available to replace in memory messaging with other event sources such as the Kafka Channel. By combining Camel K and Red Hat AMQ Streams (Red Hats Kafka Implementation) on Openshift you create a powerful / reliable cloud native eventing platform for your applications.

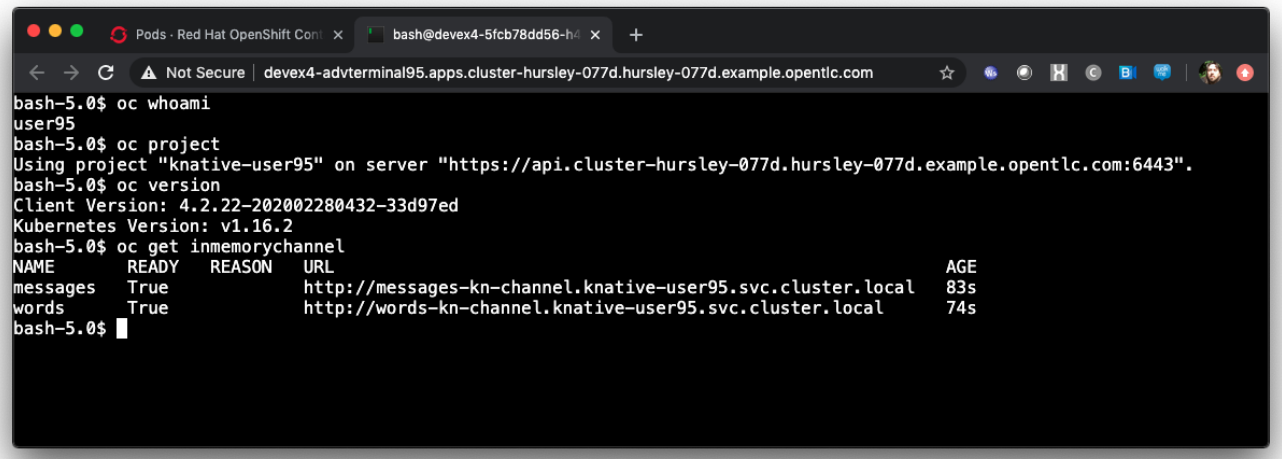
In the browser terminal window type the following:

```
cd /workspace/examples/knative
oc apply -f messages-channel.yaml
oc apply -f words-channel.yaml
```

To make sure the channels have been created correctly type:

```
oc get inmemorychannel
```

You should see a screenshot like the one below



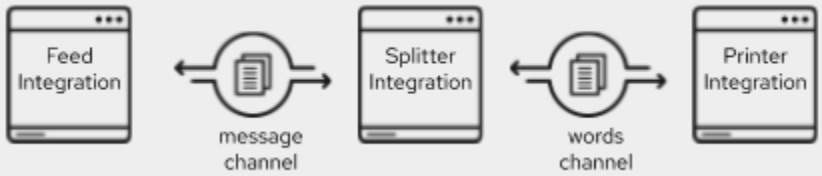
You are looking for *READY* to be *True*

Deploy the Integrations

Introduction to the integrations that we will use

Now that we have deployed 2 message channels, we will deploy 3 Camel K Integrations. *feed.groovy* will generate a simple sentence every 3 seconds, and send this to the *message channel*, *splitter.groovy* will subscribe to the *message channel*, take the message, split the message into individual words before sending the individual words to *words channel*. Finally, *printer.groovy* will subscribe to the *words.channel*, read the words from the channel and print them to the output log.

The flow looks like:



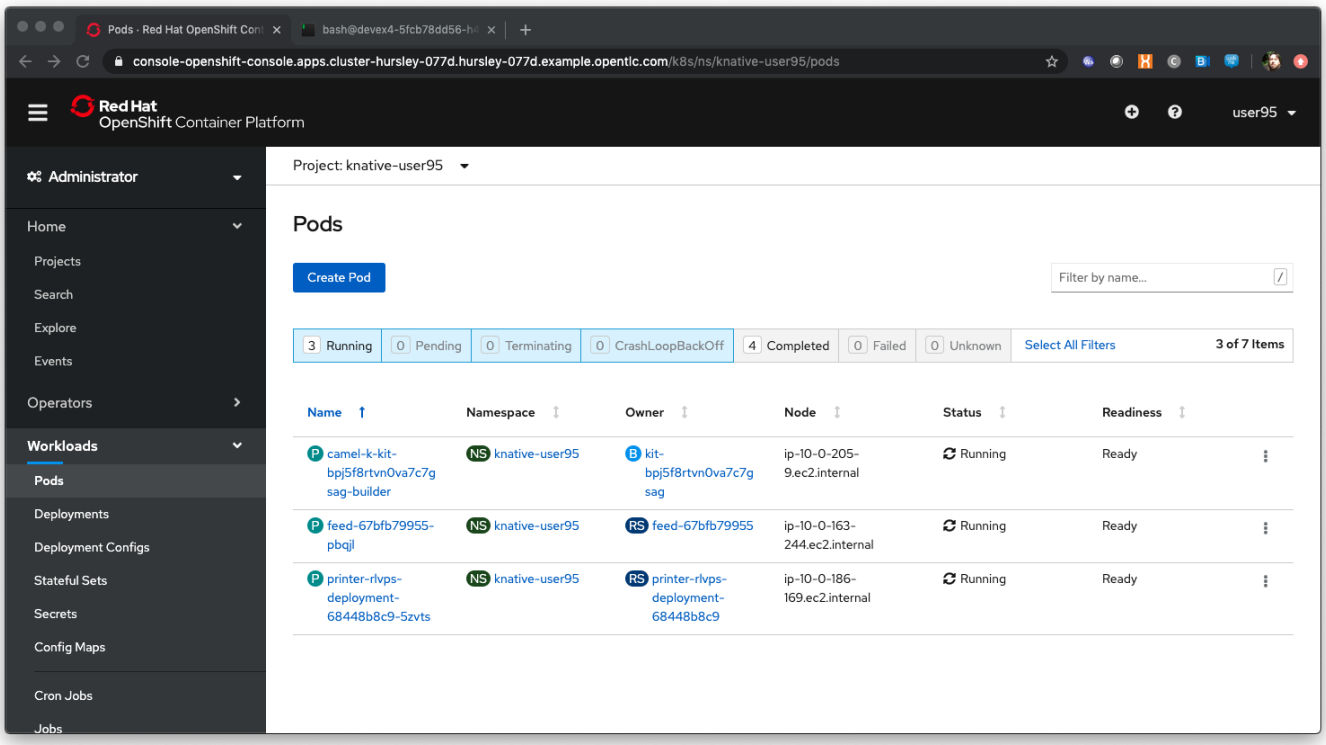
In the terminal window, deploy the 3 integrations

```
kamel run feed.groovy
kamel run splitter.groovy
kamel run printer.groovy
```

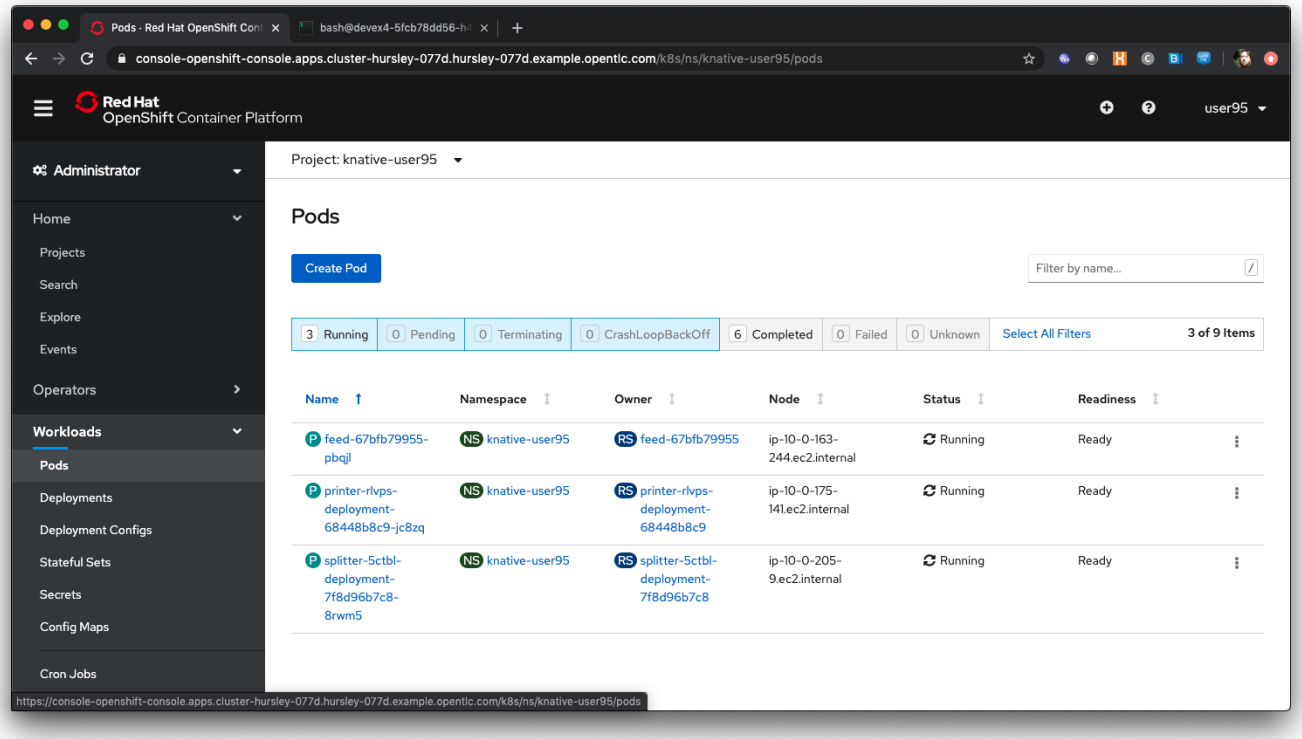
First go to the Administrator view in the OpenShift console - at the top left make sure the view is set to Administrator

Go to Workloads/Pods. As you are using a single namespace for the workshop this should display the active Pods in sandboxX, where X is your user number

Watch as the integrations are created using builder and deployment containers. This may take a little while.



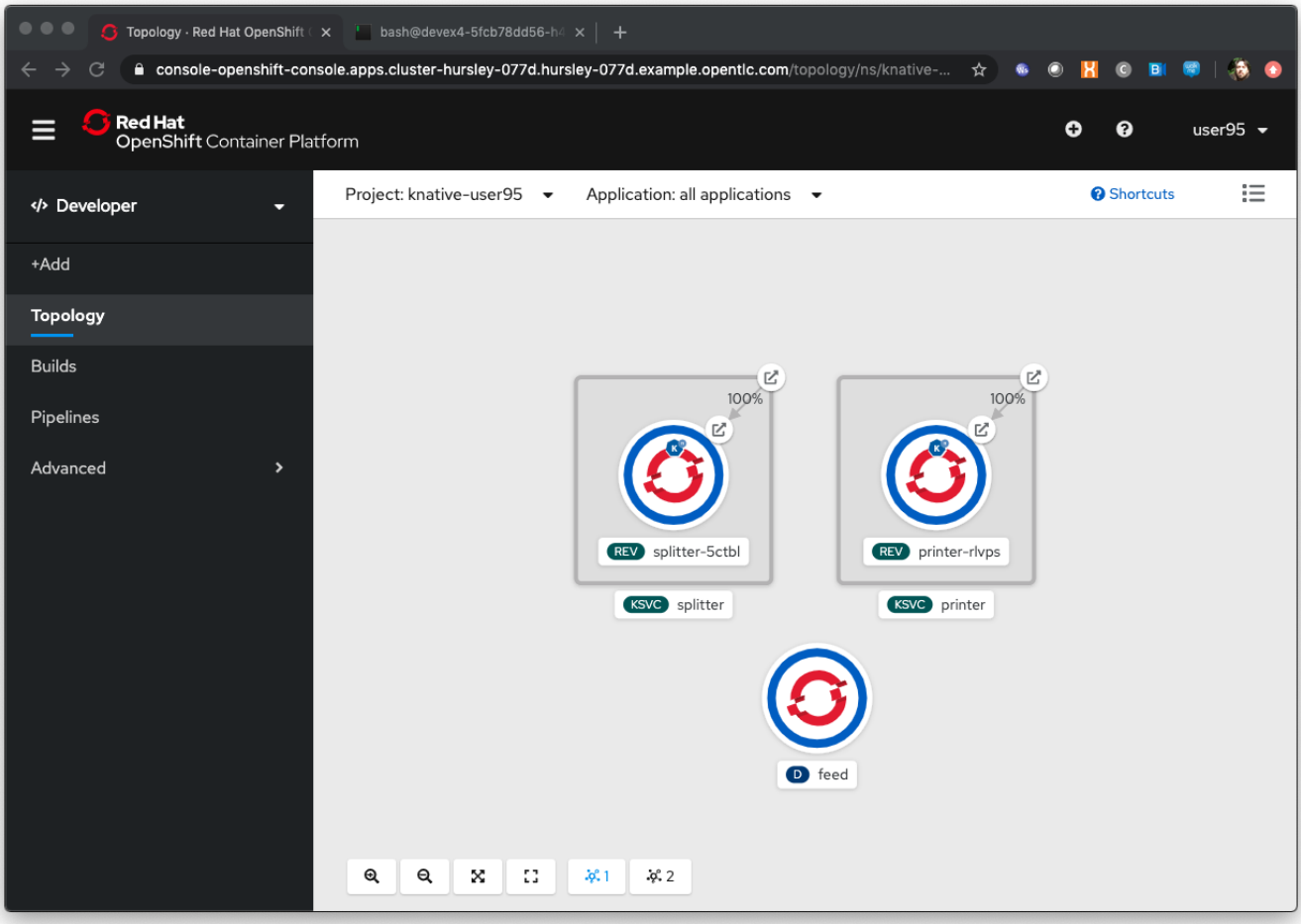
Once it has finished deploying the integrations you will have three Pods active as shown similar to the screenshot below (ignore the devex Pod, this is your terminal Pod)



Now go to the developer view in the OpenShift Console

Now that all 3 of the Integrations are deployed, the topology view should look like the screenshot

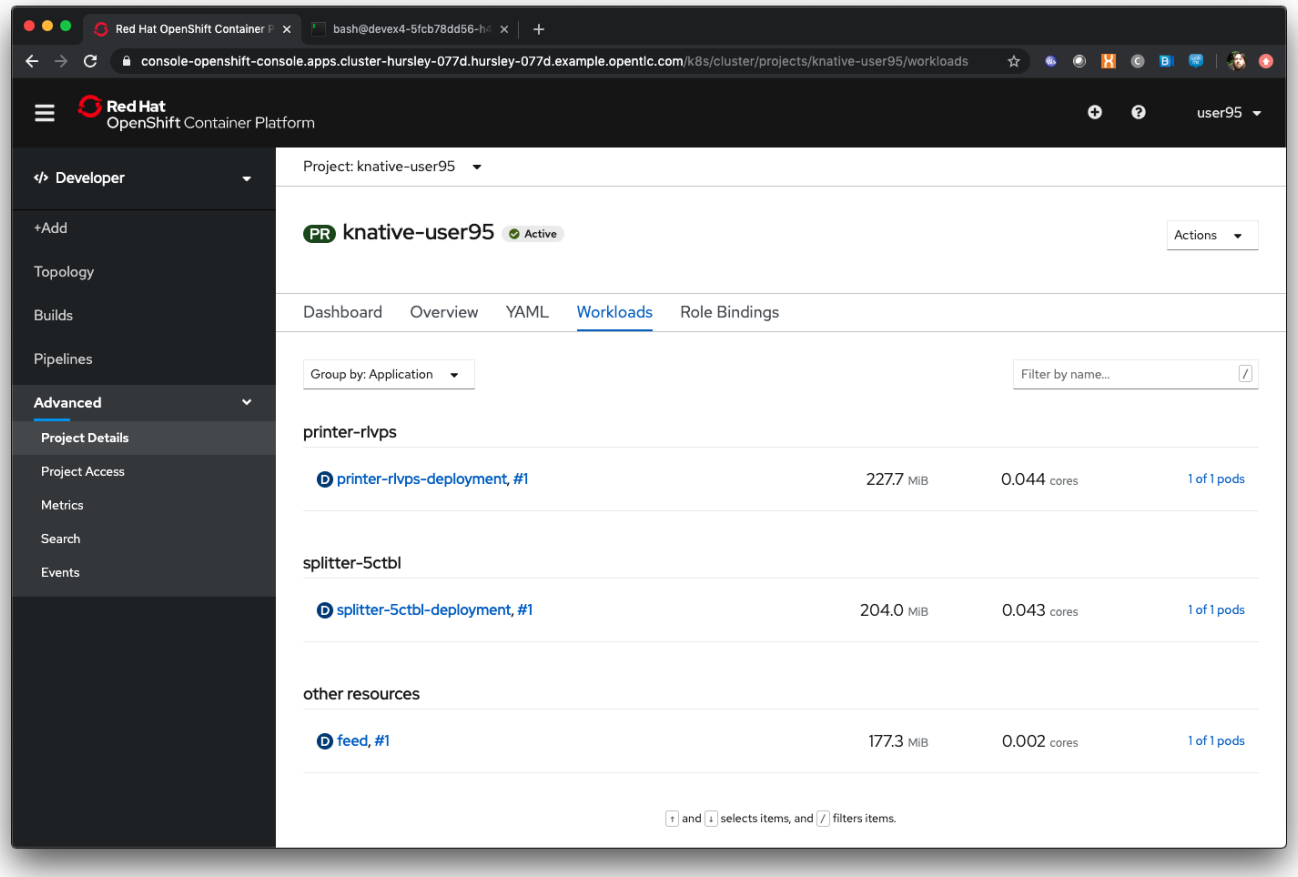
below



The Knative service is represented by the square box. You should see 2 of these in the topology view. On the Openshift red logo in the middle of the service you will see the Knative "K" logo. You definiately know that you have a Knative service now. You will also notice an artefact called "KSVC", this is the Knative Service defined to Openshift. There is also an artefact called "REV", this is the Knative revision that is current running. Revisions can be used to implement a Canary Release strategy. The diagram shows that 100% of the traffic is routed to the revision shown on the topology view. If you click on one of the "KSVC" on the topology view you will see an option to set the traffic distribution

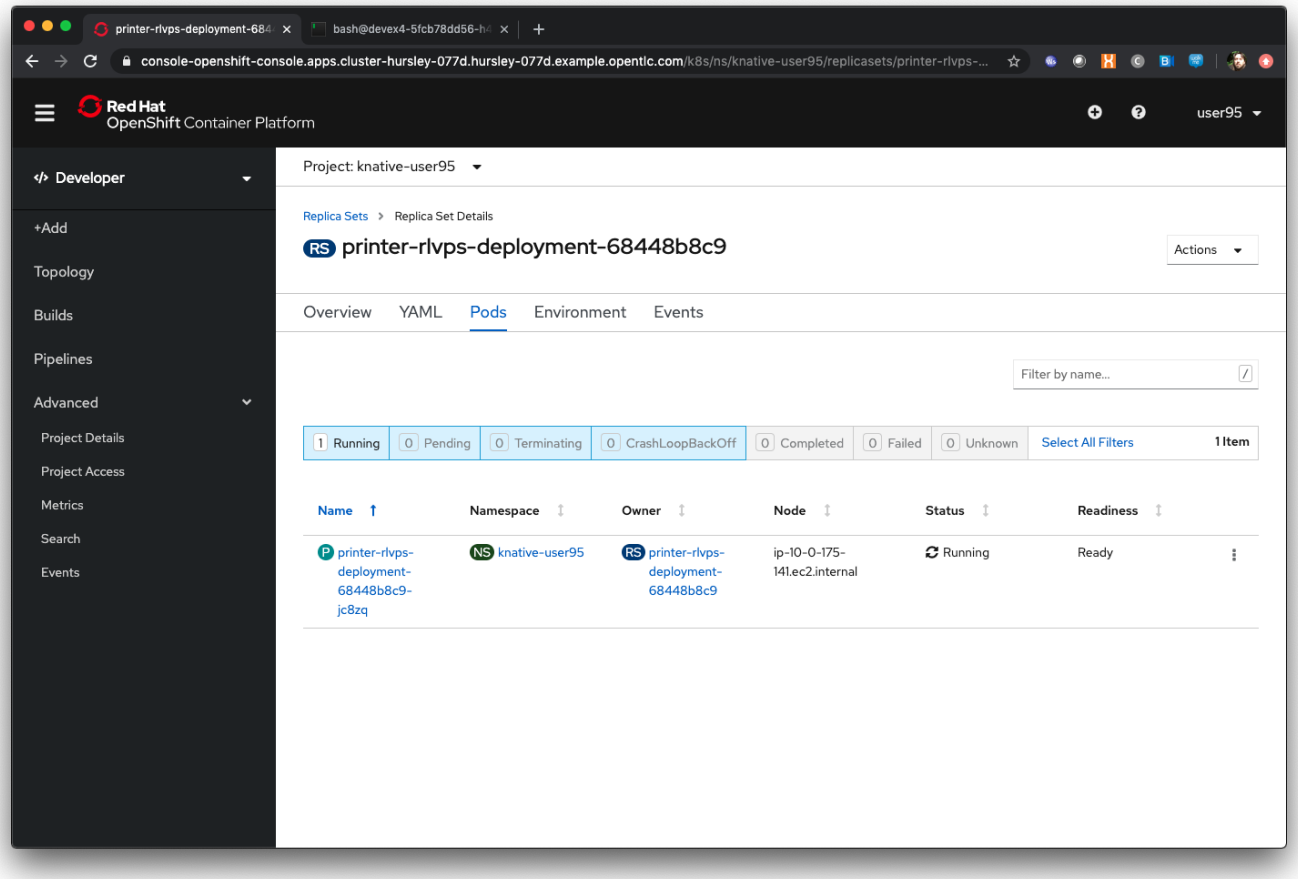
Each of the integrations is producing log information.

At the time of writing, there is no easy way to view the pod log files of a knative service in the console, so in the developer view click on Advanced/Project Details and choose Workloads



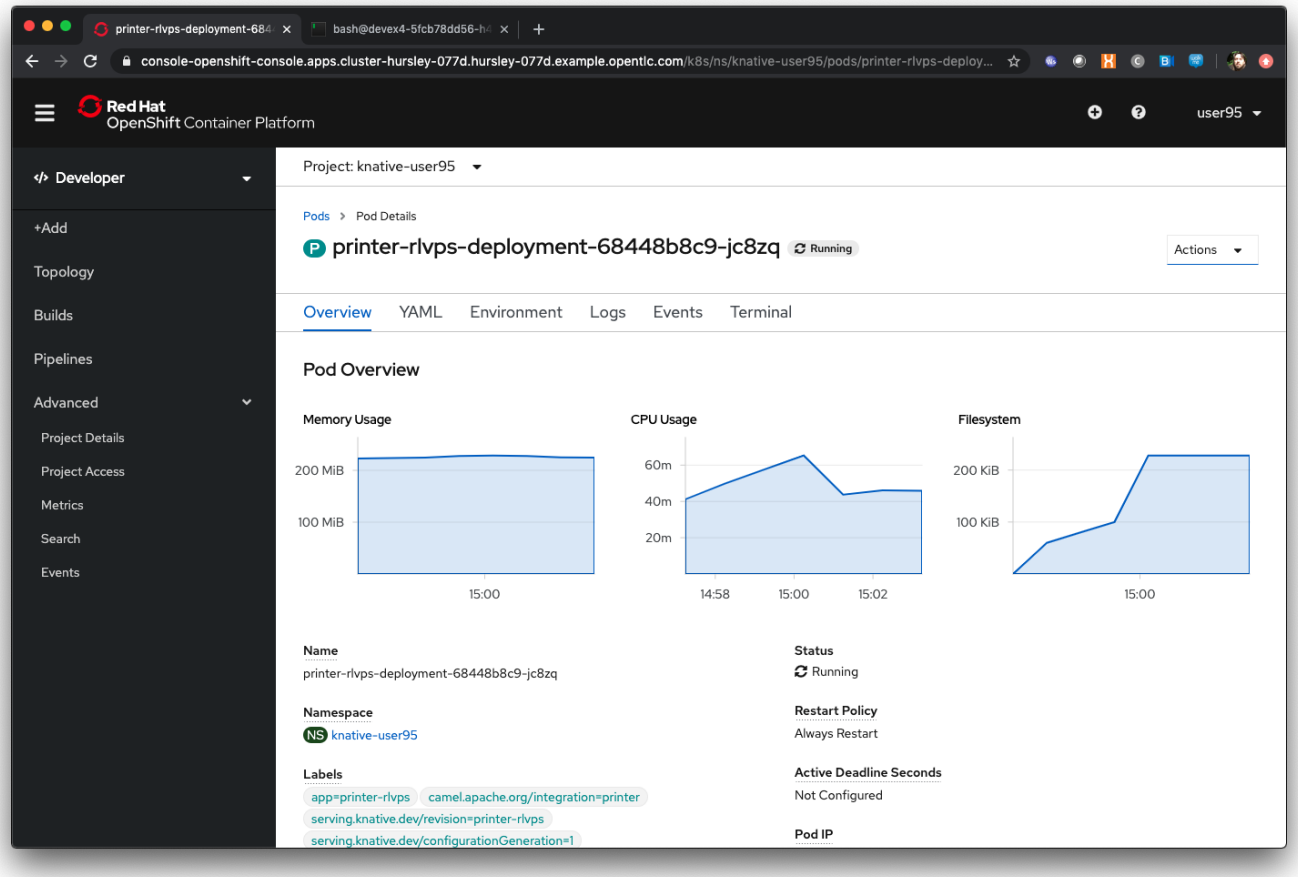
For each workload, you should see a *1 of 1 pods* on the right hand side. *Click* on the *1 of 1 pods*.

You should see a screen similar to the one below

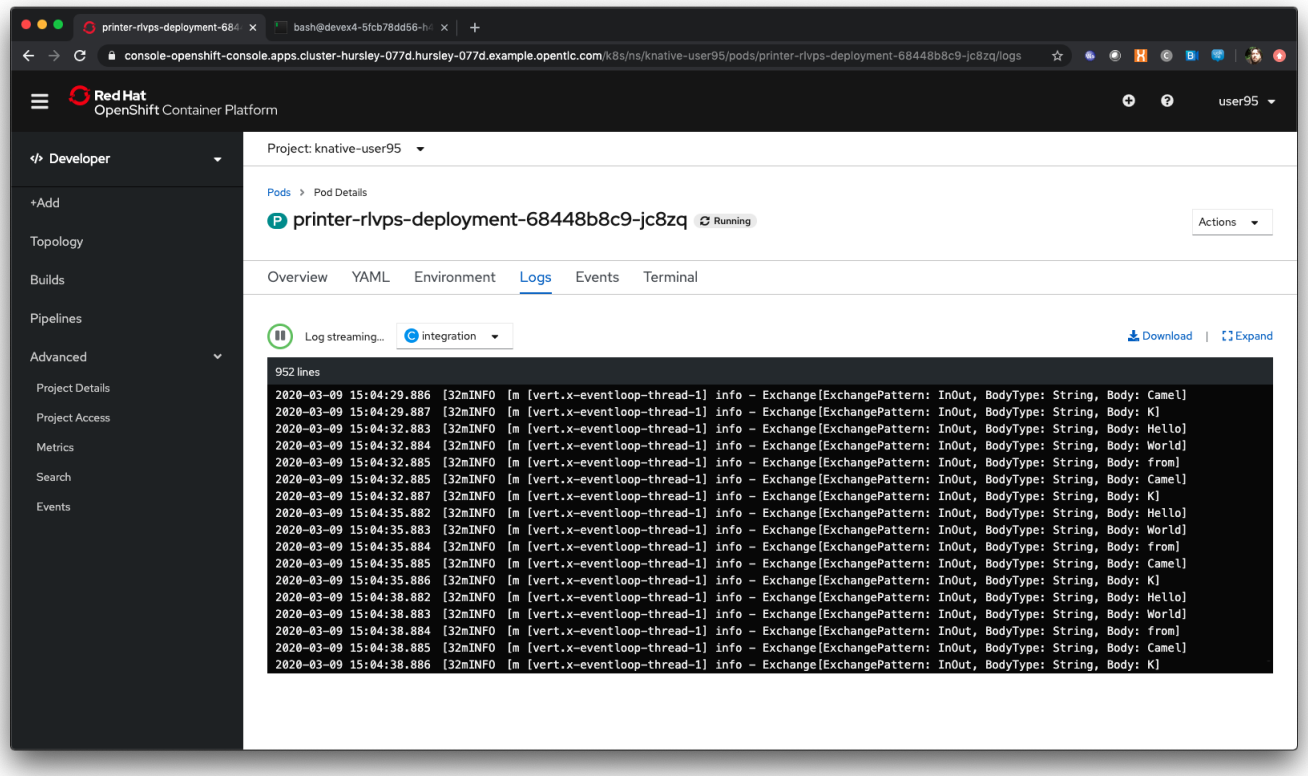


Click on the Pod name on the left e.g. printer-xxxxxxxxxxxx

This should show you a screen similar to the one below



Click on *Logs* to view the log for the pod. It should look something like the one below



Repeat the steps above for the other Integrations if you like.

Edit the Integration to use a Counter and Cache

NOTE

Because the output from the Feed Integation doesn't change, it's hard to see if all the messages are being processed, or indeed if some are being dropped. Lets make a small change to the Integration. The change will add a cache, and a counter to ensure that each message has a counter in it.

In the terminal window edit the Integration called feed.groovy

```
cd /workspace/examples/knative
vi feed.groovy
```

Between the line starting with **from** and the line starting with **.setBody** insert the follow code (copy the code by higlightling it and copying it)

```
        .setHeader("CamelCaffeineAction", constant("GET"))
        .setHeader("CamelCaffeineKey", constant("count"))
        .toF("caffeine-cache://%s", "messagecount")
        .choice()
                .when().simple('${body} == null') // When no counter stored, default to
zero
                        .setHeader('counter').constant(0)
                .otherwise() // retrieve the counter
                        .setHeader('counter').simple('${body}')
        .end()
        .setHeader('counter').ognl('request.headers.counter + 1')
        .setBody().simple('${header.counter}')
        .setHeader("CamelCaffeineAction", constant("PUT"))
        .setHeader("CamelCaffeineKey", constant("count"))
        .toF("caffeine-cache://%s", "messagecount")
        .setBody().simple('Hello${header.counter} World${header.counter}
from${header.counter} Camel${header.counter} K${header.counter}')
```

TIP

I'm no vi expert, but if you don't know vi, use the keyboard arrow keys to move to the line beginning with **from**, then to go to the end of the line press **$**, press **i**, press the **right arrow** once to move the cursor to the end of the line and press **enter** (this shoud insert a blank line and move the cursor to the beginning of that line. Paste in the code by pressing **ctrl v**. Don't worry about the indentation too much. Once pasted in press **esc**.

The final line pasted in is **.setBody**. There is an existing **.setBody** line that we need to delete, the line looks like :-

```
.setBody().constant("Hello World from Camel K")
```

TIP

To delete, move the cursor to the line and press **dd** Finally save your work by typing **:wq** and press **enter**

Once complete, the integration should look like :-

```
// camel-k: language=groovy
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

from('timer:clock?period=3s')
        .setHeader("CamelCaffeineAction", constant("GET"))
        .setHeader("CamelCaffeineKey", constant("count"))
        .toF("caffeine-cache://%s", "messagecount")
        .choice()
                .when().simple('${body} == null') // When no counter stored, default to
zero
                        .setHeader('counter').constant(0)
                .otherwise() // retrieve the counter
                        .setHeader('counter').simple('${body}')
        .end()
        .setHeader('counter').ognl('request.headers.counter + 1')
        .setBody().simple('${header.counter}')
        .setHeader("CamelCaffeineAction", constant("PUT"))
        .setHeader("CamelCaffeineKey", constant("count"))
        .toF("caffeine-cache://%s", "messagecount")
        .setBody().simple('Hello${header.counter} World${header.counter}
from${header.counter} Camel${header.counter} K${header.counter}')
        .to('knative:channel/messages')
        .log('sent message to messages channel')
```

Each word should now have the counter appended to it

Test your work by typing :-

```
kamel run feed.groovy
```

Use the skills you've learned to view the output of the container logs to check that the messages now contain a counter.

Knative in action

Make sure you are in the developer view of the console, looking at the Topology view before continuing

The 2 Integrations "hooked" into Knative Eventing are the *spilter* and *printer* integrations (you can visually see this on the topology view).

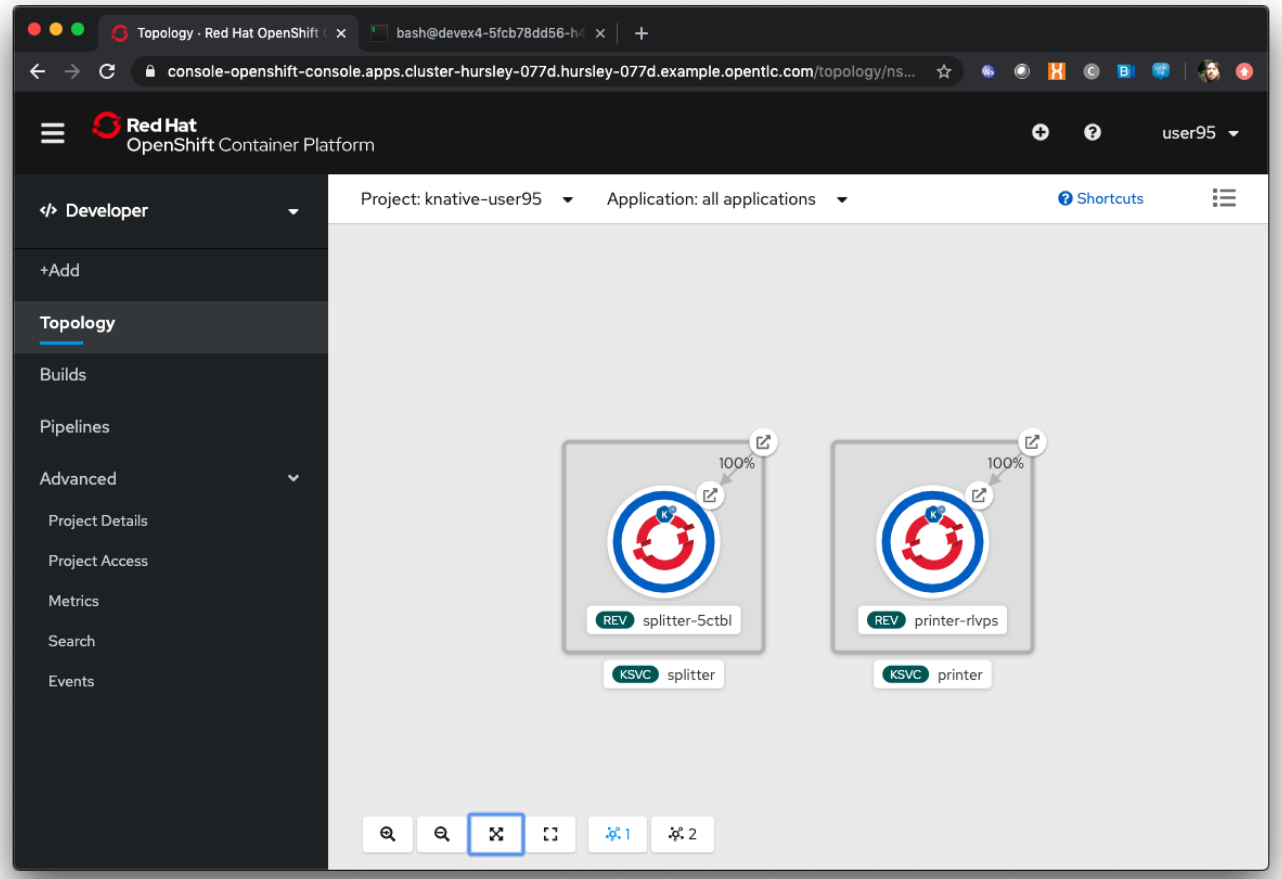
Let's see if the promise of scale to zero works.

To stop the integrations, we need to stop messages arriving at the "messages.channel". To do this, we need to stop the feed integration.

In the terminal browser window, type

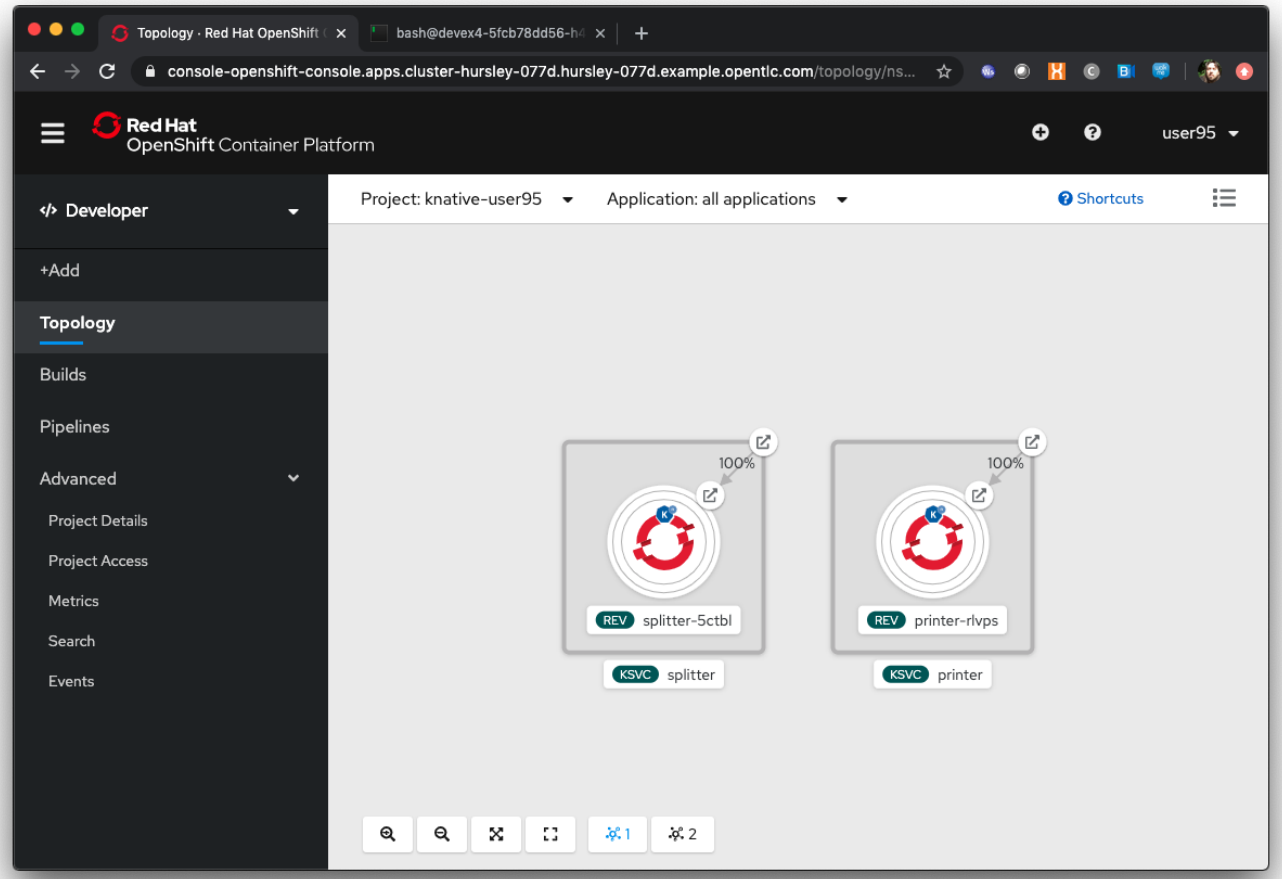
```
kamel delete feed
```

Go back to the topology view, you will notice that the feed integration has gone.



Show some patience now, keep lookng at the topology view, we are waiting (and hoping!) that the integrations scale down to zero.

You will know when this starts as the rings around the circles will change from the normal blue to a very dark blue, before going white. Once they are white, the integrations are scaled to zero just like the screenshot below



To wake the Integrations up again, redeploy the *feed* integration.

```
kamel run feed.groovy
```

Go back to the topology view and you should see the *feed* integration redeploy, and the *spillter* and *printer* integrations awake from their slumber.

This shows the potential for effective serverless behaviour by the down-scaling of unused applications, combined with the ease of Camel-K integrations.

To clean up before the next chapter run the following commands in the terminal:

```
kamel delete feed
kamel delete splitter
kamel delete printer
```

Replace Knative in memory messaging with AMQ Streams (Kafka)

Introduction

You may have noticed that the default Knative event channel is in memory. This means that there is the potential for message loss in the solution, and also the potential for some subcribers to miss messages.

Most applications need some form of persistent messaging, avoiding message loss in the event of something going wrong. A popular choice in the microservice world for publish and subsribe eventing is Apache Kafka. In addition to the InMemoryChannel used in the first part of this lab, Knative also has a channel type called KafkaChannel. As the name suggests, this allows Knative Eventing to use a Kafka Topic as the persistent store for the messages ensuring the reliable delivery to all subscribers.

So, we can use Kafka as a persistent store, but how do we get a Kafka Cluster installed on Openshift?

Red Hat has an operator based Enterprise Kafka distribution called AMQ Streams. AMQ Streams is based on the opensource project Strimzi (https://strimzi.io). At the time of writing, Strimzi is a sandbox project within CNCF.

Using Kafka behind Knative eventing means that developers also get access to the full Kafka eco-system. Change Data capture could be used to publish messages that are consumed by Knative eventing clients. Kafka clients such as the streaming api could be used to read messages published by Knative based services. All Kafka API's and services could be used as part of the solution.

Before you can continue, you need to delete the in memory channels created earlier in the lab.

```
oc delete inmemorychannel messages
oc delete inmemorychannel words
```

Double check that the inmemorychannels have been successfully deleted

```
oc get inmemorychannels
```

The output should look like :-

```
No resources found in sandboxX namespace.
```

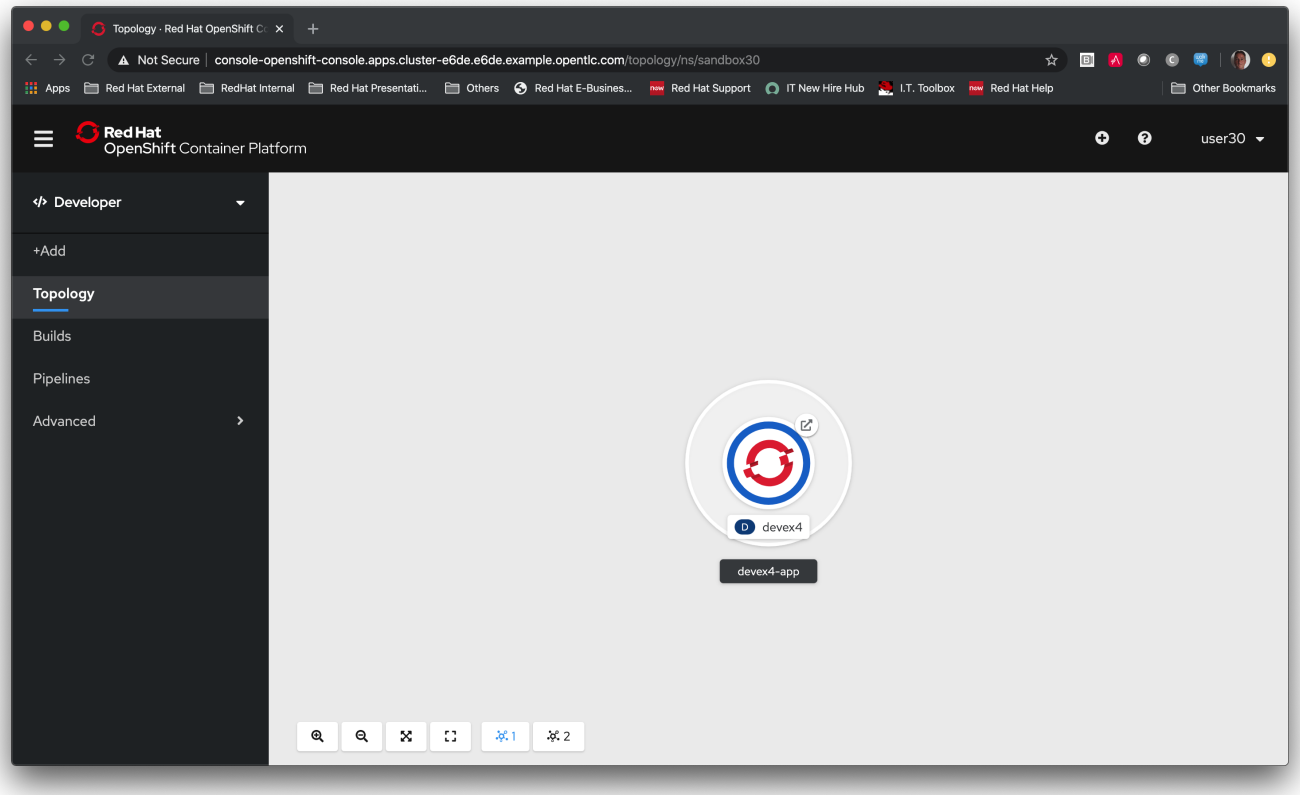
Install a Kafka Broker using AMQ Streams

AMQ Streams using an Operator to perform all of the administration tasks on Openshift. The operator is already installed for you and makes available a number of new Custom Resource Definitions (CRD). These are Kafka, KafkaBridge, KafkaConnect, KafkaConnectS2I, KafkaConnector, KafkaMirrorMaker, KafkaMirrorMaker2, KafkaTopic, and KafkaUser. As you an see, the AMQ Streams operater has a rich set of functionality. All of the configuration work can be performed throught the operator without detailed know of Kafka Installations. You will see in the next step how easy it is to create an Kafka cluster including the Kafka Brokers, and Zookeeper clusters with one simple YAML file.

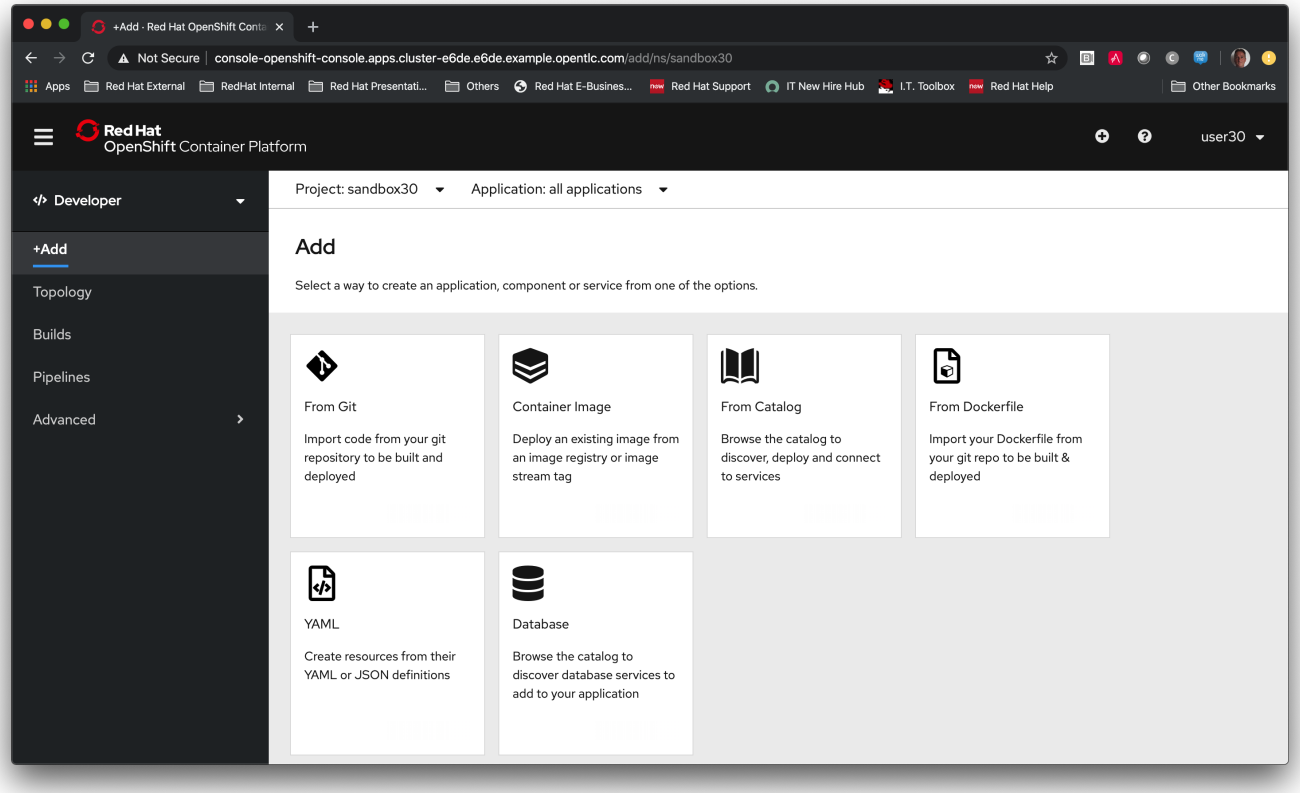
Once the operators are installed, the CRD's become available for you to use through the developer view in the Openshift console.

In the Openshift console, make sure you in developer view looking at your sandboxxX project.

This should look similar to the screenshot below



Click on "+Add"



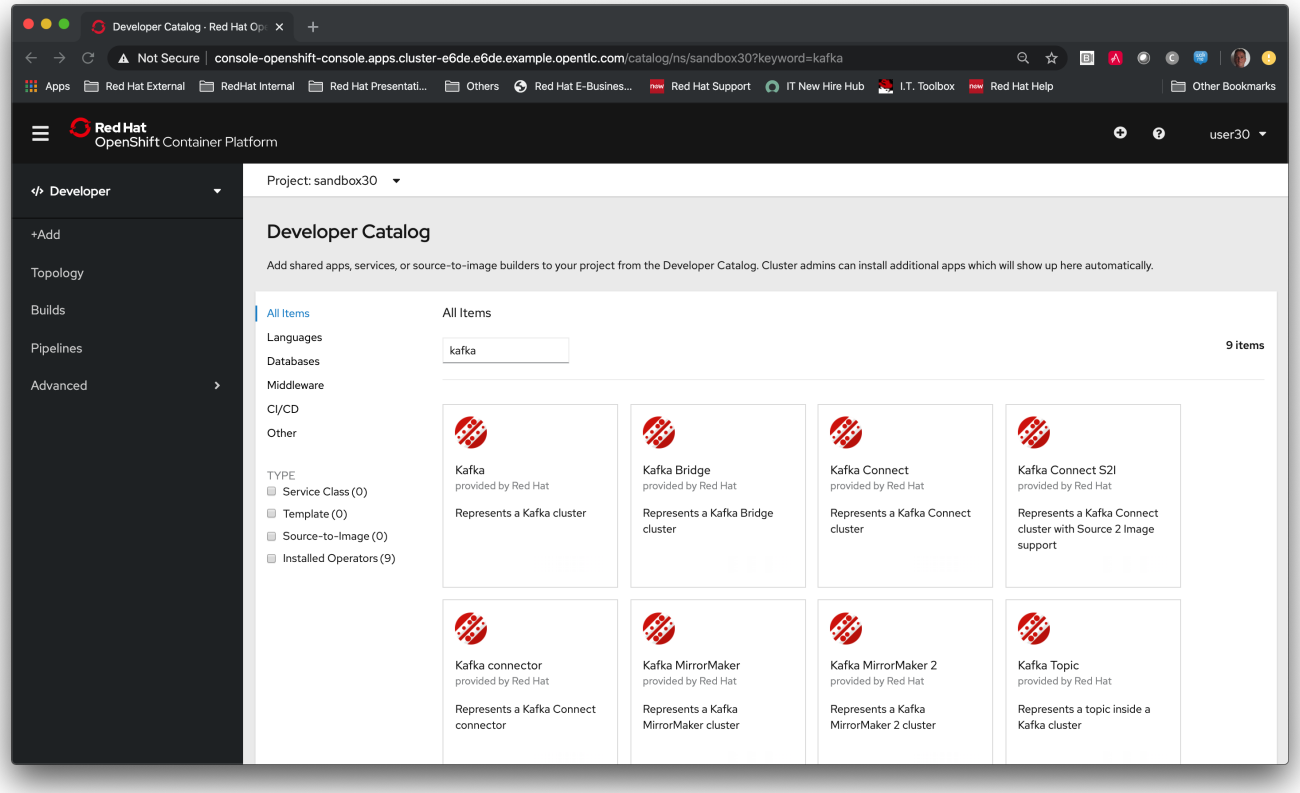
You will see "From Catalog" on the screenshot above, *click* on it

This will present you with the developer catalog.

Underneath the words "All Items" you will see a text box that says "Filter by keyword"

In that text box, type "kafka"

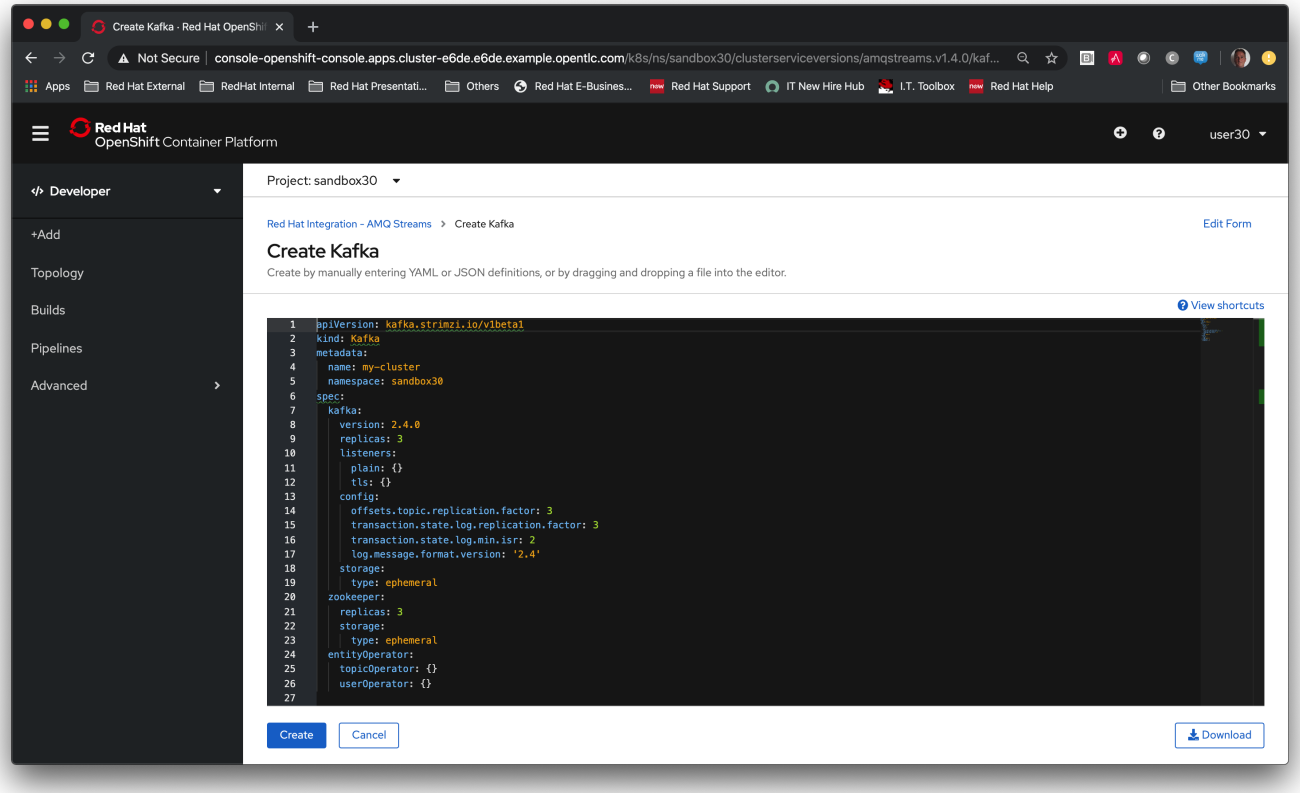
You should see a screenshot similar to the one below



Click on Kafka

Press the Create button

You will now be presented with a sample Kafka yaml file that looks similar to the screenshot below



Due to the number of workshop attendees and the size of the Openshift cluster we need to make the Kafka deployment smaller by reducing the number of Kafka Brokers in the cluster to 1, and the number of zookeeper instances to 1.

To do this, change

spec.kafka.replicas = 1

and

```
spec.zookeeper.replicas = 1
```

Because the kafka brokers are now set to one replica, you have to modify the configuration elements below

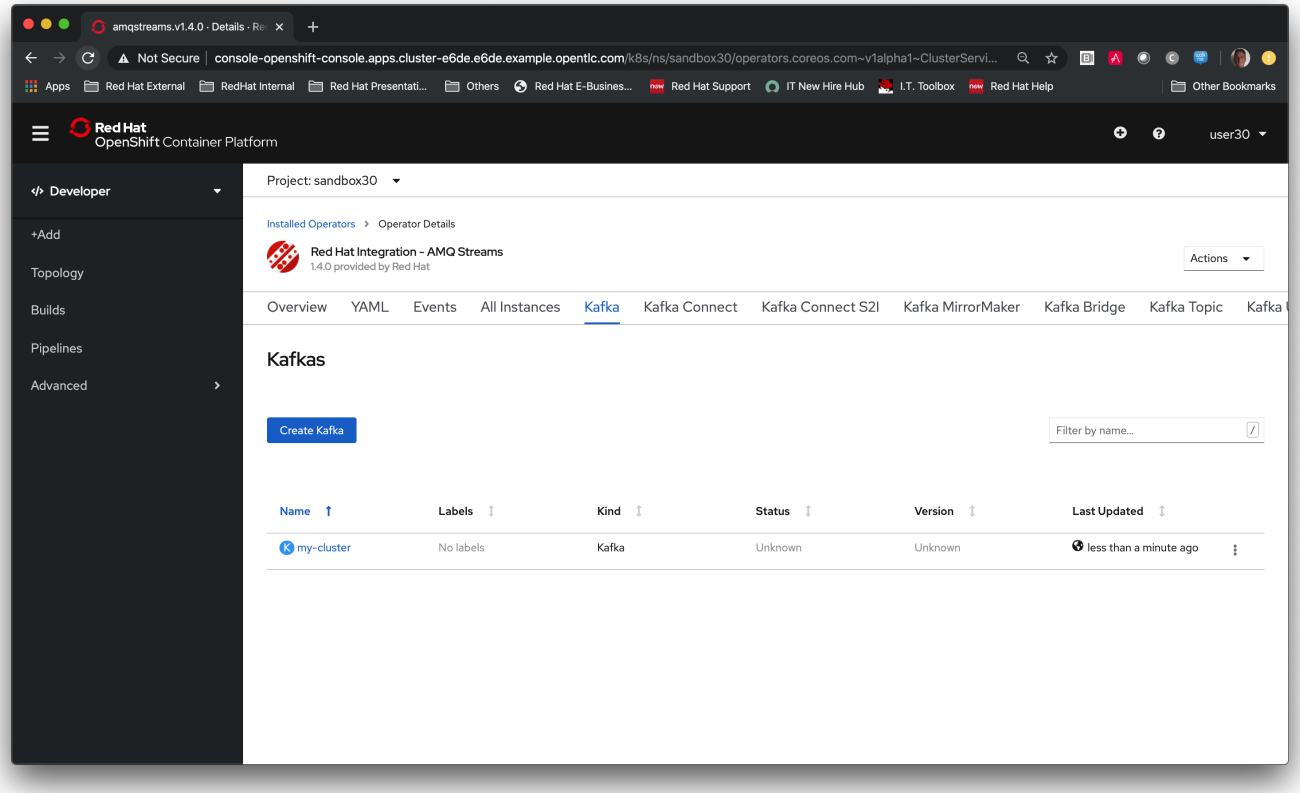
```
offset.topic.replication.factor: 1

transaction.state.log.replication.factor: 1

transaction.state.log.min.isr: 1
```

Your final Kafka yaml file should look like :-

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  namespace: sandbox30
spec:
  kafka:
    version: 2.4.0
    replicas: 1
    listeners:
      plain: {}
      tls: {}
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min.isr: 1
      log.message.format.version: '2.4'
    storage:
      type: ephemeral
  zookeeper:
    replicas: 1
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

Press create.

You will be presented with a screen that looks similar to the screenshot below. (This is looking at the AMQ Streams operator)

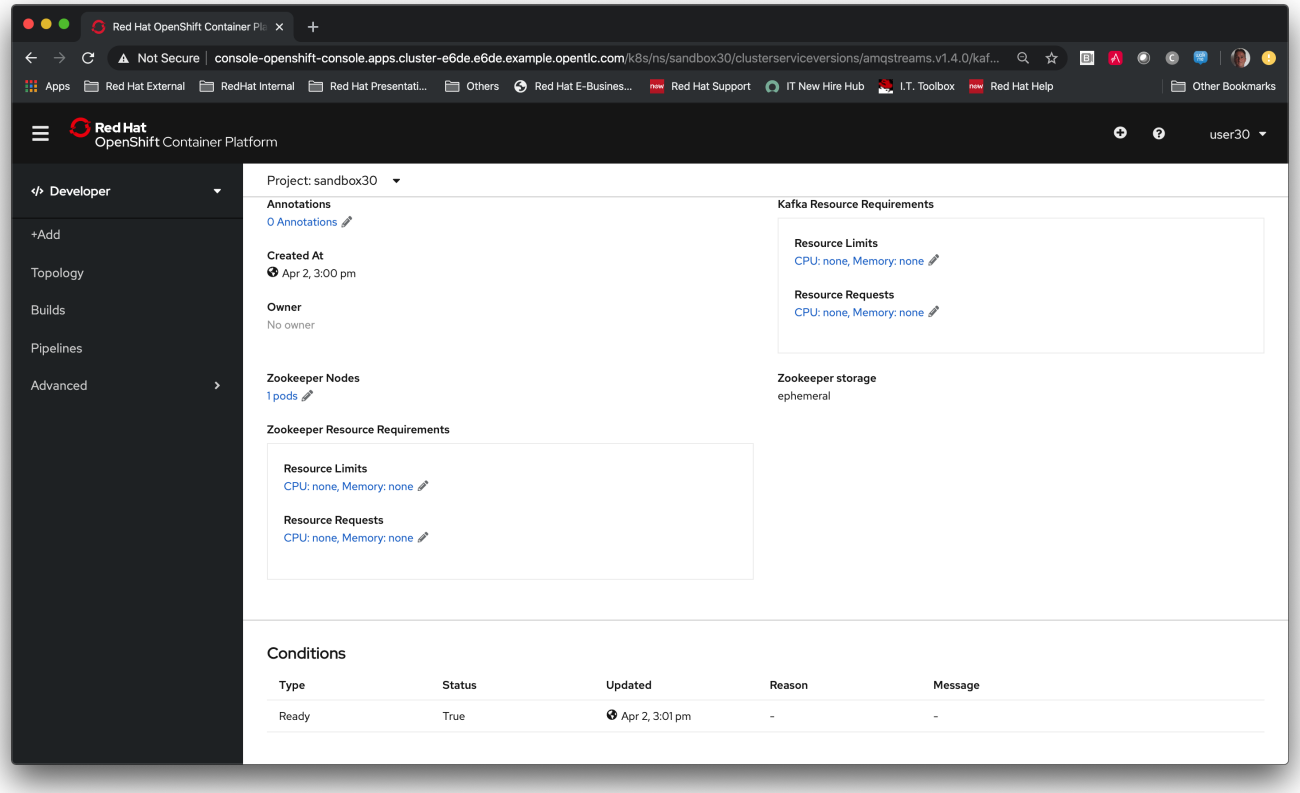


If you click on "my-cluster", you can see the current state of the Kafka deployment by scrolling to the bottom of the screen

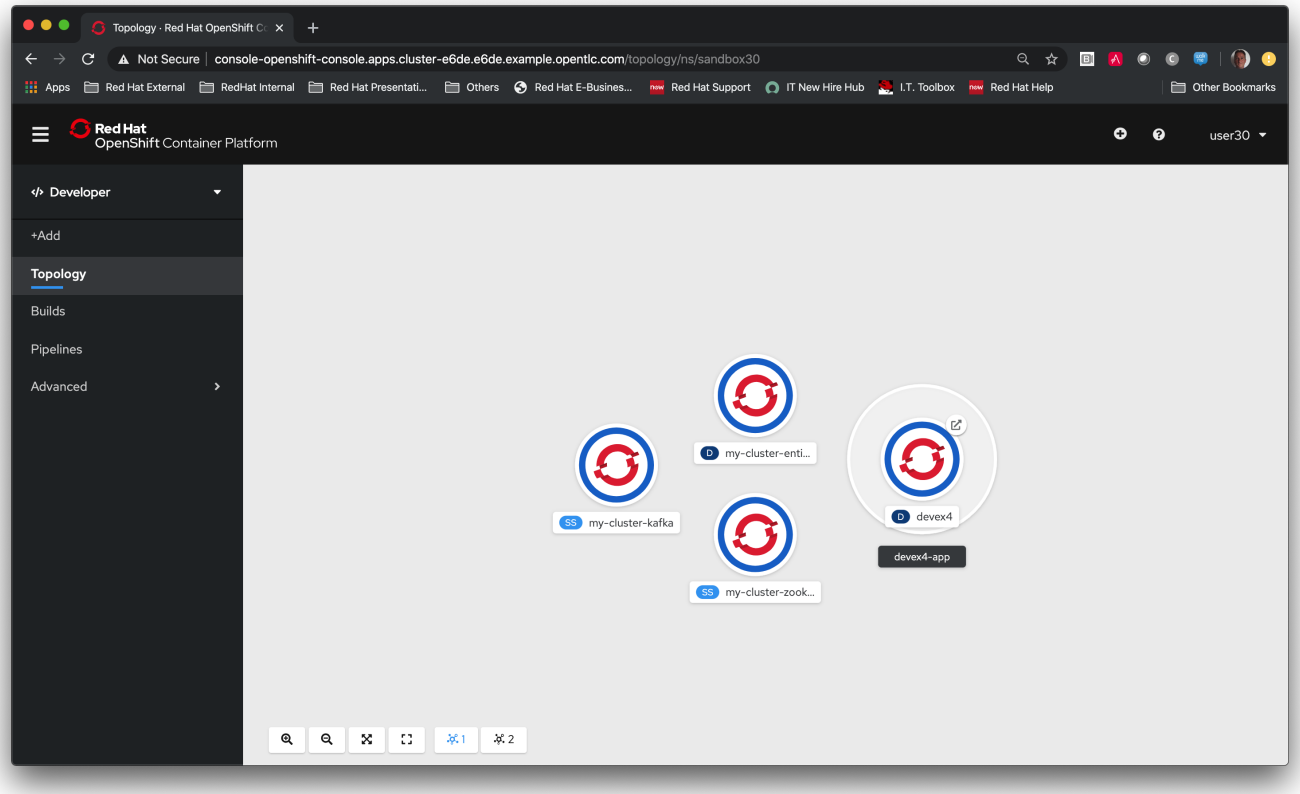
Underneath Conditions, you are looking for

Type Status Ready True

It should like the screenshot below



You can also go back to the Developer view in the console. You will see the Zookeeper and Kafka Brokers starting up. Once everything is started it should look similar to the screenshot below.



Install Knative Kafka Eventing

Note: Knative Kafka is operator based, the operator has been install previously for you so the installation should be straight forward

The CRD for Knative Kafka is called "KnativeEventingKafka", you need to add a CRD to your namespace to integrate Knative eventing with Kafka.

Fistly,

You need to find the bootstrap server of your Kafka Cluster.

To do this, go to the terminal window and type

```
oc get services
```

You will see a list of services similat to the list below

```
NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
devex4                        ClusterIP   172.30.227.91    <none>        8080/TCP
3h55m
my-cluster-kafka-bootstrap    ClusterIP   172.30.67.254    <none>
9091/TCP,9092/TCP,9093/TCP   65m
my-cluster-kafka-brokers      ClusterIP   None             <none>
9091/TCP,9092/TCP,9093/TCP   65m
my-cluster-zookeeper-client   ClusterIP   172.30.178.174   <none>        2181/TCP
66m
my-cluster-zookeeper-nodes    ClusterIP   None             <none>
2181/TCP,2888/TCP,3888/TCP   66m
```

You are after the service name ending in "bootstrap" - make a note of it as you will need it in the next step

```
cd /workspace/workshop4/camelfiles/streams
```

Create a KnativeEventing Kafka CRD

```
vi mykafkaknative.yaml
```

Paste the following into the file, ensuring you replace the bootstrapServer with the one you noted above. Please make sure you keep the single quotes in the file.

Replace the *X* o *sandbox* with your allocated number

```
apiVersion: eventing.knative.dev/v1alpha1
kind: KnativeEventingKafka
metadata:
  name: knative-eventing-kafka
  namespace: sandboxX
spec:
  bootstrapServers: 'replacewithyourbootstrapserver:9092'
  setAsDefaultChannelProvisioner: true
```

Once you have changed the bootstrapserver, save the file

You can now store the CRD into Openshift by typing the following :-

```
oc apply -f mykafkaknative.yaml
```

This will now perform perform the Integration between Knative Eventing and Kafka

To check the status of the Integration, type

```
oc describe KnativeEventingKafka knative-eventing-kafka
```

The output will look similar to below

```
Name:         knative-eventing-kafka
Namespace:    sandbox30
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"eventing.knative.dev/v1alpha1","kind":"KnativeEventingKafka","metadata":{"
annotations":{},"name":"knative-eventing-kafka","...
API Version:  eventing.knative.dev/v1alpha1
Kind:         KnativeEventingKafka
Metadata:
  Creation Timestamp:  2020-04-02T15:18:07Z
  Generation:          1
  Resource Version:    196044
  Self Link:
/apis/eventing.knative.dev/v1alpha1/namespaces/sandbox30/knativeeventingkafkas/knative-
eventing-kafka
  UID:                 f8278481-3991-4d8d-a145-4d11ddfa56fb
Spec:
  Bootstrap Servers:                   my-cluster-kafka-bootstrap:9092
  Set As Default Channel Provisioner:  true
Status:
  Conditions:
    Last Transition Time:  2020-04-02T15:18:28Z
    Status:                True
    Type:                  DeploymentsAvailable
    Last Transition Time:  2020-04-02T15:18:19Z
    Status:                True
    Type:                  InstallSucceeded
    Last Transition Time:  2020-04-02T15:18:28Z
    Status:                True
    Type:                  Ready
  Version:                 0.13.2
Events:                    <none>
```

Very near the bottom of the output, it should say Type = Ready. This means that you are good to go.

Run the examples

```
cd /workspace/workshop4/camelfiles/streams
```

Create the Knative Eventing channel backed by Kafka as a persistent store

```
oc apply -f messages.yaml
```

Lets make sure that everything appears to be wired up.

Check to make sure that the Kafka channel is deployed and ready

Type

```
oc get kafkachannel
```

If everything worked as expected, you should see the output similar to below

```
NAME                                        PARTITIONS   REPLICATION FACTOR
knative-messaging-kafka.sandbox30.messages   1            1
```

OK, so the above output suggests that the channel is ready and working.

Lets make sure that a topic has been created in Kafka for us.

If you recall, the AMQ streams operator has a number of different CRD's, therefore the creeation of the Knative Channel, should have created a KafkaTopic CRD. Lets check to make sure that's the case.

Type

```
oc get kafkatopics
```

The output should look similar to below

```
NAME                                        PARTITIONS   REPLICATION FACTOR
knative-messaging-kafka.sandbox30.messages   1            1
```

This goes to show that the Kafka Topic has succesfully been create with the number of partitions and replication factor that was asked for when creating the channel.

and finally, if you are more comfortable using the Kafka CLI, then you still have access to this within the Kafka statefulset.

Assuming you did not change the name of your Kafka Cluster then the stateful set should be called "my-cluster-kafka-0"

If you did change the name of the cluster, then please change the name of the statefulset accordingly.

Type

```
oc exec -it my-cluster-kafka-0 -- bin/kafka-topics.sh --zookeeper localhost:2181 --list
```

The output should look like the following

```
OpenJDK 64-Bit Server VM warning: If the number of processors is expected to increase
from one, then you should configure the number of parallel GC threads appropriately using
-XX:ParallelGCThreads=N
knative-messaging-kafka.sandbox30.messages
```

It confirms that there is a topic called knative-messaging-kafka.sandbox30.messages defined to the cluster.

and finally, you can check how the topic is laid out in the cluster (Not so exciting as there is only one broker)

Type

```
oc exec -it my-cluster-kafka-0 -- bin/kafka-topics.sh --describe --zookeeper
localhost:2181  --topic knative-messaging-kafka.sandboxX.messages
```

The output should look like the following

```
----OpenJDK 64-Bit Server VM warning: If the number of processors is expected to increase
from one, then you should configure the number of parallel GC threads appropriately using
-XX:ParallelGCThreads=N
Topic: knative-messaging-kafka.sandbox30.messages       PartitionCount: 1
ReplicationFactor: 1    Configs:
        Topic: knative-messaging-kafka.sandbox30.messages       Partition: 0    Leader: 0
Replicas: 0     Isr: 0
----
```

This shows that 1 partition has been created. ISR is zero because the replication factor is 1. The partition is not being replicated. This is not recommended for production. This is purely to limit the resources on our Openshift Cluser.

Lets deploy the second channel called *words* again, this time based on a Kafka Channel

```
cd /workspace/workshop4/camelfiles/streams
```

Create the Knative Eventing channel backed by Kafka as a persistent store

```
oc apply -f words.yaml
```

We wont go through all the checking we did above, lets just type

```
oc get kafkachannels
```

The output should look similar to that below

```
NAME       READY   REASON   URL                                                  AGE
messages   True             http://messages-kn-channel.sandbox30.svc.cluster.local   25m
words      True             http://words-kn-channel.sandbox30.svc.cluster.local      8s
```

We are now going to re-run exactly the same integrations that we used earlier in the lab, as a reminder, this is what the flow looks like

Make sure you are in the "Topology view" within the Openshift console so you can see things deploy and run.

```
cd /workspace/examples/knative
```

Run the Integration that starts everything off

```
kamel run feed.groovy
```

If you look at the log in the running pod for the feed applicaion you should see that messages it's creating (If you cannot remember how to do this, review the instructions for the first part of this lab)

Return to the Topology view

Now run the Splitter Integration

```
kamel run splitter.groovy
```

Take a look at the log file for the splitter Integration, you will notice something a bit weird

The spitter is outputting the message like this :-

```
2020-04-02 16:11:07.990 [32mINFO [m [vert.x-eventloop-thread-0] route1 - sending
"SGVsbG8gV29ybGQgZnJvbSBDYW1lbCBL" to words channel
2020-04-02 16:11:10.814 [32mINFO [m [vert.x-eventloop-thread-0] route1 - sending
"SGVsbG8gV29ybGQgZnJvbSBDYW1lbCBL" to words channel
2020-04-02 16:11:13.840 [32mINFO [m [vert.x-eventloop-thread-0] route1 - sending
"SGVsbG8gV29ybGQgZnJvbSBDYW1lbCBL" to words channel
2020-04-02 16:11:16.611 [32mINFO [m [vert.x-eventloop-thread-0] route1 - sending
"SGVsbG8gV29ybGQgZnJvbSBDYW1lbCBL" to words channel
2020-04-02 16:11:19.636 [32mINFO [m [vert.x-eventloop-thread-0] route1 - sending
"SGVsbG8gV29ybGQgZnJvbSBDYW1lbCBL" to words channel
2020-04-02 16:11:22.657 [32mINFO [m [vert.x-eventloop-thread-0] route1 - sending
"SGVsbG8gV29ybGQgZnJvbSBDYW1lbCBL" to words channel
2020-04-02 16:11:25.676 [32mINFO [m [vert.x-eventloop-thread-0] route1 - sending
"SGVsbG8gV29ybGQgZnJvbSBDYW1lbCBL" to words channel
2020-04-02 16:11:28.697 [32mINFO [m [vert.x-eventloop-thread-0] route1 - sending
"SGVsbG8gV29ybGQgZnJvbSBDYW1lbCBL" to words channel
```

The Kafka client used has serialized the messages into Base64... not so helpful for this little example.

We can fix this by making a small change to the *spiltter.groovy* integration. This will allow the integration to transform the Base64 back to normal ascii text

Type

```
vi spiltter.groovy
```

The integration will look like

```
from('knative:channel/messages')
  .split().tokenize(" ")
  .log('sending ${body} to words channel')
  .to('knative:channel/words')
```

Before it does the ".split", add a new line and insert

```
/.unmarshal().base64()
```

The line above with transform from Base64 to a Java byte array. The integration should now look like

```
from('knative:channel/messages')
  .unmarshal().base64()
  .split().tokenize(" ")
  .log('sending ${body} to words channel')
  .to('knative:channel/words')
```

redeploy the integration

Type

```
kamel run --dependency=camel-base64 splitter.groovy
```

You will notice that the Camel Base64 dependency has been added, this enables Camel K to download the dependancy and package it into the run

Check the pods log again, you should be able to read the text again

If you look at the printer Integration log, you will notice that it has exactly the same problem.

See if you can fix the problem for this integration by following the same steps above. The name of the file is printer.groovy

Accessing the Kafa Topics directly

In the terminal window, type :-

```
oc exec -it my-cluster-kafka-0 -- bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic knative-messaging-kafka.sandboxX.messages --from-beginning
```

There you go, all the messaging sent using Knative eventing channels available in a Kafka topic. This means that all the events flying around the event driven network can also be made available to many other tools supported by Kafka. This could be big data systems, AI systems, mirroring events to different Kafka Clusters and many more thinks. Of course, the biggest benefit is that the events are now persisted, partitioned for scale, and replicated using classic Kafka functionality. == Camel K and Openshift Serverless Serving

Author: Phil Prosser (feedback to pprosser@redhat.com)

Introduction

OpenShift Serverless

Openshift Serverless, based on Knative is the serverless technology that was introduced in Openshift 4.2. Openshift Serverless enables Pods running on Openshift to be scaled to 0 therefore taking zero processing power. Only when called, Openshift Serverless will scale the Pod up on demand before processing the request. Openshift Serverless also has the ability to autoscale based on load before eventually scaling back to zero when no requests are being received.

Openshift Serverless supports "Serving" and "Eventing"

At the time of writing, Serving is in technology preview, and Eventing is in Developer Preview

"Serving" enables request/response workloads, and Eventing enables asynchronous event based workloads using cloudevents. In this lab, we are going to look at serving, and how easy it is to integrate with Camel K.

Check the project is ready to use

In the browser based terminal window, check you are still logged on and using the correct project by typing:

```
oc whoami
oc project
```

If the response from the commands indicates that you are not userX (where X is your assigned user number) and not using the project sandboxX please repeat the commands in the pre-requisites.

Openshift Serverless and the Operator Lifecycle Manager

Openshift Serverless and the Operator Lifecycle Manager

Openshift Serverless uses the Operator Lifecycle manager, this means that its operator and Custom Resource Definitions (CRDs) will be added to Openshift via "OLM". Once created, the new CRDs will extend the Openshift data model allowing Openshift Serverless to be managed using the standard ‘oc’ command. Installing operators requires a higher cluster privilege so the presenter will have already set these up for you.

Creating the pre-requisites for the chapter

Before we can create an integration, we need to check that the camel-k integration added in a previous chapter is still active

In the terminal window, type

```
cd /workspace/workshop4/camelfiles/camelkplatform
oc apply -f integrationplatform.yaml
```

This will either create the integration platform or, if it is still active, indicate it is unchanged

now type

```
oc get integrationplatform
```

The output should look similar to below

```
NAME      PHASE
camel-k   Ready
```

Once the "Phase" says "Ready", you can continue

Deploy a Restful Integration using Camel K and an OpenAPI definition

Introduction to the integrations that we will use

This integration is a simple hello world Restful integration. The Restful API interface is defined by the OpenAPI json document called greeting-api.json. This is demonstrating top down API design with Apache Camel. An Apache Camel Route can implement this interface without the need for code. Using a "from" action called "direct" the integration enables the Camel Route to listen/respond to the operation id defined in the OpenAPI document.

Feel free to have a look at both of the files to understand whats going on.

In the terminal window, the files are located in the directory /workspace/examples

OpenAPi definition: greetings-api.json Camel K integration: greetings.groovy

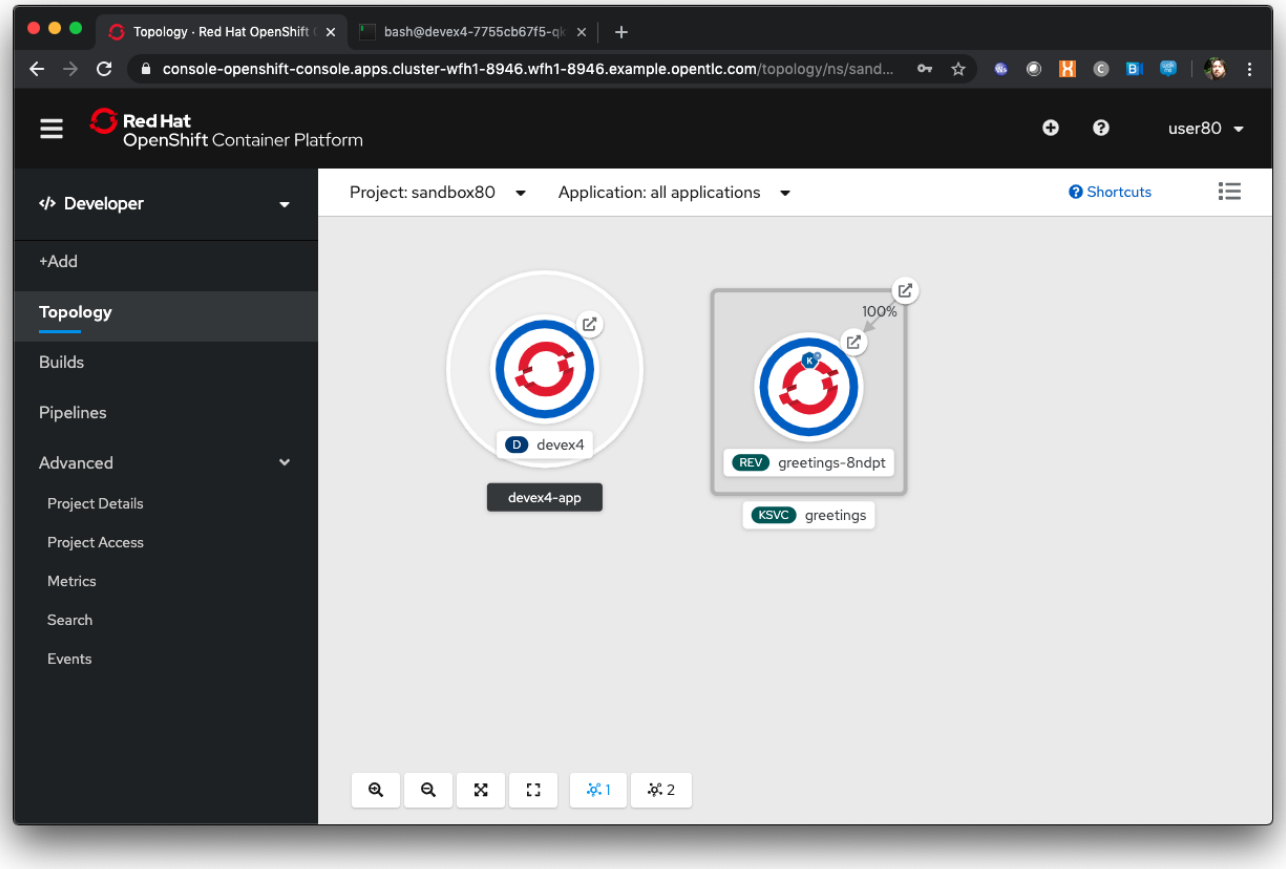
In the terminal window, deploy the integration

```
cd /workspace/examples
kamel run --name greetings --dependency=camel-rest --dependency camel-undertow --property
camel.rest.port=8080 --open-api greetings-api.json greetings.groovy
```

Go to the developer view in the Openshift Console

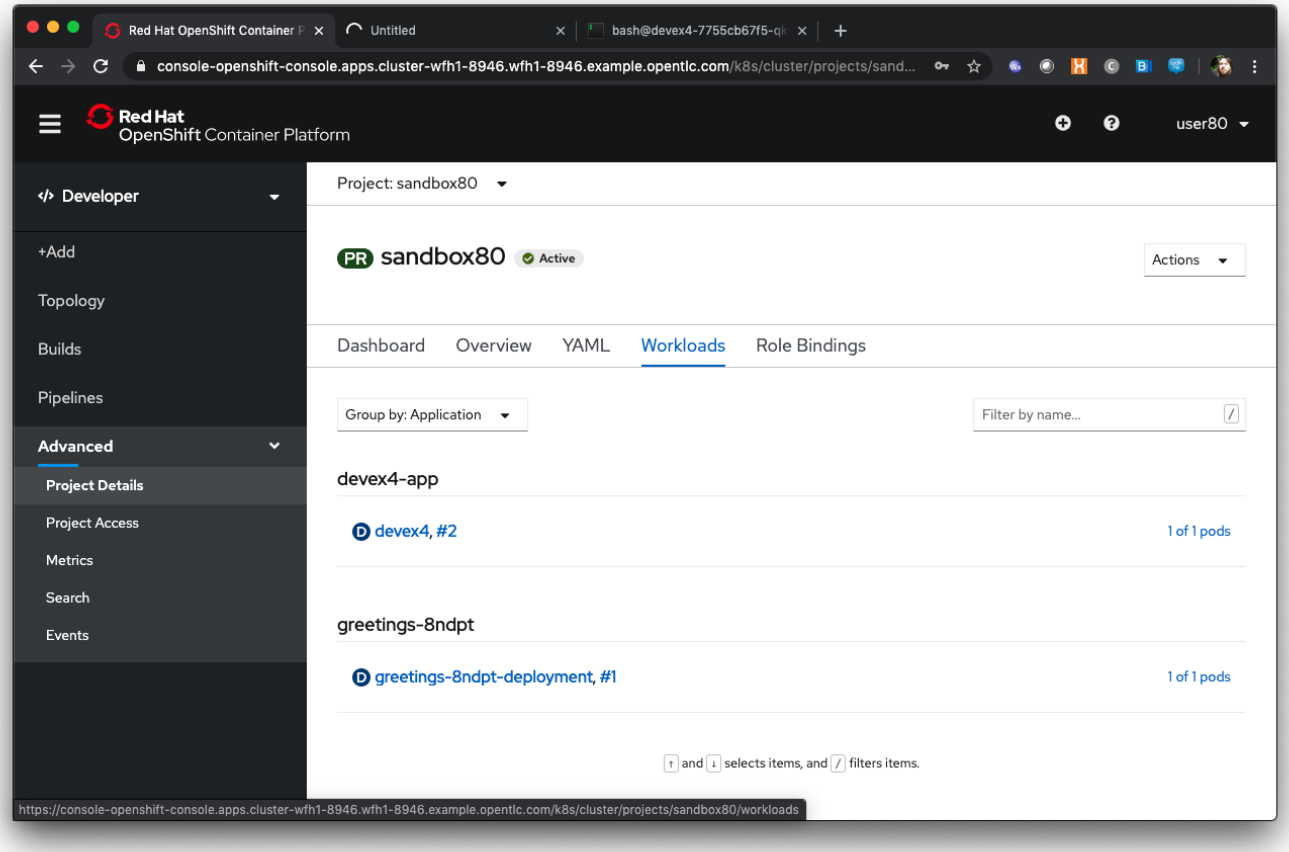
If this is the first time you've deployed a Camel K integration, it will take a few minutes to download the dependencies

Once, the Integration is deployed, the topology view should look like the scrennshot below



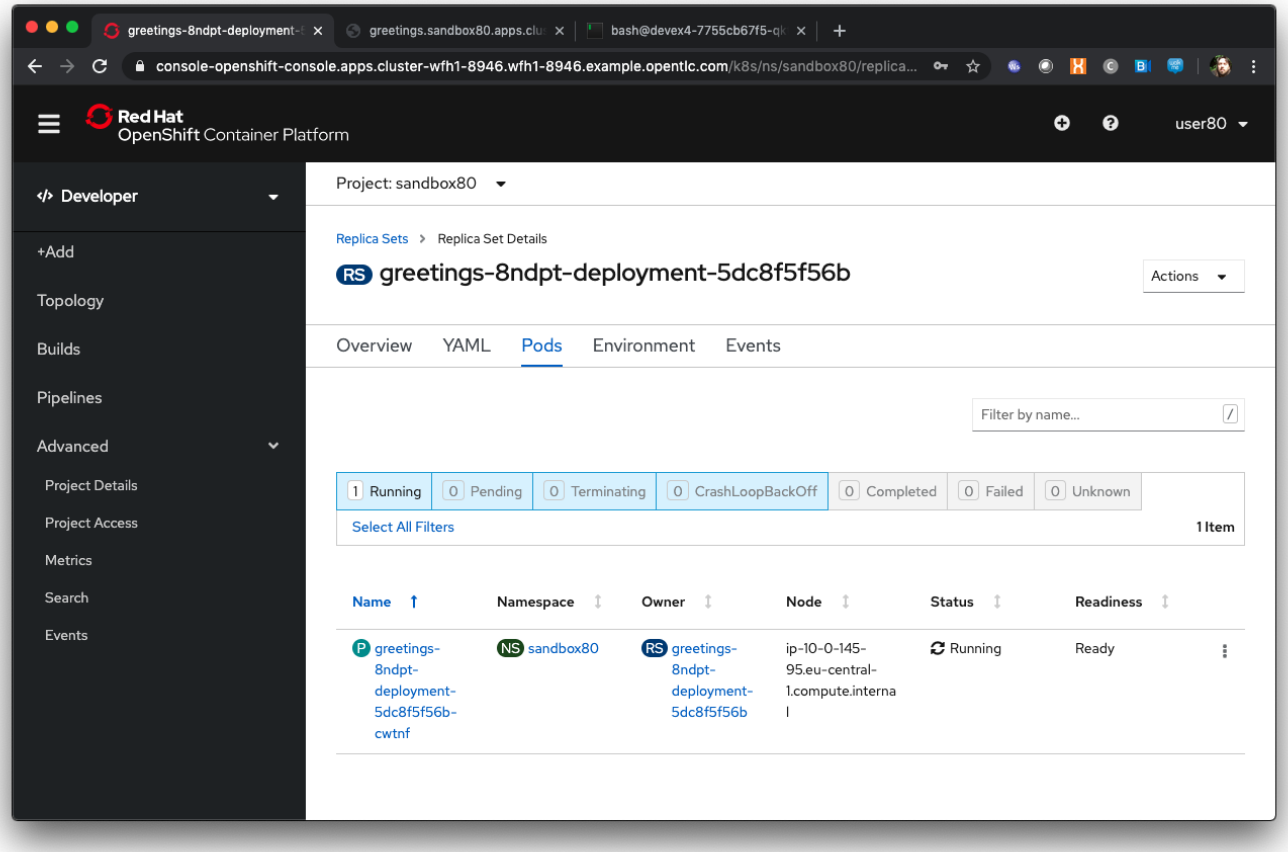
Camel K has detected that Openshift Serverless is installed and automatically deployed the integration as a Knative serving (because it's API based) service.

At the time of writing, there is no easy way to view the pod log files of a knative service in the console, so in the Developer view click on Advanced/Project Details and select Workloads. The interface should look like:



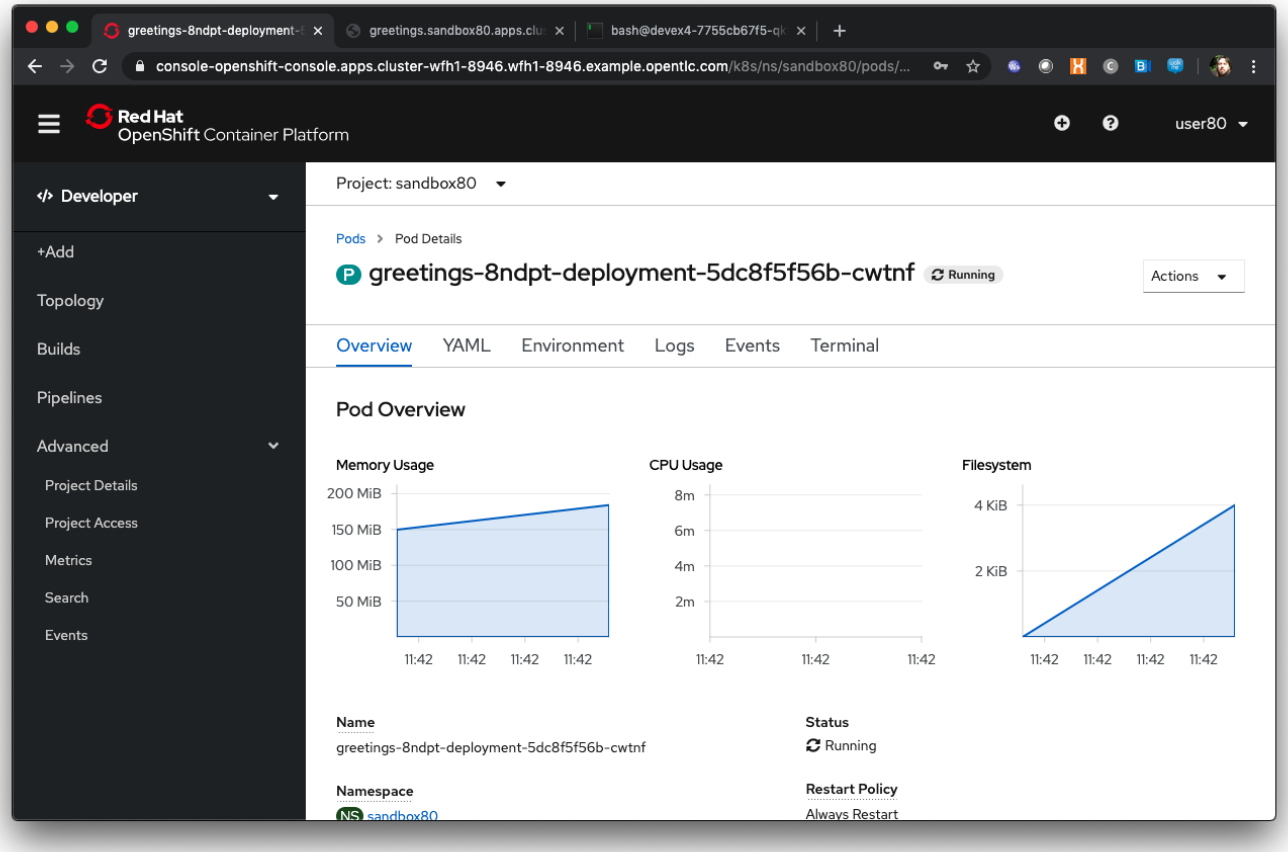
For the greetings integration, you should see a *1 or 1 pods* on the right hand side. *Click* on it.

You should see a screen similar to the one below

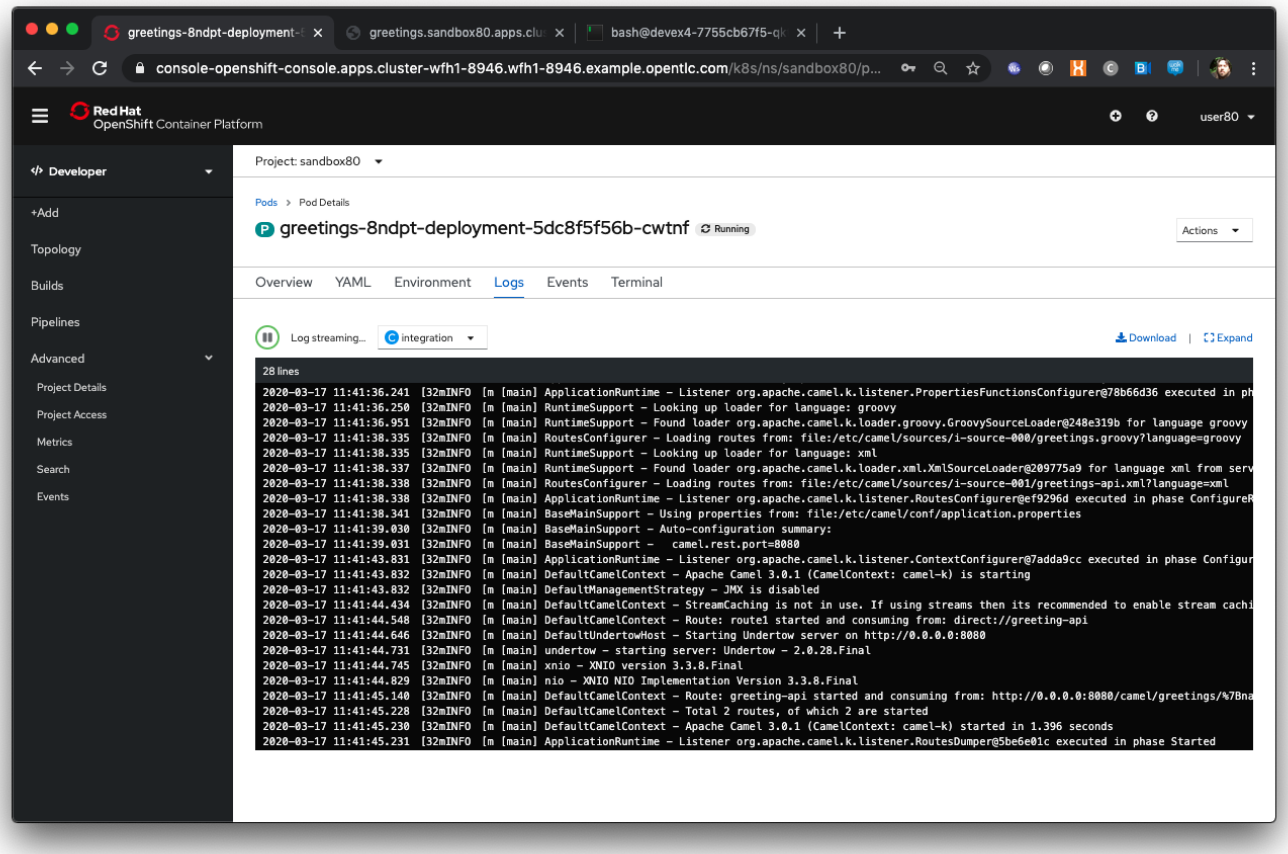


Click on the row name e.g. greetings-xxxxxxxxxxxx

This should show you a screen similar to the one below



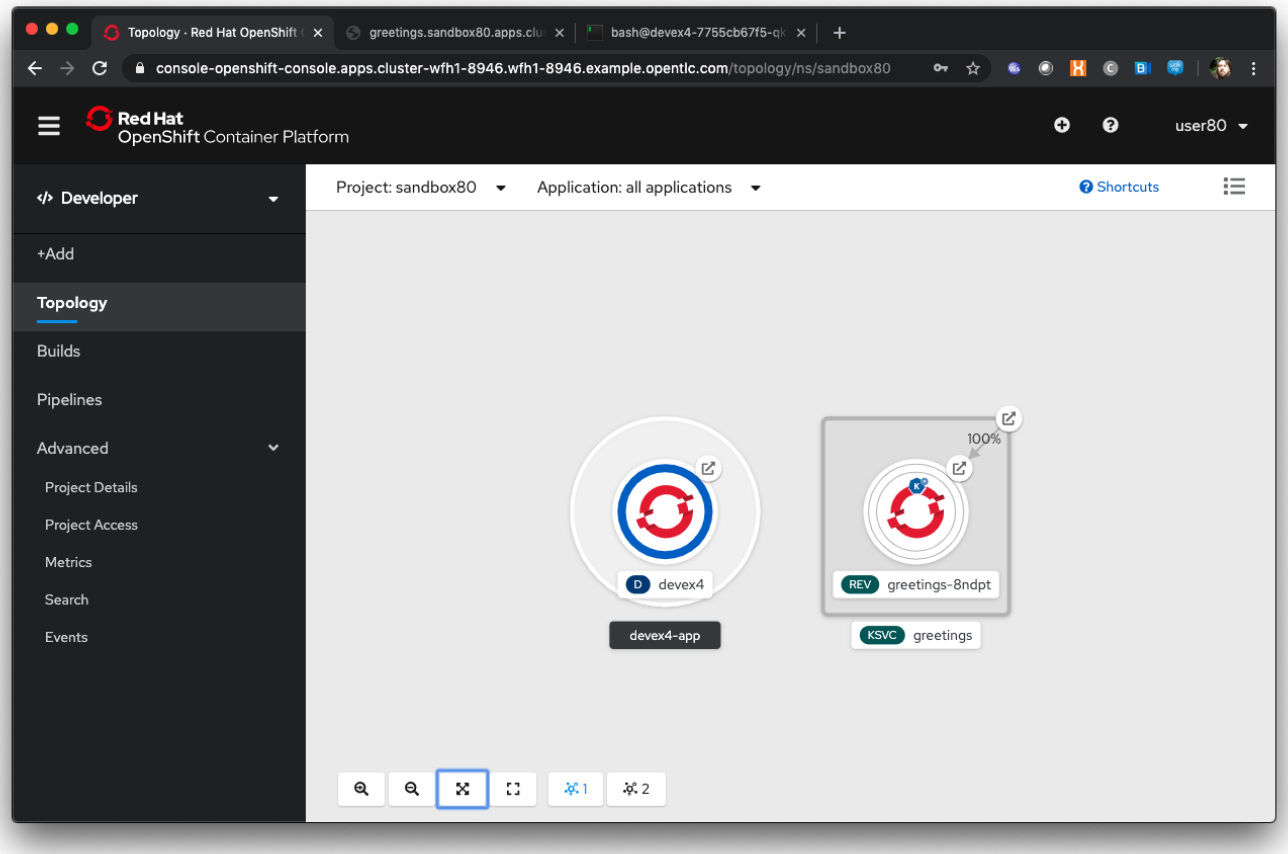
Click on *Logs* to view the log for the pod. It should look something like the one below



Knative in action

Following the steps above, you might find that the pod you are looking for is not there. That's because Knative Serving has scaled the pod back to zero.

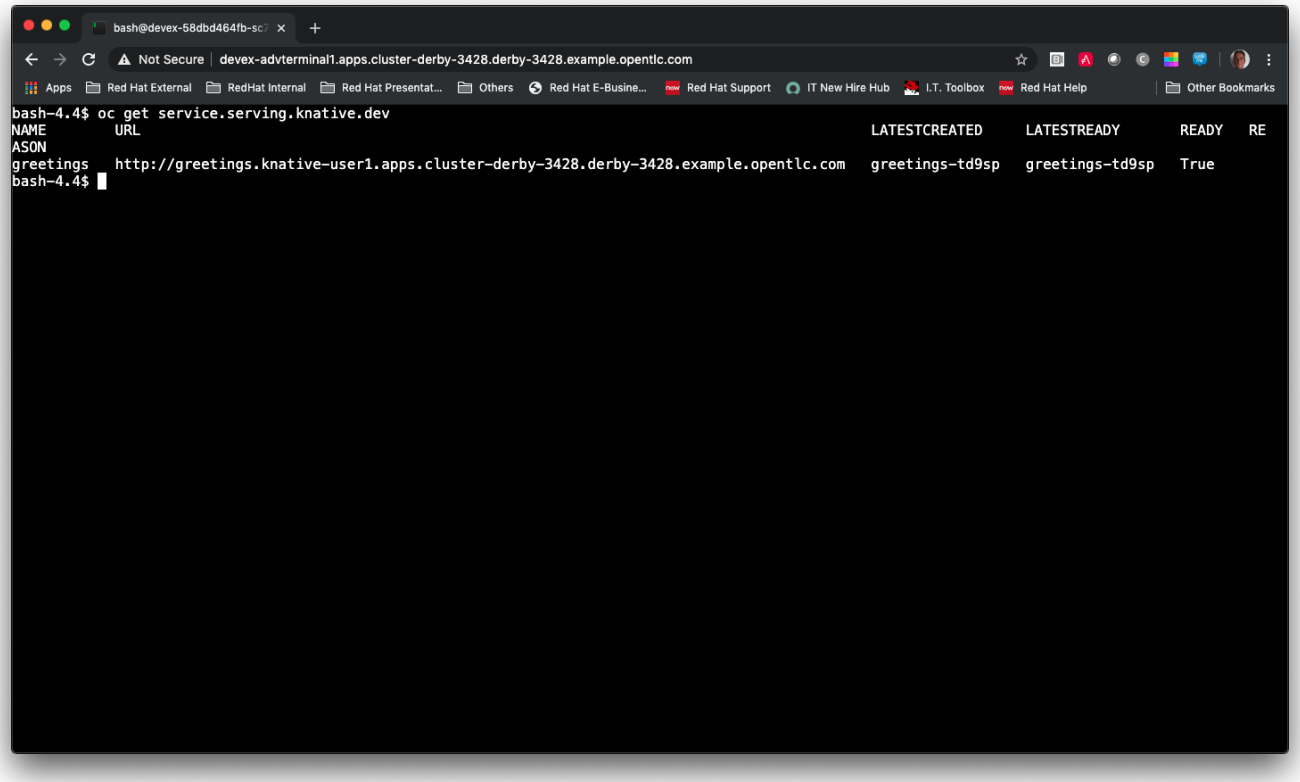
With the Openshift console switch to developer view, vewing the toplogy. Your screen might look like the one below. If it doesn't, wait for up to 30 seconds and it scale down to zero. By default, Knative scales down an inactive pod after 30 seconds.



Next we are going to make a call to the greetings API. To do this, you need to find out the name of http route. To do this, in the terminal window, type in the following command

```
oc get service.serving.knative.dev
```

You should see something similar to the picture below

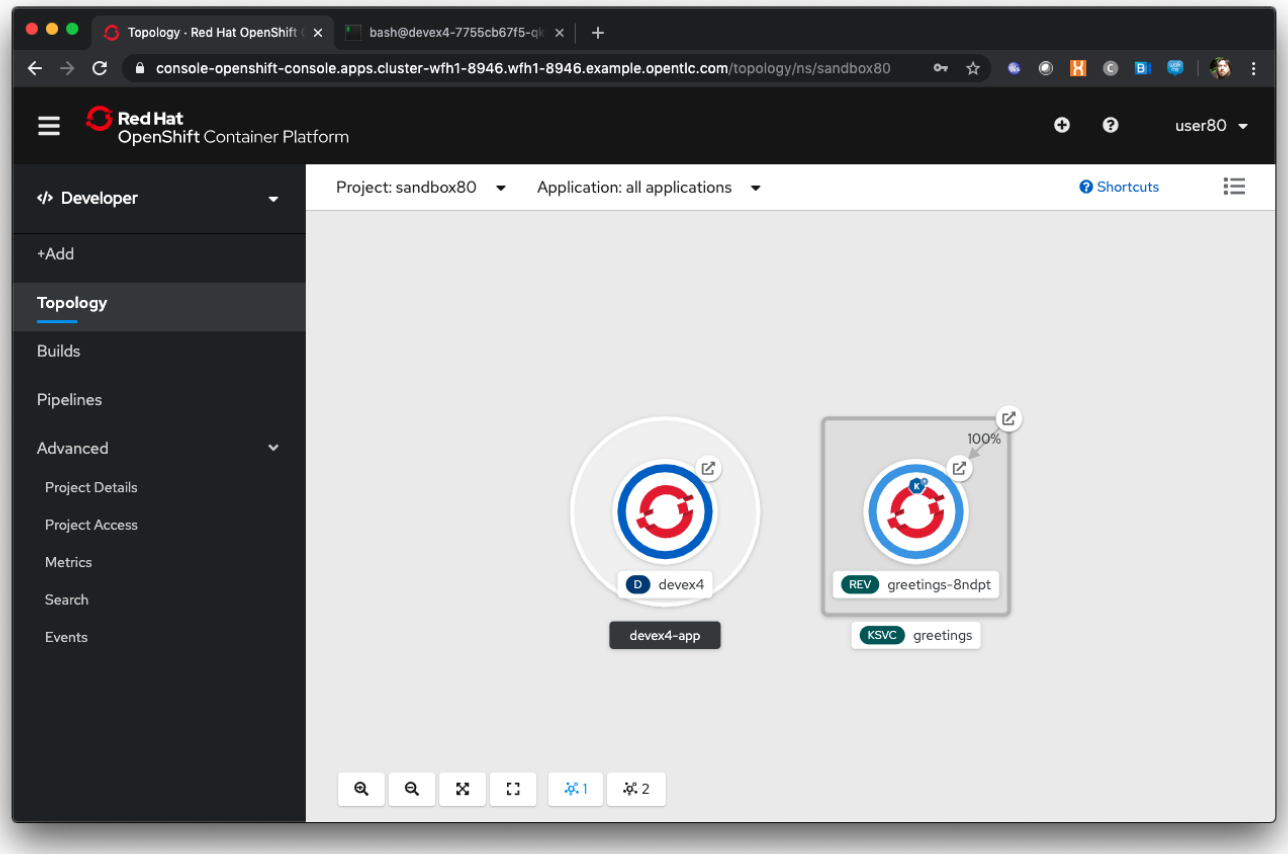


In the terminal window type (make sure you substitute the URL you see from running the command above). Note we are adding the /camel/greetings/YOURNAMEHERE bit to the URL

```
curl -m 60 URLFROMABOVE/camel/greetings/YOURNAMEHERE
```

After a few seconds you should get a response, and the topology view should now look similar to the picture below. The dark blue circle indicates that the service is now executing.

At the time of writing, the inital start up time issue is known to engineering and is documented in the Red Hat documentation. A fix for this will be coming in a future release. Please remember that Knative serving is still in Tech Preview



Knative has automatically scaled the service to one pod, and processed the curl request.

Options are available in Knative to determine how to scale based on concurrent calls or cpu useage. Options are also available to determine maximum number or pods, and also the inactivity time before a pod scales its self down - by default, all the way back down to zero.

Deploying Integration Services with Knative can't get easier than that!

To clean up before the next chapter run the following commands in the terminal:

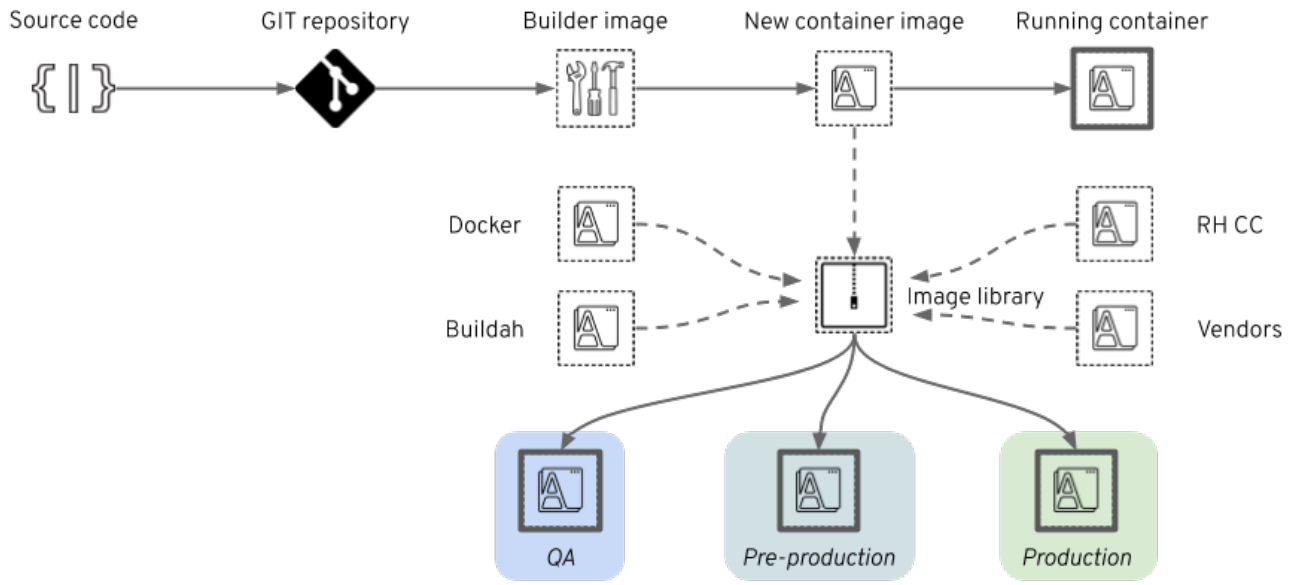
```
kamel delete greetings
```

OpenShift and the Quay Image Repository

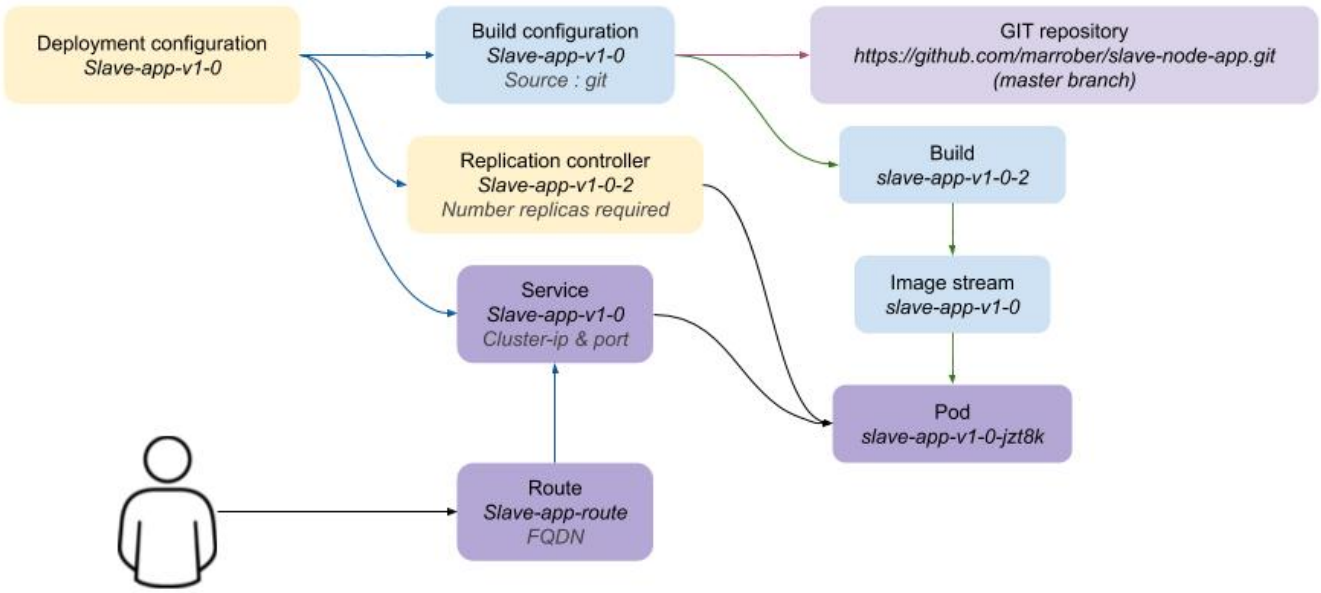
Author: Mark Roberts (feedback to mroberts@redhat.com)

Introduction

During the development process a number of container images will be produced. The source-2-image capability described in the basic workshop creates images catalogued under an image stream within a project. Other container images may be created using command line functions such as docker or buildah (both using docker files potentially) or images may be downloaded from the Red Hat Container Catalogue here, or other vendor supplied images may be used. From a variety of sources and through a variety of mechanisms images will be created that form the microservices of an application. Such images need to be managed through a lifecycle as they progress from Development to QA and finally to production in a process described in the diagram below. The purpose of Red Hat Quay is to provide a repository for the secure and structured storage of images that may be accessed by multiple users from multiple clusters. Quay also provides an image vulnerability scanning capability to identify issues with images stored within it.



The image below shows an OpenShift project and the various artefacts that are created. The image stream object is shown here within the context of the builder process that creates it and the running pod that consumes it.



The structure of Quay

The Quay image repository uses a number of specific objects to manage the structure, security an maintenance of images.

1. Organisation - An organisation maps neatly to a project within OpenShift and is a structural container for a number of repositories.
2. Repository - A repository maps neatly to an image stream within OpenShift (or any other single source of images). A repository may contain multiple tagged versions of a single image.

The organisation may have the following characteristics:

1. Default permissions to be applied to any new repositories created within the organisation.
2. Teams that control access to the organisation and its repositories.
3. Robot accounts for automated access to the repositories from scripted or command line processes.

The objectives of this chapter are:

1. To understand the Quay user interface and the objects created within Quay
2. To create a process for pushing images from the OpenShift image repository to the Quay repository
3. To pull images from Quay to a new project in OpenShift to simulate the use of the images in a QA cluster

Using the Quay workshop cluster

In order to use the Quay instance for the workshop you need to request that an id is created for you by the workshop administrators. This will take very little time and will result in the creation of the following:

1. An organisation will be created in Quay using the same name as the OpenShift project used in this section, for example master-slave-user23.
2. A Quay user matching the OpenShift username and password.
3. The Quay user will have read/write access to the organisation, but you will have to create access to the specific repositories as part of the steps to follow.

Creating the OpenShift Project

In order to interact with Quay each user requires an OpenShift project containing images. While the administrator is creating your Quay content use the commands below to quickly create the OpenShift project and the required content:

```
oc new-project master-slave-userXX

oc new-app --name node-app-slave  https://github.com/marrober/slave-node-app.git

oc new-app --name node-app-master  https://github.com/marrober/master-node-app.git

oc expose service node-app-master --name="master-app-route"
```

where XX is your user number

The above commands will create image streams for the master and the slave which can be viewed using the command below.

```
oc get is
```

This will display image streams similar to that which is shown in the table below:

NAME	IMAGE REPOSITORY	TAGS	UPDATED
node-app-master	default-route-openshift-<reduced>.com/master-slave-userX/node-app-master	latest	22 hours ago
node-app-slave	default-route-openshift-<reduced>.com/master-slave-userX/node-app-slave	latest	22 hours ago

The image repository field will show the route to use to get access to the container images within the

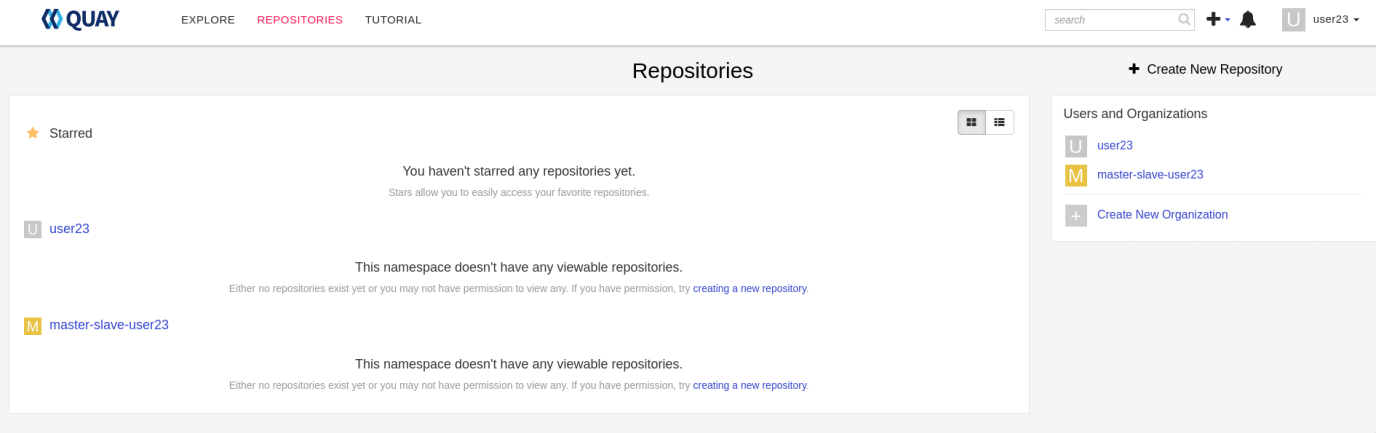
container image repository. Now that you have some images to manage it is necessary to create the repositories in Quay.

Creating the Quay image repositories

When the workshop administrator tells you that they have created your user and organisation in Quay you may login at the provided Quay URL. The username and password for Quay will be the same as those used for OpenShift.

The example screen shots below show a user ID of user23, but obviously yours may be different.

When you have logged into Quay you will see the web user interface similar to that shown below.

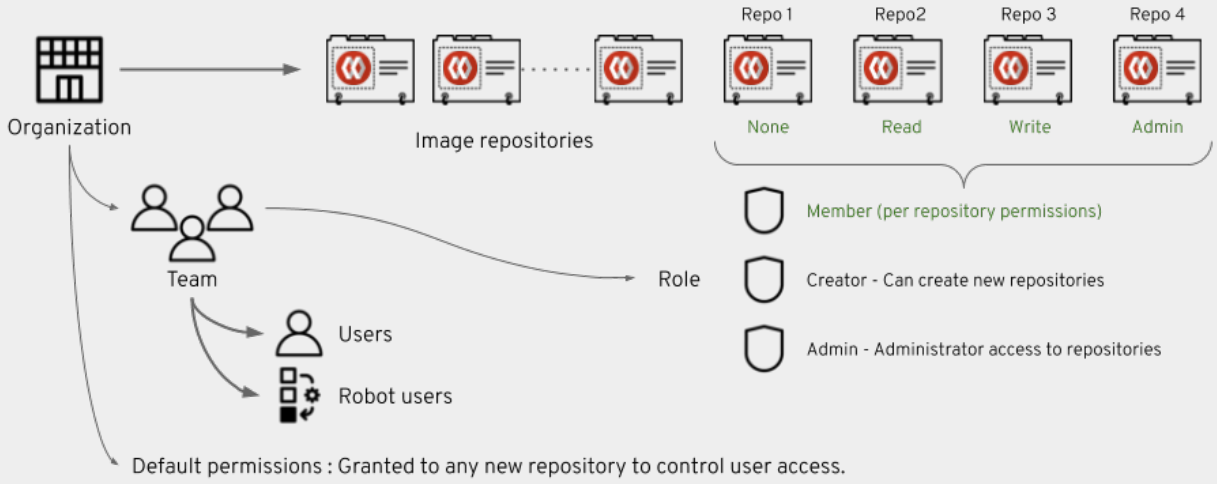


You see two namespaces within Quay for the storage of repositories. The namespace identified by your user name can hold repositories outside the scope of an organization. The organization can hold repositories with the added benefits provided by the organization which are :

1. Default permissions may be set on the organization that apply to new repositories created within it.
2. The use of teams and users to manage the allocation of permissions to the organisation and the repositories created within it.

Quay permissions

The diagram below shows the permission structure for an organization.



1. Teams contain users and robot (automation) users who have permissions on specific repositories.
2. Teams can have a role of *Creator* - Create new repositories which will inherit the organisation default permissions)or *Admin* - perform all actions including delete, or *member* - permission is indicated by each repository.
3. Member permissions are allocated on a per repository basis and can include *none* - no access, *read* - can pull images, *write* - can pull and push images, *admin* - can pull, push and perform admin functions such as delete and manage permissions.

Managing the Quay organization and repositories

Select the Quay organisation called master-slave-userXX.

This will show a largely blank screen with options down the left hand side.

Click on the + *Create New Repository* link at the top right of the screen.

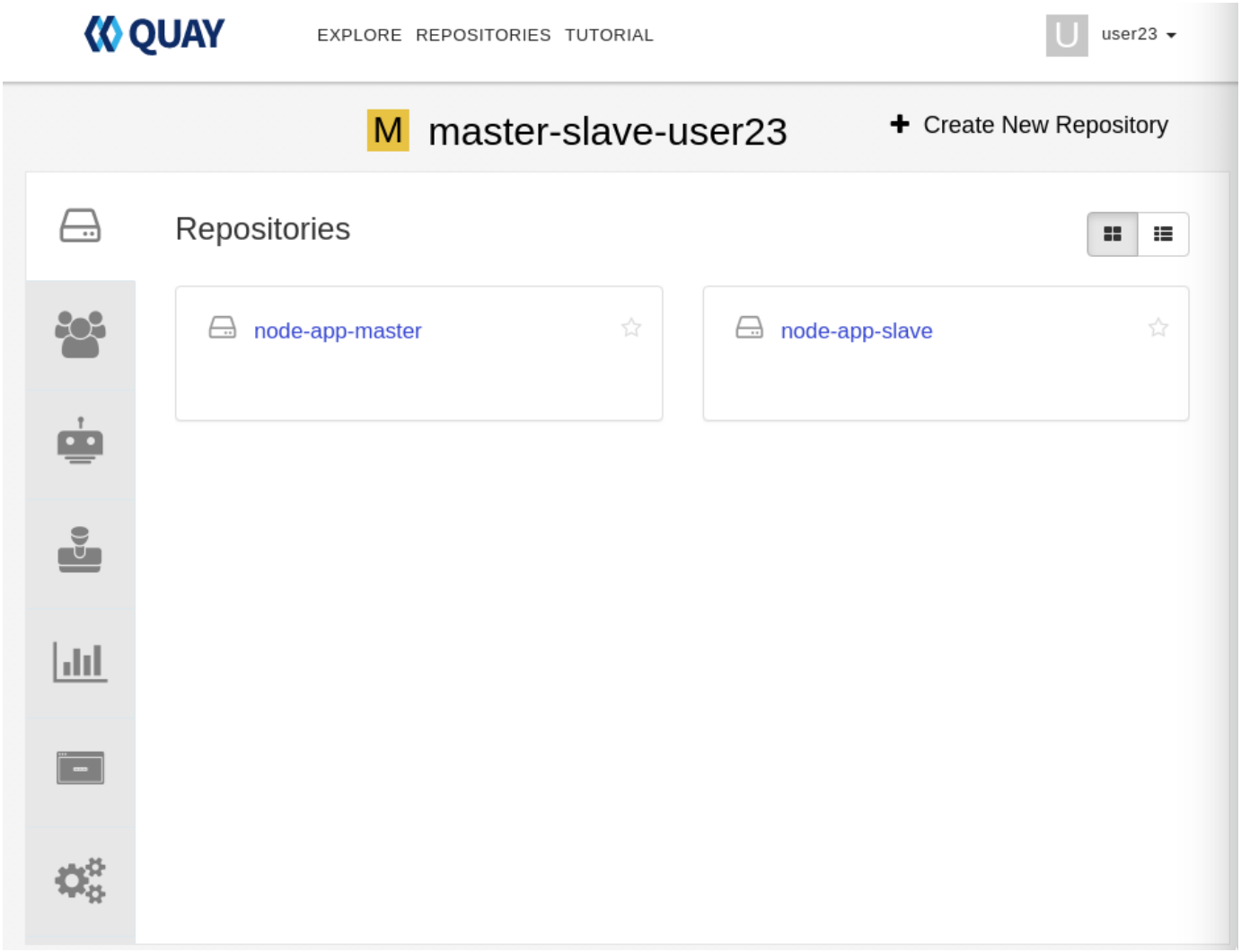
Since the names of the repositories need to match the names of the image streams in OpenShift refer back to the command line window and the names of the image streams from the *oc get is* commands used above.

Enter the name of one of the image streams for the repository, select public for access and then click on *Create Public Repository*.

Click the browser back button to go back to the repository creation screen and repeat the repository creation process for the second image stream.

Press the left facing arrow on the top left of the screen to go back to the list of repositories.

Select the master-slave-userXX organization and you should see the details of the organization as shown below.



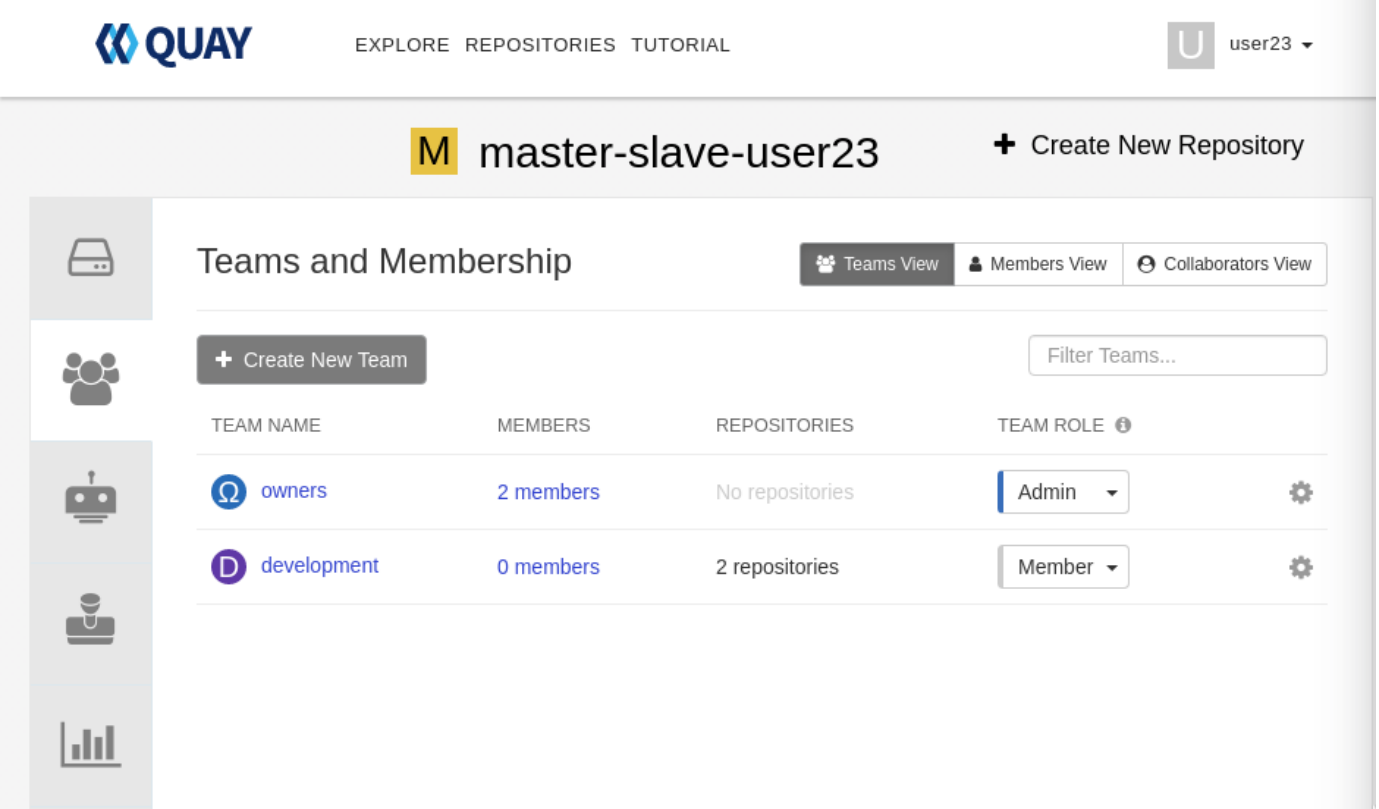
Granting permissions to repositories

Select the Teams and Membership tab on the left hand side of the screen (2nd icon down). Here you can create new teams and manage the users and permissions of existing teams.

Create a new team called *development* (only lower case letters and numbers are allowed).

You will then be prompted to add permissions for the two existing repositories. Select *Write* permission for both repositories.

When the permissions have been added for the development team you will see the summary for teams and memberships as shown below.



At this point the development team has no members so click on the link stating *0 members* and add userXX to the team, by typing the user name into the *add user* field on the right hand side. Press the left pointing arrow at the top left corner to return to the organization and you should see that the development team has 1 member and 2 repositories.

Creating a robot account

Click on the next tab down from the teams and memberships tab on the left hand side of the screen to select Robot accounts. Create a new robot account called userXX_automation (where XX is you user number). You may optionally add a description if you want to.

Grant write permission to the robot account on both repositories and then click *close*.

Click on the cog on the right hand side of the robot account name and select *view credentials*.

You will see a list of many different types of credentials that you can generate such as token, Kubernetes secret, rkt configuration, Docker login, Docker configuration and Mesos credentials. For the access required in the workshop copy the username and token from the Robot Token tab and store them in a local editor or notepad ready to use later. Once they are copied close the dialog box.

Back on the organisation screen take a look at the options for creating default permissions (the next tab down on the left). It is possible to create default permissions to be applied to new repositories for specific uses, teams and robot users as appropriate.

Summary of Quay UI work

The organization, repositories, user, robot user and permissions are all now in place within Quay for the images to be pulled from OpenShift and pushed to Quay.

Pulling OpenShift images and pushing to Quay

Buildah will be used to pull images to a local repository, re-tag the images for the location on Quay and then push the images to Quay.

Image management tools

A number of tools exist for the management of images, three of which are described below.

Buildah is an image building open source project that can either use Buildah specific commands to build an image or it can simply use an existing docker file. One major advantage of Buildah for some users is that it does not require a docker process to be constantly running on the workstation as root. In the workshop Buildah will be used to get images from / to OpenShift and from / to Quay.

Podman overlaps somewhat with Buildah but its main focus is with regard to the running and interaction with container images.

Skopeo can be used to copy container images from one image repository to another. It can also be used to convert images between formats. It is possible to perform many of the actions in this workshop with Skopeo but by using Buildah it is possible to see what is being created in an intermediate local repository which may add some value for users.

Login to the OpenShift registry using Buildah

In order to pull the images it is necessary to login to the OpenShift image repository using the Buildah command even though you may already be logged into the OpenShift cluster using the oc command. The URL for the OpenShift repository is the address in the image repository table up to and including .com.

To get just the image repository URL use the command :

```
oc get is -o jsonpath={.items[0].status.publicDockerImageRepository} | cut -d'/' -f1
```

This will return a string similar to :

```
default-route-openshift-image-registry.apps.cluster-wfh1-8946.wfh1-
8946.example.opentlc.com
```

The Buildah login command takes the form :

```
buildah login --username <username> --password <token> repository-URL
```

The token for the login command will be generates from the command :

```
oc whoami -t
```

Combined together the Buildah login command (for the example repository-URL, and where XX is replaced by your user number) becomes :

```
buildah login --username userXX --password $(oc whoami -t) default-route-openshift-image-
registry.apps.cluster-wfh1-8946.wfh1-8946.example.opentlc.com
```

You should get a response of "Login Succeeded!"

Login to Quay using Buildah

It is also necessary to login to the Quay image repository using the Buildah command so that images can be pushed to Quay. The URL for the Quay repository is the address in the browser window for Quay up to and including .com and excluding the https:// part.

The username and password are those which were generated and noted earlier on for the Quay robot user.

The Quay login command will be similar to :

```
buildah login --username master-slave-user23+user23_automation --password
6A6ODEQT39ID52S9HZ4IRCBO3EK4O5KNAGZ2HWKSOQQUMU9QSKMBBPYNO6A3ED0O quay-b2b3.apps.shared-
rhpds.rhpds.openshift.opentlc.com
```

You should get a response of "Login Succeeded!"

You are now logged into both OpenShift and Quay with buildah and you are ready to pull and push images.

Examine the local buildah repository

Use the command below to view the local buildah image repository. You should see that it contains no images.

```
buildah images
```

Use the command below to list the images and their location within the OpenShift image repository :

```
oc get is -o jsonpath='{range.items[*]}{.metadata.name}{"
"}{.status.publicDockerImageRepository}{"\n"}'
```

This command will generate a list of all image streams and the registry location to use in the pull command. To pull the image use the full docker image repository name in the command below :

```
buildah pull docker://<full-image-path>
```

for example

```
buildah pull docker://default-route-openshift-image-registry.apps.cluster-wfh1-8946.wfh1-
8946.example.opentlc.com/master-slave-user23/node-app-master
```

The command will show the progress of pulling image layers and will complete with a message similar to that which is shown below :

```
Getting image source signatures
Copying blob 455ea8ab0621 done
Copying blob 6a4fa4bc2d06 done
Copying blob bb13d92caffa done
Copying blob 2dd72bf14df1 done
Copying blob ff52b8e1303b done
Copying blob 84e620d0abe5 done
Copying config abc6f7ad19 done
Writing manifest to image destination
Storing signatures
abc6f7ad19646ed135d9b76946ccce2ae9b4c796a66472f34d853df009dbd18e
```

View the local image repository with the command :

```
buildah images
```

The result will be similar to that which is shown below:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
default-route-openshift-image-registry.apps.cluster-wfh1-8946.wfh1-8946.example.opentlc.com/master-slave-user23/node-app-master	latest	abc6f7ad1964	6 hours ago	547 MB

An image now exists in a local repository - either on your laptop or within the terminal container depending on where you ran the command.

Repeat the process to pull the image for the slave too. Notice this time that some of the layers are skipped as those layers already exist within the local repository.

Tagging images for the Quay repository

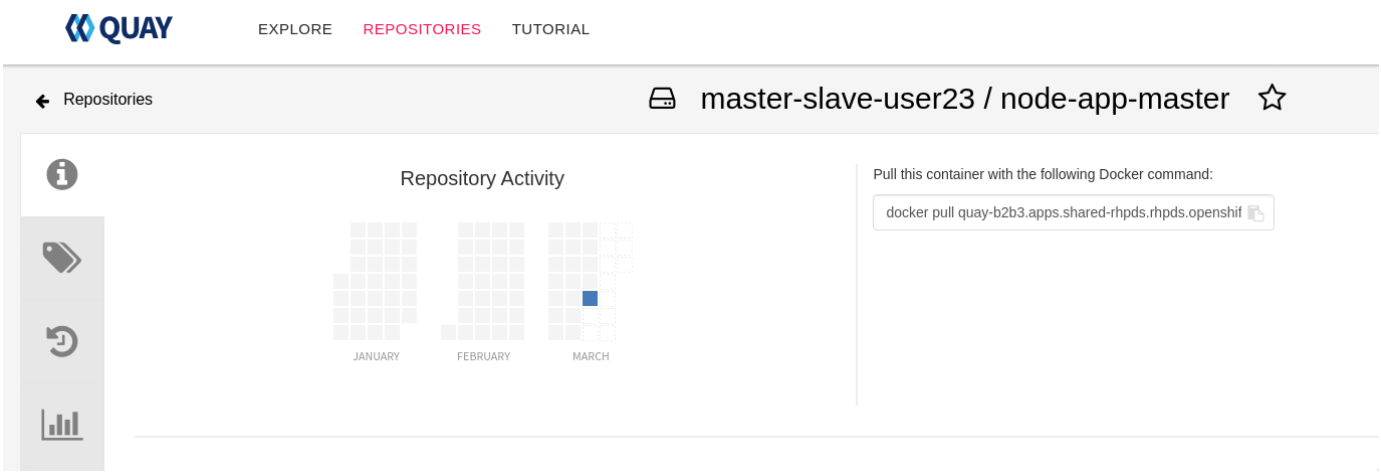
In order to push images to Quay they must have a repository identifier and tag attached to them. This is done using the Buildah tag command. The Buildah tag command takes the format :

```
buildah tag <existing-repository-location>:<tag> <new-repository-location>:<tag>
```

The actual tag names used for the existing location need to match what is in the repository, while the new tag can be whatever is appropriate such as an incremental number, *latest* or some other useful identifier. To reduce the amount of command line copy and paste operations when creating the existing repository location and tag the command below can be used :

```
oc get is -o jsonpath='{range.items[*]}{.metadata.name}{"
"}{.status.publicDockerImageRepository}{":"}{.status.tags[0].tag}{"\n"}'
```

The new repository location is the Quay URL address, organization and repository name. The easiest way to get this is to go to the Quay web user interface, select *Repositories* on the top menu and then select the master repository within the master-slave organization. This will show a screen similar to that which is shown below:



Under the heading "Pull this container with the following Docker command:", copy the URL after the *docker pull* text in the text field. It will look similar to the below:

quay-b2b3.apps.shared-rhpds.rhpds.openshift.opentlc.com/master-slave-user23/node-app-master

Create the Builah tag command from the information collected above such that it looks similar to the below:

```
buildah tag default-route-openshift-image-registry.apps.cluster-wfh1-8946.wfh1-
8946.example.opentlc.com/master-slave-user23/node-app-master:latest quay-
b2b3.apps.shared-rhpds.rhpds.openshift.opentlc.com/master-slave-user23/node-app-master:1
```

Note that the tag used in the command for the destination tag is 1.

Execute the command and then use the command below to list the images :

```
buildah images
```

Repeat the similar command for the slave image.

Push the images to Quay

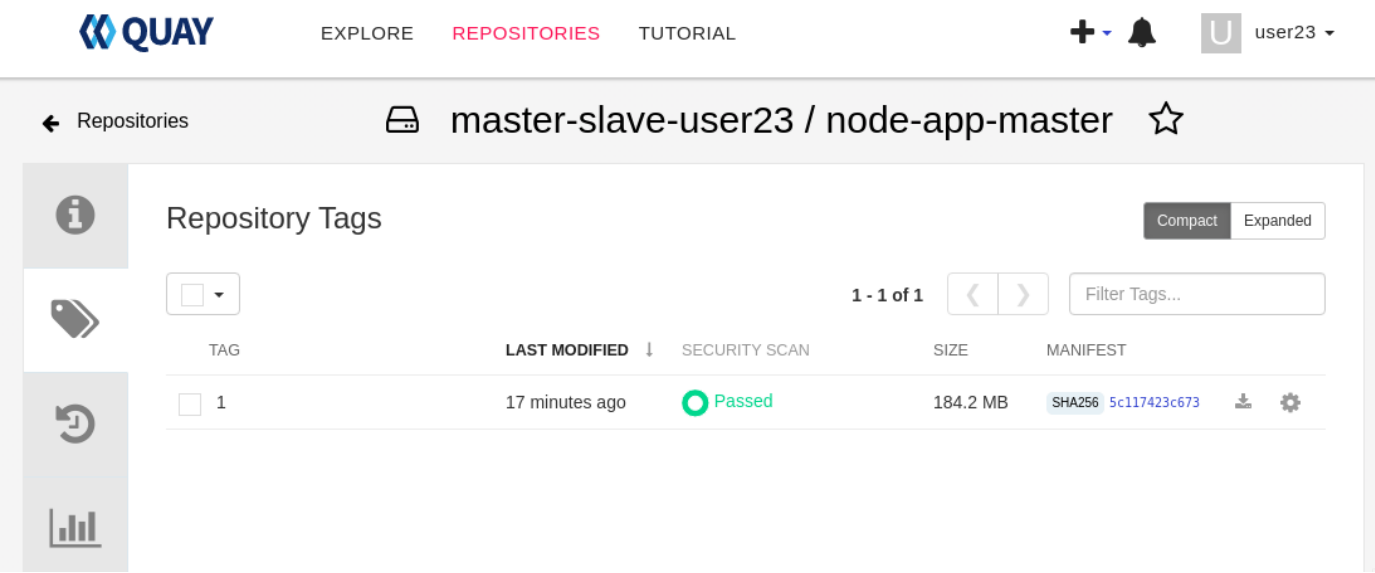
Push the images to Quay using the commands of the format :

```
buildah push <new-repository-location>:<tag>
```

for example :

```
buildah push quay-b2b3.apps.shared-rhpds.rhpds.openshift.opentlc.com/master-slave-
user23/node-app-master:1
buildah push quay-b2b3.apps.shared-rhpds.rhpds.openshift.opentlc.com/master-slave-
user23/node-app-slave:1
```

Switch to the Quay web user interface. If you are still displaying the repository information page where the image pull / push URL was copied from then refresh the browse window and then select the *tags* view (2nd option down on the repository menu). This will show the tags view similar to that which is shown below.



The tags view shows information on the image tag and the buttons on the right of each line allow the user to select different mechanisms for extracting and manipulating the image.

Using the images in a QA environment

Referring to the image at the top of this section the image may now be pulled to different clusters such as a QA cluster, pre-production cluster and production cluster. Specific users will have the appropriate role based permissions to pull the images into those clusters to control the necessary separation of responsibilities within an organization. For this exercise you will create a new project with the same name as the existing project but with -qa on the end of the name to simulate the deployment to QA.

The original commands used to create the images at the start of this section used the source-2-image capability and pulled the source code. The process from this point forward has no interaction with the application source code and pulls the immutable images into each successive cluster (simulated in the case of the workshop), with environment specific information being injected into the running containers using config maps. This use of immutable images is one significant advantage of containers and hence is another reason for the use of a secure image repository.

Creating the OpenShift Project for QA

Use the commands below to create the OpenShift project using the content from Quay as the source:

```
oc new-project master-slave-userXX-qa

oc new-app --docker-image=<master image URL & tag that was pushed above> --name=node-app
-master
oc new-app --docker-image=<master image URL & tag that was pushed above> --name=node-app
-slave
oc expose service node-app-master --name="master-app-route"
```

where XX is your user ID

For example:

```
oc new-app --docker-image=quay-b2b3.apps.shared-rhpds.rhpds.openshift.opentlc.com/master
-slave-user23/node-app-master:1 --name=node-app-master
oc new-app --docker-image=quay-b2b3.apps.shared-rhpds.rhpds.openshift.opentlc.com/master
-slave-user23/node-app-slave:1 --name=node-app-slave
oc expose service node-app-master --name="master-app-route"
```

To test the application get the route with the command:

```
oc get route -o jsonpath='{.items[0].spec.host}{"/ip\n"}'
```

Then issue the following curl command :

```
curl -k <url from the above command>
```

The response should be the ip address of the master node and the slave node similar to that which is shown below:

```
"master ip address 10.131.0.174    ----> slave ip address 10.128.2.157 v1.0"
```

Cleaning up

Finally, lets clean up the project by typing

```
oc delete project master-slave-userXX
oc delete project master-slave-userXX-qa
```

where XX is your user ID

This will delete the projects