# Knative Tutorial

Red Hat Developer Experience Team

Version 0.0.1

# Table of Contents

Overview 1

Setup 2

- Prerequisite CLI tools 2
- Download Tutorial Sources 2
- Minishift 2
- Enable Admission Controller Webhook 3
- Knative Setup 4
- Configuring OpenShift project for Knative applications 7

Basics and Fundamentals 8

- Prerequisite 8
- Build Containers 8
- Deploy Service 8
- Invoke Service 10
- See what you have deployed 10
- Deploy new Revision Service 11
- Pinning service to a revision 13
- Cleanup 14

Configurations, Routes and Traffic distribution 15

- Prerequisite 15
- Build 15
- Deploy Configuration 15
- Invoke Service 17
- Deploy Route 17
- See what you have deployed 18
- Deploy new Revision Service 19
- Distributing traffic 20
- Cleanup 22

Scaling 23

- Prerequisite 23
- Build Containers 23
- Deploy Service 23
- Invoke Service 25
- Auto Scaling 28
- Minimum Scale 30
- Cleanup 32

FAQs 33

- How to access the Knative services ? 33
- Why `dev.local` suffixes for container images? 33

What is revision simpler terms? . . . . . 35

# Overview

Serverless epitomize the very benefits of what cloud platforms promise: offload the management of infrastructure while taking advantage of a consumption model for the actual utilization of services. While there are a number of server frameworks out there, Knative is the first serverless platform specifically designed for Kubernetes and OpenShift.

> Serverless computing refers to the concept of building and running applications that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment"

***Source***: https://www.cncf.io/blog/2018/02/14/cncf-takes-first-step-towards-serverless-computing/

This tutorial will act as step-by-step guide in helping you to understand Knative starting with setup, understanding fundamentals concepts such as service, configuration, revision etc., and finally deploying some use cases which could help deploying serverless applications at enterprises.

This content is brought to you by Red Hat Developer Program - Register today!

# Setup

## Prerequisite CLI tools

You will need in this tutorial:

| Tool | macOS | Fedora |
|---|---|---|
| `minishift` | https://github.com/minishift/minishift/releases | https://github.com/minishift/minishift/releases |
| docker | Docker for Mac | `dnf install docker` |
| kubectl | Mac OS | `dnf install kubernetes-client` |
| Apache Maven | `brew install maven` | `dnf install maven` |
| stern | `brew install stern` | `sudo curl --output /usr/local/bin/stern -L https://github.com/wercker/stern/releases/download/1.6.0/stern_linux_amd64 && sudo chmod +x /usr/local/bin/stern` |
| `curl`, `gunzip`, `tar` | built-in or part of your bash shell | should also be installed already, but just in case… `dnf install curl gzip tar` |
| git | `brew install git` | `dnf install git` |
| xmlstarlet | `brew install xmlstarlet` | http://xmlstar.sourceforge.net/ |
| yq | `brew install yq` | https://github.com/mikefarah/yq/releases/latest |
| httpie | `brew install httpie` | `dnf install httpie` |
| go | `brew install golang` | https://dl.google.com/go/go1.11.5.linux-amd64.tar.gz |
| hey | `go get -u github.com/rakyll/hey` | `go get -u github.com/rakyll/hey` |

## Download Tutorial Sources

Before we start to setting up the environment, lets clone the tutorial sources and call the cloned folder as `$TUTORIAL_HOME`

```
git clone https://github.com/kameshsampath/knative-tutorial
```

All examples in this tutorial are tested with **OpenShift 3.11** and **Knative 0.3.0**

## Minishift

All the demos in this tutorial will be run using minishift, the single node OKD(the Origin Community Distribution of Kubernetes that powers Red Hat OpenShift) cluster that can be used for development purposes.

### Setup minishift

```
export TIMEZONE=$(sudo systemsetup -gettimezone | awk '{print $3}') ①
minishift profile set knative-tutorial
minishift config set openshift-version v3.11.0
minishift config set memory 8GB # if you more then preferred is 10GB
minishift config set cpus 4
minishift config set disk-size 50g
minishift config set image-caching true
minishift addons enable admin-user

minishift start --timezone $TIMEZONE

eval $(minishift docker-env) && eval $(minishift oc-env)
```

① Some examples we use in this tutorial are time sensitive, hence its recommended to set the appropriate timezone. If you are linux please use the appropriate command to get the timezone of the host.

## Enable Admission Controller Webhook

```
#!/bin/bash

minishift openshift config set --target=kube --patch '{
  "admissionConfig": {
    "pluginConfig": {
      "ValidatingAdmissionWebhook": {
        "configuration": {
          "apiVersion": "apiserver.config.k8s.io/v1alpha1",
          "kind": "WebhookAdmission",
          "kubeConfigFile": "/dev/null"
        }
      },
      "MutatingAdmissionWebhook": {
        "configuration": {
          "apiVersion": "apiserver.config.k8s.io/v1alpha1",
          "kind": "WebhookAdmission",
          "kubeConfigFile": "/dev/null"
        }
      }
    }
  }
}'

# Allowing few minutes for the OpenShift to be restarted
until oc login -u admin -p admin 2>/dev/null; do sleep 5; done;
```

# Knative Setup

## Login as admin user

Since there was an `admin` addon added during minishift configuration, its now possible to login with the `admin` user. For example:

```
oc login -u admin -p admin
```

## Install Istio

### Configure Istio Policies

Knative depends on Istio, to deploy Istio you need to run the following commands to configure necessary privileges to the service accounts used by Istio.

```
oc adm policy add-scc-to-user anyuid -z istio-ingress-service-account -n istio-system
oc adm policy add-scc-to-user anyuid -z default -n istio-system
oc adm policy add-scc-to-user anyuid -z prometheus -n istio-system
oc adm policy add-scc-to-user anyuid -z istio-egressgateway-service-account -n istio-
system
oc adm policy add-scc-to-user anyuid -z istio-citadel-service-account -n istio-system
oc adm policy add-scc-to-user anyuid -z istio-ingressgateway-service-account -n istio-
system
oc adm policy add-scc-to-user anyuid -z istio-cleanup-old-ca-service-account -n istio-
system
oc adm policy add-scc-to-user anyuid -z istio-mixer-post-install-account -n istio-
system
oc adm policy add-scc-to-user anyuid -z istio-mixer-service-account -n istio-system
oc adm policy add-scc-to-user anyuid -z istio-pilot-service-account -n istio-system
oc adm policy add-scc-to-user anyuid -z istio-sidecar-injector-service-account -n
istio-system
oc adm policy add-scc-to-user anyuid -z cluster-local-gateway-service-account -n
istio-system
oc adm policy add-scc-to-user anyuid -z istio-galley-service-account -n istio-system
```

### Deploy Istio Components

```
curl -L https://github.com/knative/serving/releases/download/v0.3.0/istio.yaml \
  | sed 's/LoadBalancer/NodePort/' \
  | kubectl apply --filename -
```

The above command throws some warnings and error as to some objects not found, it can safely ignore or run the command again

You can moinitor the Istio components via the command:

```
kubectl get pods -n istio-system -w
```

It will take a few minutes for all the Istio components to be up and running. Please wait for all the Istio pods to be running before deploying Knative Serving. Use `CTRL+C` to exit watch mode.

#### Update Istio sidecar injector ConfigMap

The Istio v1.0.1 release automatic sidecar injection has removed `privileged:true` from init contianers,this will cause the Pods with istio proxies automatic inject to crash. Run the following command to update the **istio-sidecar-injector** ConfigMap.

The following command ensures that the `privileged:true` is added to the **istio-sidecar-injector** ConfigMap:

```
kubectl apply -n istio-system -f $TUTORIAL_HOME/patches/istio-sidecar-injector.yaml
```

Run the above command only once per minishift instance

### Install Knative Build

```
# Setup Knative Build Policies
oc adm policy add-scc-to-user anyuid -z build-controller -n knative-build

# Install Knative Build components
kubectl apply --filename
https://github.com/knative/build/releases/download/v0.3.0/release.yaml

# give cluster admin privileges to Service Account Build Controller on project
knative-build
oc adm policy add-cluster-role-to-user cluster-admin -z build-controller -n knative-
build
```

```
oc get pods -n knative-build -w
```

It will take a few minutes for all the Knative Build components to be up and running. Use `CTRL+C` to exit watch mode.

### Install Knative Serving

```
# Setup Knative Serving Policies
oc adm policy add-scc-to-user anyuid -z controller -n knative-serving
oc adm policy add-scc-to-user anyuid -z autoscaler -n knative-serving

# Install Knative Serving components
curl -L https://github.com/knative/serving/releases/download/v0.3.0/serving.yaml \
  | sed 's/LoadBalancer/NodePort/' \
  | kubectl apply --filename -

# give cluster admin privileges to Service Account Controller on project knative-
serving
oc adm policy add-cluster-role-to-user cluster-admin -z controller -n knative-serving
```

You can monitor the Knative Serving components via components via the command:

```
oc get pods -n knative-serving -w
```

It will take a few minutes for all the Knative Serving components to be up and running. Use `CTRL`+`C` to exit watch mode.

## Install Knative Eventing

```
# Setup Knative Eventing Policies
oc adm policy add-scc-to-user anyuid -z eventing-controller -n knative-eventing
oc adm policy add-scc-to-user anyuid -z in-memory-channel-dispatcher -n knative-
eventing
oc adm policy add-scc-to-user anyuid -z in-memory-channel-controller -n knative-
eventing

# Install Knative Eventing components
kubectl apply --filename
https://github.com/knative/eventing/releases/download/v0.3.0/release.yaml
kubectl apply --filename https://github.com/knative/eventing-
sources/releases/download/v0.3.0/release.yaml

# give cluster admin privileges to Service Accounts on project knative-eventing
oc adm policy add-cluster-role-to-user cluster-admin -z eventing-controller -n
knative-eventing
oc adm policy add-cluster-role-to-user cluster-admin -z default -n knative-sources
oc adm policy add-cluster-role-to-user cluster-admin -z in-memory-channel-dispatcher
-n knative-eventing
oc adm policy add-cluster-role-to-user cluster-admin -z in-memory-channel-controller
-n knative-eventing
```

You can monitor the Knative Eventing components via components via the command:

```
oc get pods -n knative-eventing -w
```

It will take a few minutes for all the Knative Eventing components to be up and running. Use `CTRL+C` to exit watch mode.

## Configuring OpenShift project for Knative applications

```
oc new-project knativetutorial
oc adm policy add-scc-to-user privileged -z default ①
oc adm policy add-scc-to-user anyuid -z default
```

① The `oc adm policy` adds the **privileged** Security Context Constraints(SCCs)to the **default** Service Account. The SCCs are the precursor to the PSP (Pod Security Policy) mechanism in Kubernetes.

# Basics and Fundamentals

At the end of this chapter you will be able to understand and know how to :

- Deploy first Knative service ?
- Deploying multiple revisions of the service
- What different service deployment strategies possible (service vs configurations/routes) ?
- Always running latest vs pinning revision to services

## Prerequisite

Ensure all CLI tools are in path and accessible

```
# to make sure we are connected to minishift docker daemon and using right openshift
client
eval $(minishift docker-env) && eval $(minishift oc-env)
# right kubernetes and openshift versions
# oc v3.11.0+0cbc58b kubernetes v1.11.0+d4cacc0
oc version
# right maven version Apache Maven 3.5.4
./mvnw --version
```

## Build Containers

```
cd $TUTORIAL_HOME/01-basics/java
./mvnw clean package
docker build -t dev.local/rhdevelopers/greeter:0.0.1 .

# check if images are built and available
# dev.local/rhdevelopers/greeter   0.0.1              6aa8771da8dd        2 hours ago
456MB
docker images | grep dev.local
```

You can also use the script `buildImage.sh` that will build source and create the container image

## Deploy Service

Navigate to folder `$TUTORIAL_HOME/01-basics/knative`.

The following snippet shows how a Knative service YAML will look like:

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: greeter
spec:
  runLatest: ①
    configuration:
      revisionTemplate:
        spec:
          container:
            image: dev.local/rhdevelopers/greeter:0.0.1 ②
```

**service.yaml**

① Makes Knative to always run the latest revision of the deployment

② It is very important that the image is a fully qualified name docker image name with tag. For more details on this Question 2 of FAQ

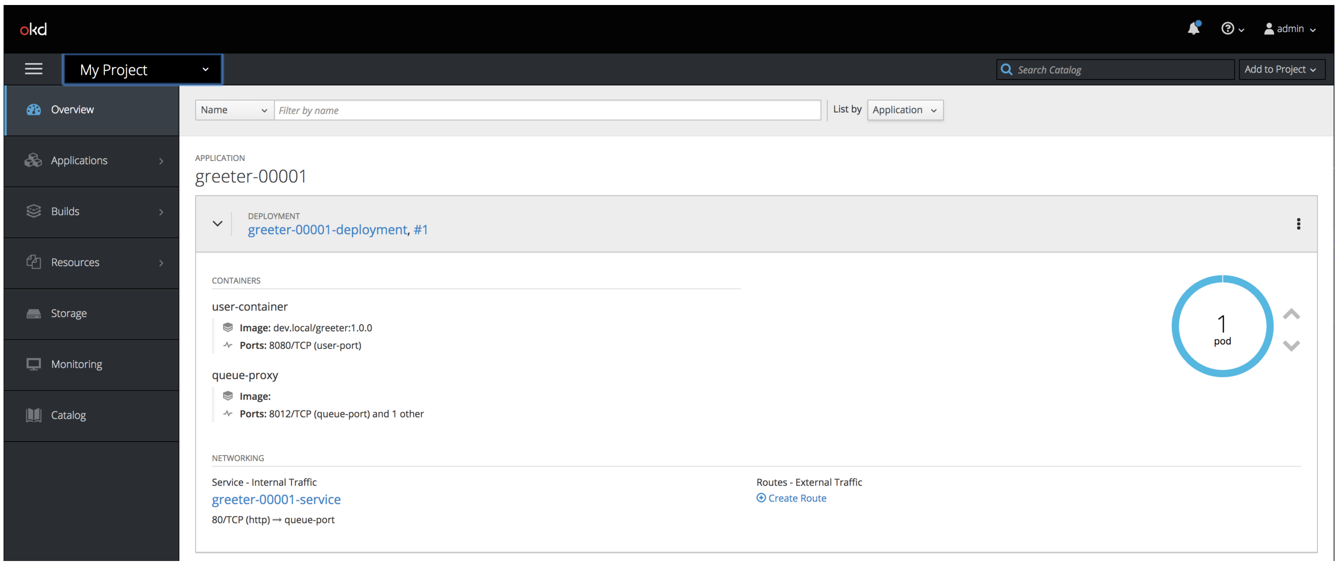
The service could be deployed using the command:

```
oc apply -n knativetutorial -f service.yaml
```

**(OR)**

```
oc apply -n knativetutorial -f service.yaml
```

After successful deployment of the service we should see a Kubernetes deployment called `greeter-00001-deployment` available in the OpenShift dashboard:



APPLICATION
greeter-00001

DEPLOYMENT
greeter-00001-deployment, #1

CONTAINERS

user-container
Image: dev.local/greeter:1.0.0
Ports: 8080/TCP (user-port)

queue-proxy
Image:
Ports: 8012/TCP (queue-port) and 1 other

1 pod

NETWORKING

Service - Internal Traffic
greeter-00001-service
80/TCP (http) → queue-port

Routes - External Traffic
Create Route

## Invoke Service

```
INGRESSGATEWAY=istio-ingressgateway
IP_ADDRESS="$(minishift ip):$(kubectl get svc $INGRESSGATEWAY --namespace istio-system
--output 'jsonpath={.spec.ports[?(@.port==80)].nodePort}')"
```

```
curl -H "Host: greeter.knativetutorial.example.com" $IP_ADDRESS
```

**(OR)**

```
http $IP_ADDRESS 'Host: greeter.knativetutorial.example.com'
```

The last curl command should return a response like **Hi greeter ⇒ greeter-00001-deployment-5d696cc6c8-m65s5: 1**

Sometimes the response might not be returned immediately especially when the pod is coming up from dormant state, at those times try giving request again

## See what you have deployed

The service based deployment strategy that we did now will create many knative resources, the following commands will help you to query and find what has been deployed.

### service

```
# get a Knative Service (short name ksvc) called greeter
oc -n knativetutorial  get services.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial  get services.serving.knative.dev greeter
```

### configuration

```
# get a Knative configuration called greeter
oc -n knativetutorial get configurations.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial get configurations.serving.knative.dev greeter
```

### routes

```
# get a Knative routes called greeter
oc -n knativetutorial get routes.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial get routes.serving.knative.dev greeter
```

When the service was invoked with `curl -H "Host: greeter.knativetutorial.example.com" $IP_ADDRESS`,you noticed that we added a **Host** header to the request with value `greeter.knativetutorial.example.com`,this FQDN is automatically assigned to your Knative service by the Knative Routes,it uses the format like **`<service-name>.<namespace>.<domain-suffix>`**.

- The domain suffix in this case *example.com* is configurable via the config map **config-domain** of **knative-serving** namespace.

### revisions

```
# get a Knative revisions called greeter you will see only one like greeter-00001
oc -n knativetutorial get revisions.serving.knative.dev
```

**(OR)**

```
kubectl -n knativetutorial get revisions.serving.knative.dev
```

- add `-oyaml` to the commands above to see more details

## Deploy new Revision Service

As Knative follows 12-Factor application principles, any new configuration change will trigger new revision of the deployment.

To deploy a new revision of the greeter service, we will add an environment variable to the existing service as shown below:

### Service revision 2

```yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: greeter
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: dev.local/rhdevelopers/greeter:0.0.1
            env:
             - name: MESSAGE_PREFIX ①
               value: Hello
```

**service-env.yaml**

① Adding an environment variable that will be used a message prefix

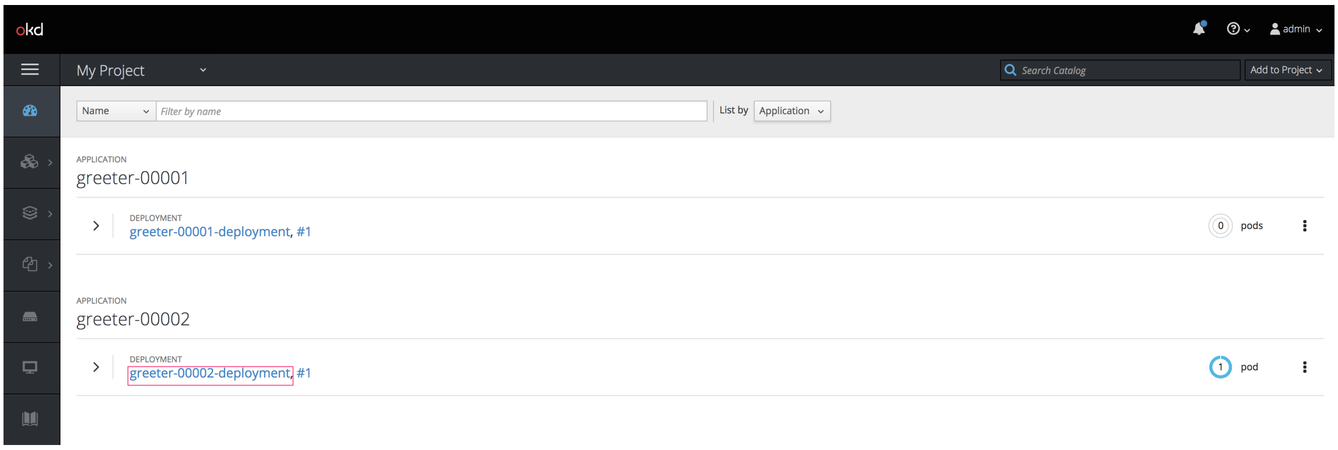
Let us deploy the new revision using the command:

```
oc apply -n knativetutorial -f service-env.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f service-env.yaml
```

After successful deployment of the service we should see a Kubernetes deployment called `greeter-00002-deployment` available in the OpenShift dashboard:



Now running the command will show two revisions namely `greeter-00001` and `greeter-0002`.

Invoking Service will now show an output like **Hello greeter ⇒ greeter-00002-deployment-8d9984dc8-rgzx6: 2**, where *Hello* is the value that we configured via environment variable in the Knative service resource file.

## Pinning service to a revision

As you noticed that the Knative services always routes the traffic to the **latest** revision of the service deployment, thats because of the **runLatest** attribute in service resource file.

Let us now make the greeter service use earlier revision `greeter-00001`.

You can use the get [show-knative-revisions] command to find the available revisions for the greeter service.

### Service pinned to revision 1

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: greeter
spec:
  pinned: ①
    revisionName: greeter-00001
  configuration:
    revisionTemplate:
      spec:
        container:
          image: dev.local/rhdevelopers/greeter:0.0.1
```

**service-pinned-rev1.yaml**

① The **pinned** attribute in service resource file will make Knative use the revision specified in the **revisionName** attribute.

Let redeploy the greeter service to be pinned to revision *greeter-00001*:

```
cd $TUTORIAL_HOME/knative
```

```
oc apply -n knativetutorial -f service-pinned-rev1.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f service-pinned-rev1.yaml
```

Invoking Service will now show an output like **Hi greeter ⇒ greeter-00001-deployment-5d696cc6c8-m65s5: 3**.

## Cleanup

```
oc -n knativetutorial delete services.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial delete services.serving.knative.dev greeter
```

# Configurations, Routes and Traffic distribution

At the end of this chapter you will be able to understand and know how to :

- Deploying Knative configurations and routes separately
- Distributing traffic between revisions of the service

## Prerequisite

Ensure all CLI tools are in path and accessible

```
# to make sure we are connected to minishift docker daemon and using right openshift
client
eval $(minishift docker-env) && eval $(minishift oc-env)
# right kubernetes and openshift versions
# oc v3.11.0+0cbc58b kubernetes v1.11.0+d4cacc0
oc version
# right maven version Apache Maven 3.5.4
./mvnw --version
```

## Build

```
cd $TUTORIAL_HOME/01-basics/java
./mvnw clean package
docker build -t dev.local/rhdevelopers/greeter:0.0.1 .

# check if images are built and available
# dev.local/rhdevelopers/greeter   0.0.1               6aa8771da8dd        2 hours ago
456MB
docker images | grep dev.local
```

You can also use the script `buildImage.sh` that will build source and create the container image

## Deploy Configuration

In previous chapter we saw how we can deploy service using holistic service resource file, in this chapter we will see how to deploy the service using configurations and resource files.

Navigate to folder `$TUTORIAL_HOME/02-configs-and-routes`:

## Deploy Configuration revision 1

The following snippet how a Knative configuration resource YAML will look like:

```yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Configuration
metadata:
  name: greeter
spec:
  revisionTemplate:
    metadata:
      labels:
        app: greeter
    spec:
      container:
        image: dev.local/rhdevelopers/greeter:0.0.1 ①
```

**configuration-rev1.yaml**

① It is very important that the image is a fully qualified name docker image name with tag. For more details on this Question 2 of FAQ

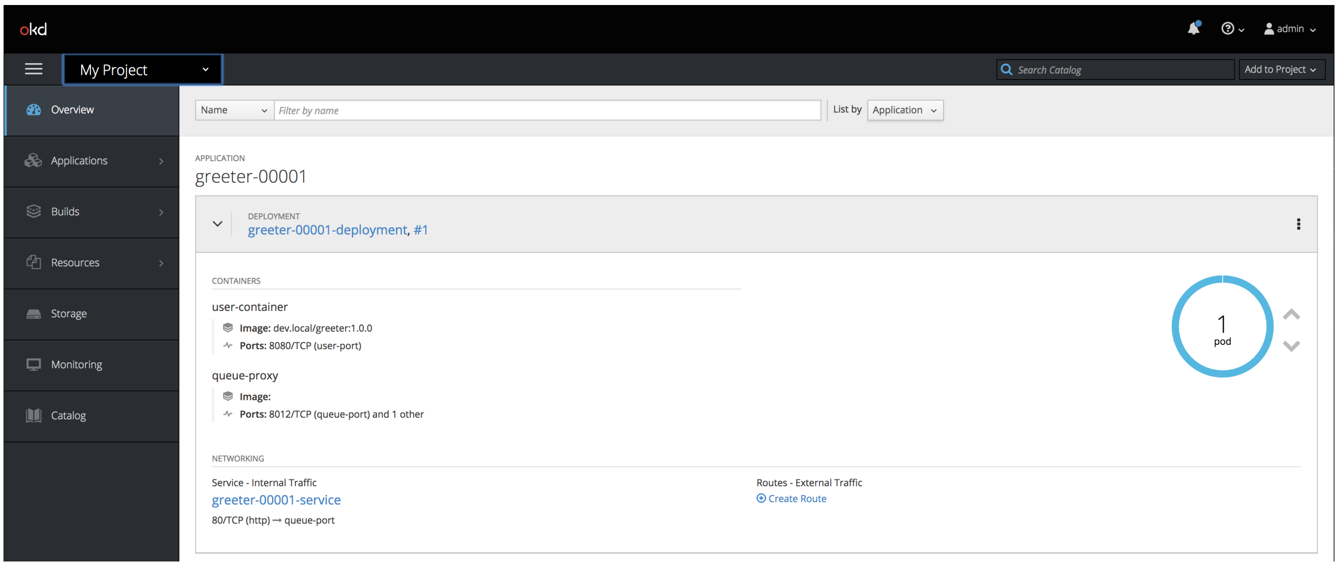
The service could be deployed using the command:

```
oc apply -n knativetutorial -f config/configuration-rev1.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f config/configuration-rev1.yaml
```

After successful deployment of the service we should see a Kubernetes deployment called `greeter-00001-deployment` available in the OpenShift dashboard:



APPLICATION
greeter-00001

DEPLOYMENT
greeter-00001-deployment, #1

CONTAINERS

user-container
**Image:** dev.local/greeter:1.0.0
**Ports:** 8080/TCP (user-port)

queue-proxy
**Image:**
**Ports:** 8012/TCP (queue-port) and 1 other

1 pod

NETWORKING

Service - Internal Traffic
greeter-00001-service
80/TCP (http) → queue-port

Routes - External Traffic
Create Route

## Invoke Service

```
INGRESSGATEWAY=istio-ingressgateway
IP_ADDRESS="$(minishift ip):$(kubectl get svc $INGRESSGATEWAY --namespace istio-system
--output 'jsonpath={.spec.ports[?(@.port==80)].nodePort}')"
```

```
curl -H "Host: greeter.knativetutorial.example.com" $IP_ADDRESS
```

**(OR)**

```
http $IP_ADDRESS 'Host: greeter.knativetutorial.example.com'
```

The command will return HTTP 404 as there are no routes deployed yet. Let us now deploy a route

## Deploy Route

Let us now deploy a default route that will route all the traffic to service deployed via configuration `greeter`

```yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: greeter
spec:
  traffic:
    - configurationName: greeter
      percent: 100
```

**route_default.yaml**

```
oc apply -n knativetutorial -f route/route_default.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f route/route_default.yaml
```

Invoking Service now should return a response like **Hi greeter ⇒ greeter-00001-deployment-5d696cc6c8-m65s5: 1**

Sometimes the response might not be returned immediately especially when the pod is coming up from dormant state, at those times try giving request again

## See what you have deployed

The holistic service base deployment strategy that we did now will create the following resource objects:

### service

```
# get a Knative Service (short name ksvc) called greeter
oc -n knativetutorial  get services.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial  get services.serving.knative.dev greeter
```

The command will throw an error as there are no knative services deployed as part of configuration+routes based deployment strategy.

### configuration

```
# get a Knative configuration called greeter
oc -n knativetutorial get configurations.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial get configurations.serving.knative.dev greeter
```

### routes

```
# get a Knative routes called greeter
oc -n knativetutorial get routes.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial get routes.serving.knative.dev greeter
```

When the service was invoked with `curl -H "Host: greeter.knativetutorial.example.com" $IP_ADDRESS`, you noticed that we added a **Host** header to the curl with value `greeter.knativetutorial.example.com`, this FQDN is automatically assigned to your Knative service by the Knative Routes. It follows a format `<service-name>.<namespace>.<domain-suffix>`.

- The domain suffix in this case *example.com* is configurable via the config map **config-domain** of **knative-serving** namespace.

### revisions

```
# get a Knative revisions called greeter you will see only one like greeter-00001
oc -n knativetutorial get revisions.serving.knative.dev
```

**(OR)**

```
kubectl -n knativetutorial get revisions.serving.knative.dev
```

- add `-oyaml` to the commands above to see more details

## Deploy new Revision Service

For the new revision of the service we will add an environment variable to the service deployment. As Knative follows 12-Factor application principles, any new configuration change will trigger new deployment, in this case it is the new environment variable.

### Deploy configuration revision 2

```
apiVersion: serving.knative.dev/v1alpha1
kind: Configuration
metadata:
  name: greeter
spec:
  revisionTemplate:
    metadata:
      labels:
        app: greeter
    spec:
      container:
        image: dev.local/rhdevelopers/greeter:0.0.1
        env: ①
          - name: MESSAGE_PREFIX
            value: Hello
```

**configuration-rev2.yaml**

① Adding an environment variable that will be used a message prefix

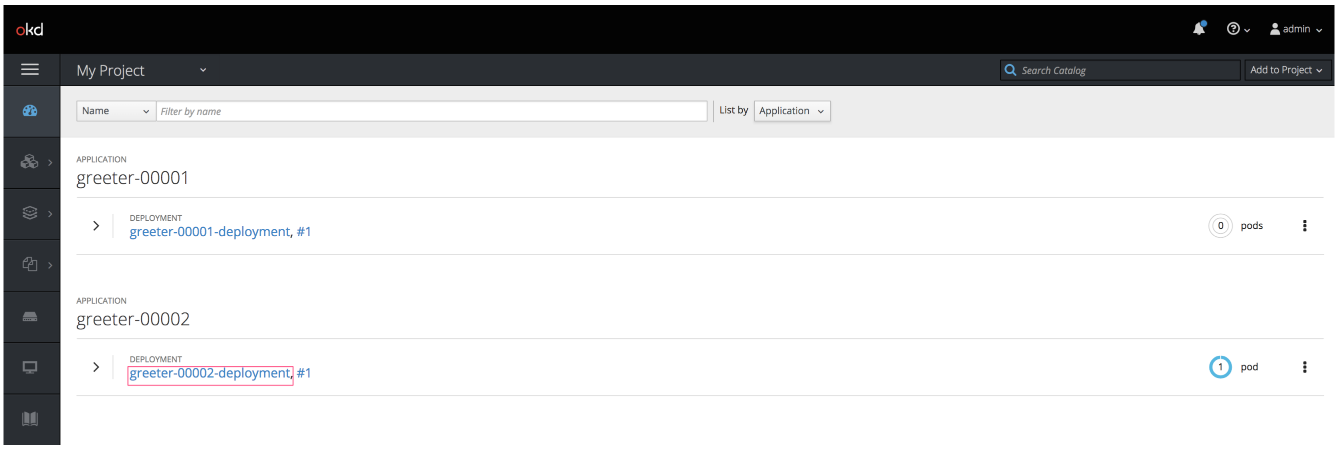
Let us deploy the new revision using the command:

```
oc apply -n knativetutorial -f config/configuration-rev2.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f config/configuration-rev2.yaml
```

After successful deployment of the service we should see a Kubernetes deployment called `greeter-00002-deployment` available in the OpenShift dashboard:



Now running the command will show two revisions namely `greeter-00001` and `greeter-0002`.

Invoking Service will now show an output like **Hello greeter ⇒ greeter-00002-deployment-8d9984dc8-rgzx6: 2**, where *Hello* is the value that we configured via environment variable in the Knative service resource file.

## Distributing traffic

When deploying services with service object based approach, the service deployment will take care of traffic distribution either 100% to latest or pinned revision. In this section we will explore how we can deploy routes to distribute traffic between deployed revisions.

For the sake of clarity we will call greeter-00001 as **revision 1** and greeter-00002 as **revision 2**

Before we start to apply routes, let us open a new terminal and run the following command `$TUTORIAL_HOME/02-configs-and-routes/bin/call.sh`, this command will keep sending requests to greeter service in every two seconds which allows us to monitor the changes to responses as we keep applying the route that will distribute the traffic between available two revisions.

### All traffic to revision 1

```
oc apply -n knativetutorial -f route/route_all_rev1.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f route/route_all_rev1.yaml
```

You will notice the output on your other terminal to be something like **Hi greeter ⇒ greeter-00001-**

**deployment-5d696cc6c8-m65s5: 34**.

### All traffic to revision 2

```
oc apply -n knativetutorial -f route/route_all_rev2.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f route/route_all_rev2.yaml
```

You will notice the output on your other terminal to be something like **Hello greeter ⇒ greeter-00002-deployment-8d9984dc8-rgzx6: 13**.

### 50-50 between revision 1 and revision 2

```
oc apply -n knativetutorial -f route/route_rev1-50_rev2-50.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f route/route_rev1-50_rev2-50.yaml
```

You will notice the output will be mix of responses like **Hi greeter ⇒ greeter-00001-deployment-5d696cc6c8-m65s5: 11** and **Hello greeter ⇒ greeter-00002-deployment-8d9984dc8-rgzx6: 10** approximately distributed 50% between each.

### 75-25 between revision 1 and revision 2

```
oc apply -n knativetutorial -f route/route_rev1-75_rev2-25.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f route/route_rev1-75_rev2-25.yaml
```

You will notice the output will be mix of responses like **Hi greeter ⇒ greeter-00001-deployment-5d696cc6c8-m65s5: 6** and **Hello greeter ⇒ greeter-00002-deployment-8d9984dc8-rgzx6:** 7, with more requests responded by revision 1 approximately distributed 75% to 25 % between revision 1 and revision 2.

### 10-90 between revision 1 and revision 2

```
oc apply -n knativetutorial -f route/route_rev1-10_rev2-90.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f route/route_rev1-10_rev2-90.yaml
```

You will notice the will be mix of responses like **Hi greeter ⇒ greeter-00001-deployment-5d696cc6c8-m65s5: 4** and **Hello greeter ⇒ greeter-00002-deployment-8d9984dc8-rgzx6: 5**, with more requests responded by revision 2 approximately 10% to 90% between revision 1 and revision 2.

In response texts e.g **Hi greeter ⇒ greeter-00001-deployment-5d696cc6c8-m65s5: 4** the numbers at the end of the responses shows a count of now may requests has been handled by the service in this example it is *4*. Also note that the output count number may vary according to number of requests that you might have issued.

## Cleanup

```
oc -n knativetutorial delete configurations.serving.knative.dev greeter
oc -n knativetutorial delete routes.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial delete configurations.serving.knative.dev greeter
kubectl -n knativetutorial delete routes.serving.knative.dev greeter
```

# Scaling

At the end of this chapter you will be able to:

- Configure the scale to zero time period
- Configure auto scaling
- Understanding types of autoscaling strategies
- How to enable concurrency based auto scaling based concurrency
- How to prevent minimum number of pods for a service i.e. no scale down to zero beyond this number of pods

## Prerequisite

Ensure all CLI tools are in path and accessible

```
# to make sure we are connected to minishift docker daemon and using right openshift
client
eval $(minishift docker-env) && eval $(minishift oc-env)
# right kubernetes and openshift versions
# oc v3.11.0+0cbc58b kubernetes v1.11.0+d4cacc0
oc version
# right maven version Apache Maven 3.5.4
./mvnw --version
```

## Build Containers

```
cd $TUTORIAL_HOME/01-basics/java
./mvnw clean package
docker build -t dev.local/rhdevelopers/greeter:0.0.1 .

# check if images are built and available
# dev.local/rhdevelopers/greeter   0.0.1               6aa8771da8dd        2 hours ago
456MB
docker images | grep dev.local
```

You can also use the script `buildImage.sh` that will build source and create the container image

## Deploy Service

The following snippet shows how a Knative service YAML will look like:

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: greeter
spec:
  runLatest: ①
    configuration:
      revisionTemplate:
        spec:
          container:
            image: dev.local/rhdevelopers/greeter:0.0.1 ②
```

**service.yaml**

① Makes Knative to always run the latest revision of the deployment

② It is very important that the image is a fully qualified name docker image name with tag. For more details on this Question 2 of FAQ

The service could be deployed using the command:

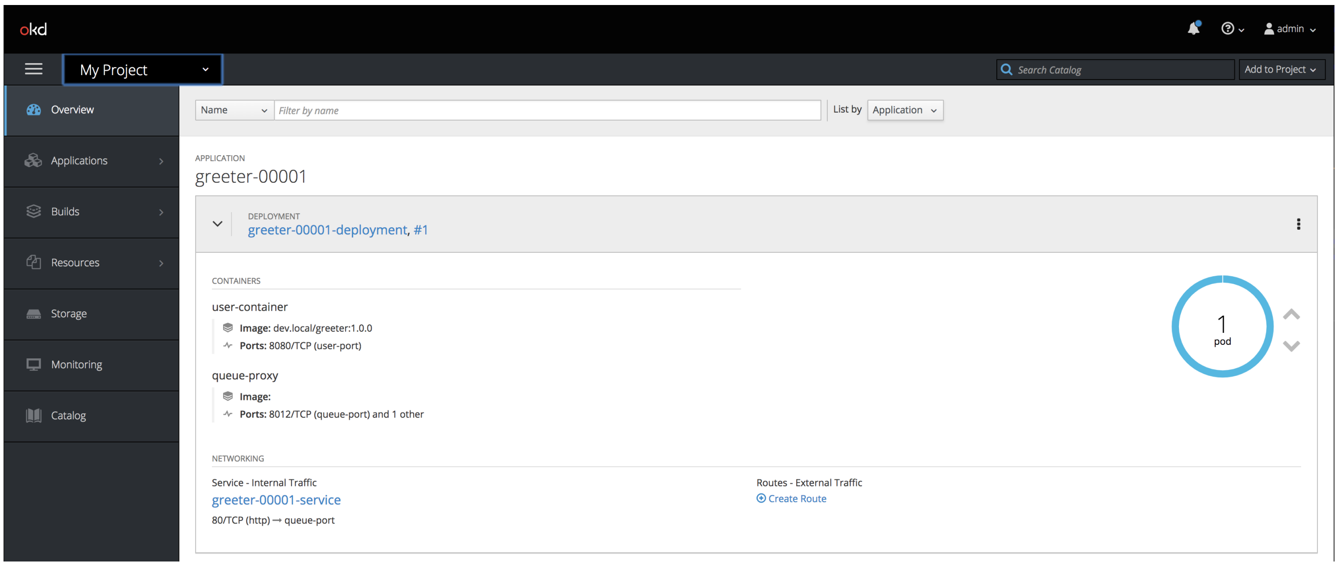
```
cd $TUTORIAL_HOME/01-basics/knative
```

```
oc apply -n knativetutorial -f service.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f service.yaml
```

After successful deployment of the service we should see a Kubernetes deployment called `greeter-00001-deployment` available in the OpenShift dashboard:



APPLICATION
greeter-00001

DEPLOYMENT
greeter-00001-deployment, #1

CONTAINERS

user-container
Image: dev.local/greeter:1.0.0
Ports: 8080/TCP (user-port)

queue-proxy
Image:
Ports: 8012/TCP (queue-port) and 1 other

1 pod

NETWORKING

Service - Internal Traffic
greeter-00001-service
80/TCP (http) → queue-port

Routes - External Traffic
Create Route

## Invoke Service

```
INGRESSGATEWAY=istio-ingressgateway
IP_ADDRESS="$(minishift ip):$(kubectl get svc $INGRESSGATEWAY --namespace istio-system
--output 'jsonpath={.spec.ports[?(@.port==80)].nodePort}')"
```

```
curl -H "Host: greeter.knativetutorial.example.com" $IP_ADDRESS
```

**(OR)**

```
http $IP_ADDRESS 'Host: greeter.knativetutorial.example.com'
```

The last curl command should return a response like **Hi greeter ⇒ greeter-00001-deployment-5d696cc6c8-m65s5: 1**

Check deployed Knative resources tfor more details of what Knative objects and resources have been created with the above service deployment.

## Configure scale to zero

Assuming that Greeter service has been deployed, if the service has been terminated, the let us invoke the service to make service available for request.

### Calculating scale to zero time period

Knative autoscaler uses the config map **config-autoscaler** of **knative-serving** namespace for autoscaling related properties. The value of the attribute `stable-window` and `scale-to-zero-grace-period` decides how long Knative autoscaler will wait before terminating the inactive revision pod.

```
stableWindow=`oc -n knative-serving get configmaps config-autoscaler -o yaml |
 yq r - data.stable-window`
scaleToZeroGracePeriod=`oc -n knative-serving get configmaps config-autoscaler -o yaml
|
 yq r - data.scale-to-zero-grace-period`
```

**(OR)**

```
stableWindow=`kubectl -n knative-serving get configmaps config-autoscaler -o yaml |
 yq r - data.stable-window`
scaleToZeroGracePeriod=`kubectl -n knative-serving get configmaps config-autoscaler -o
yaml |
 yq r - data.scale-to-zero-grace-period`
```

`termination period time in seconds = stableWindow + scaleToZeroGracePeriod`

!

- scale-to-zero-grace-period is dynamic parameter i.e. the value is immediately effected after updating the config map
- the minimum value that can be set for `scale-to-zero-grace-period` is 30 seconds

## Observing default scale down

You can open a new terminal and run the command `watch 'oc get pods -n knativetutorial'` to view the scale up and scale down to zero.

Checking default values

```
# should return 60s
echo $stableWindow
# should return 30s
echo $scaleToZeroGracePeriod
```

By default the **scale-to-zero-grace-period** is `30s`, and the **stable-window** is `60s`, firing the request to the greeter service will bring up the pod (if its already terminated) to serve the request. Leaving it without any further requests, it will automatically scale to zero `1 min 30 secs`(Compute scale to zero grace period)..

## Update scale down to 1 minute

Let us now update **scale-to-zero-grace-period** to `1m` and leave the **stable-window** to default `60s`.

```
cd $TUTORIAL_HOME/03-scaling/knative
```

```
oc -n knative-serving get cm config-autoscaler -o yaml | yq w - -s configure-scaling-to-1m.yaml | oc apply -f -
```

**(OR)**

```
kubectl -n knative-serving get cm config-autoscaler -o yaml | yq w - -s configure-scaling-to-1m.yaml | kubectl apply -f -
```

Verifying the `scale-to-zero-grace-period` value, which should return `1m`:

```
oc -n knative-serving get configmap config-autoscaler -o yaml \
  | yq r - data.scale-to-zero-grace-period
```

**(OR)**

```
kubectl -n knative-serving get configmap config-autoscaler -o yaml \
 | yq r - data.scale-to-zero-grace-period
```

Now firing the request to the greeter service will bring up the pod to serve the request.Leaving it without any further requests, it will automatically scale to zero `2 mins`(Compute scale to zero grace period)

## Update scale down to 2 minute

Let us now update **scale-to-zero-grace-period** to `2m` and leave the **stable-window** to default `60s`.

```
oc -n knative-serving get cm config-autoscaler -o yaml | yq w - -s configure-scaling-
to-2m.yaml | oc apply -f -
```

**(OR)**

```
kubectl -n knative-serving get cm config-autoscaler -o yaml | yq w - -s configure-
scaling-to-2m.yaml | kubectl apply -f -
```

Verifying the `scale-to-zero-grace-period` value, which should return `2m`:

```
# should return 2m
oc -n knative-serving get configmap config-autoscaler -o yaml \
  | yq r - data.scale-to-zero-grace-period
```

**(OR)**

```
kubectl -n knative-serving get configmap config-autoscaler -o yaml \
 | yq r - data.scale-to-zero-grace-period
```

Now firing the request to the greeter service will bring up the pod to serve the request and if we leave the service without any further requests, it will automatically scale to zero in `3 mins`(Compute scale to zero grace period).

## Reset to defaults

Let us revert the `scale-to-zero-grace-period` to defaults:

```
oc -n knative-serving get cm config-autoscaler -o yaml | yq w - -s configure-scaling-
to-30s.yaml | oc apply -f -
```

**(OR)**

```
kubectl -n knative-serving get cm config-autoscaler -o yaml | yq w - -s configure-
scaling-to-30s.yaml | kubectl apply -f -
```

Verifying the `scale-to-zero-grace-period` value, which should return `30s`:

```
# should return 2m
oc -n knative-serving get configmap config-autoscaler -o yaml \
  | yq r - data.scale-to-zero-grace-period
```

**(OR)**

```
kubectl -n knative-serving get configmap config-autoscaler -o yaml \
 | yq r - data.scale-to-zero-grace-period
```

For the sake of clarity and better understanding clean up the deployed knative service before going to next section.

## Auto Scaling

By default Knative Serving allows 100 concurrency pods, you can view the setting `container-concurrency-target-default` in the configmap **config-autoscaler** of **knative-serving** namespace.

For this exercise let us make our service handle only **10** concurrent requests,this will make the Knative-Serving to scale the pods to handle the requests more than 10 requests

### Service with concurrency of 10 requests

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: greeter
spec:
  runLatest:
    configuration:
      revisionTemplate:
        metadata:
          annotations:
            autoscaling.knative.dev/target: "10" ①
        spec:
          container:
            image: dev.local/rhdevelopers/greeter:0.0.1
            livenessProbe:
              httpGet:
                path: /healthz
            readinessProbe:
              httpGet:
                path: /healthz
```

**service-10.yaml**

① Will allow each service pod to handle max of 10 in-flight requests per pod before automatically scaling to new pod(s)

### Deploy service

```
oc apply -n knativetutorial -f service-10.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f service-10.yaml
```

### Invoke Service

```
INGRESSGATEWAY=istio-ingressgateway
IP_ADDRESS="$(minishift ip):$(kubectl get svc $INGRESSGATEWAY --namespace istio-system
--output 'jsonpath={.spec.ports[?(@.port==80)].nodePort}')"
```

Open a new terminal and run the following command:

```
watch oc get pods -n knativetutorial
```

**(OR)**

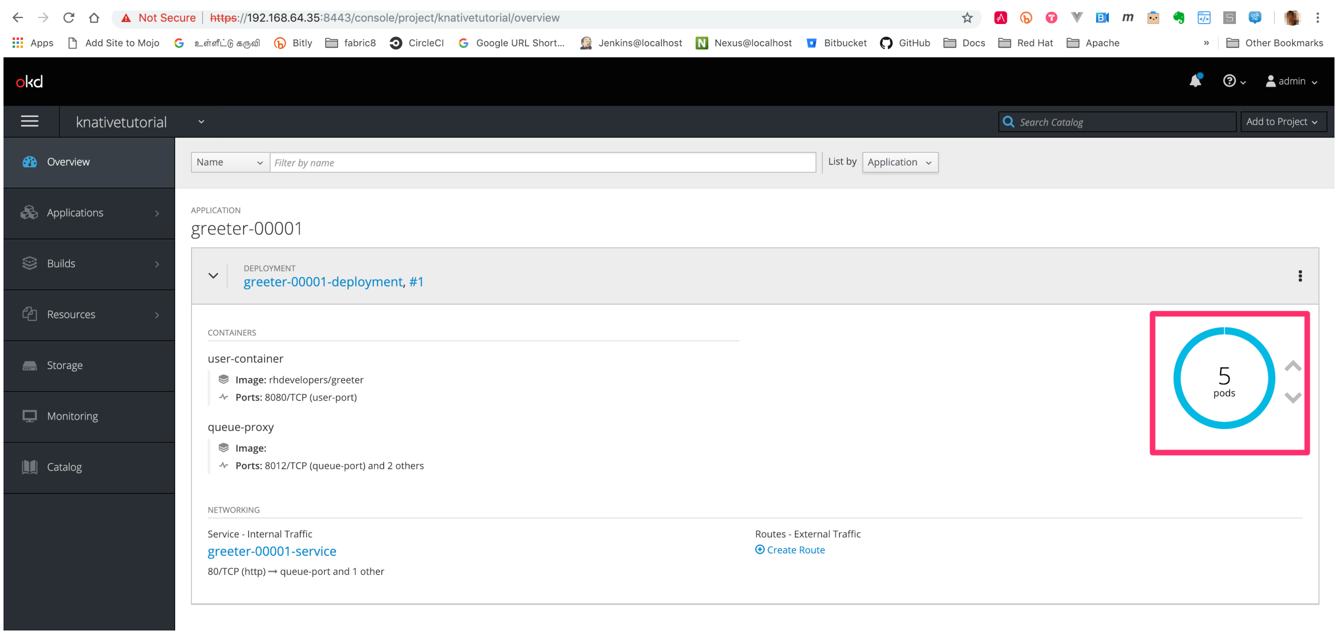
```
watch kubectl get pods -n knativetutorial
```

### Load the service

We will now send some load to the greeter service. The command below sends 50 concurrent requests (`-c 50`) for next 10s (`-z 10s`) with no connection timeout(`-t 0`)

```
hey -z 10s -c 50 -t 0 \
  -host "greeter.knativetutorial.example.com" \
  "http://${IP_ADDRESS}"
```

After successful run the load test, you will notice the number of greeter service pods would have automatically scaled to 5, you can also view the same via OpenShift dashboard as shown below:



APPLICATION
greeter-00001
DEPLOYMENT
greeter-00001-deployment, #1
CONTAINERS
user-container
Image: rhdevelopers/greeter
Ports: 8080/TCP (user-port)
queue-proxy
Image:
Ports: 8012/TCP (queue-port) and 2 others
NETWORKING
Service - Internal Traffic
greeter-00001-service
80/TCP (http) → queue-port and 1 other
Routes - External Traffic

The autoscale pods is computed using the formula:

```
totalPodsToScale = total number of inflight requests / average concurrency target
```

With this current setting of **average concurrency target**=**10**(`autoscaling.knative.dev/target: "10"`) and **total number of inflight requests**=**50** , you will see Knative automatically scales the greeter services to `50/10 = 5 pods`.

For more clarity and understanding let us clean up existing deployments before proceeding to next section.

## Minimum Scale

At times you might need that you always need to have `n` number of pods running e.g. cases where you want to handle sudden spike in requests, with Knative auto scaling we can do that with via `minScale` annotation.

## Deploy service

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: greeter
spec:
  runLatest:
    configuration:
      revisionTemplate:
        metadata:
          annotations:
            autoscaling.knative.dev/target: "10"  ①
            autoscaling.knative.dev/minScale: "2" ②
        spec:
          container:
            image: dev.local/rhdevelopers/greeter:0.0.1
            livenessProbe:
              httpGet:
                path: /healthz
            readinessProbe:
              httpGet:
                path: /healthz
```

**service-min-scale.yaml**

① Will allow each service pod to handle max of 10 in-flight requests per pod before automatically scaling to new pod(s)

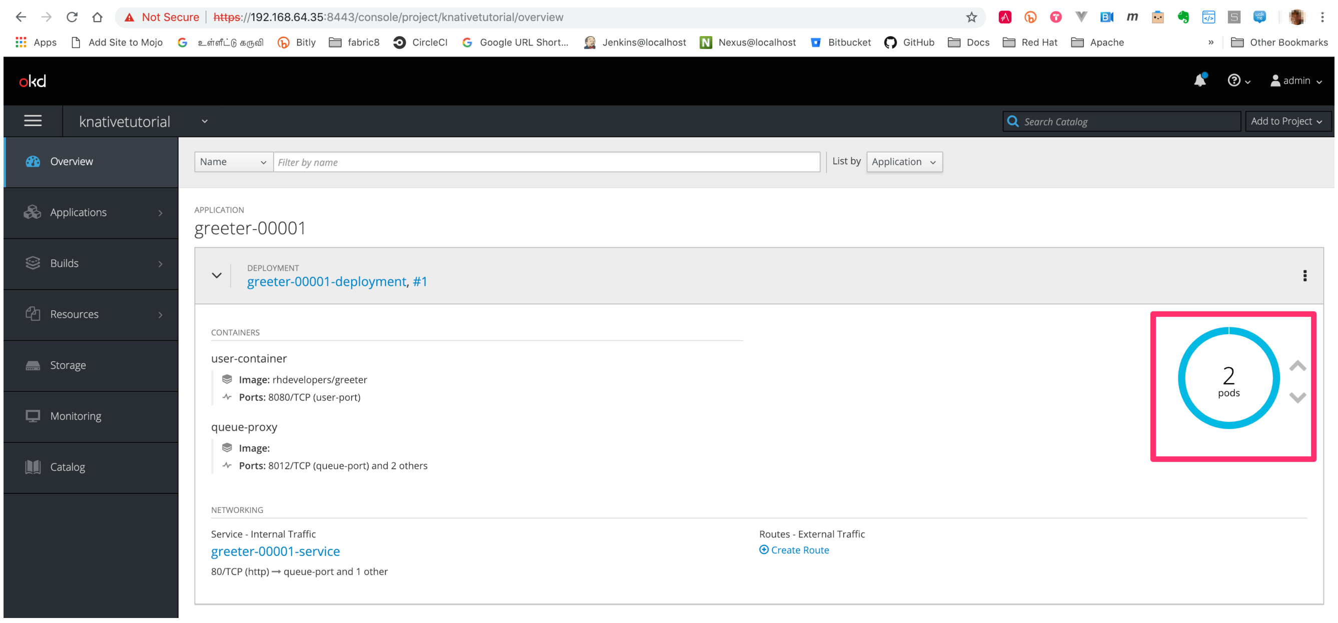
② Every new deployment of this service will minimum have two pods

```
oc apply -n knativetutorial -f service-min-scale.yaml
```

**(OR)**

```
kubectl apply -n knativetutorial -f service-min-scale.yaml
```

After successful deployment of the service we should see a Kubernetes deployment called `greeter-00001-deployment` with **two** pods readily available.



APPLICATION
greeter-00001

DEPLOYMENT
greeter-00001-deployment, #1

CONTAINERS
user-container
Image: rhdevelopers/greeter
Ports: 8080/TCP (user-port)
queue-proxy
Image:
Ports: 8012/TCP (queue-port) and 2 others

2 pods

NETWORKING
Service - Internal Traffic
greeter-00001-service
80/TCP (http) → queue-port and 1 other

Routes - External Traffic
Create Route

**OpenShift Dashboard Knative Tutorial project**

Open a new terminal and run the following command :

```
watch oc get pods -n knativetutorial
```

**(OR)**

```
watch kubectl get pods -n knativetutorial
```

Let us send some load to the service to trigger autoscaling.

When all requests are done and if you are beyond the `scale-to-zero-grace-period`, you will notice that Knative has terminated only 3 out 5 pods, this is because we have configured Knative to always run two pods via the annotation `autoscaling.knative.dev/minScale: "2"`

## Cleanup

```
oc -n knativetutorial delete services.serving.knative.dev greeter
```

**(OR)**

```
kubectl -n knativetutorial delete services.serving.knative.dev greeter
```

# FAQs

## How to access the Knative services ?

When looking up the IP address to use for accessing your app, you need to look up the NodePort for the **istio-ingressgateway** well as the IP address used for minishift. You can use the following command to look up the value to use for the {IP_ADDRESS} placeholder used in the sample

```
#!/bin/bash
# In Knative 0.2.x and prior versions, the `knative-ingressgateway` service was used
instead of `istio-ingressgateway`.
INGRESSGATEWAY=knative-ingressgateway

# The use of `knative-ingressgateway` is deprecated in Knative v0.3.x.
# Use `istio-ingressgateway` instead, since `knative-ingressgateway`
# will be removed in Knative v0.4.
if kubectl get configmap config-istio -n knative-serving &> /dev/null; then
    INGRESSGATEWAY=istio-ingressgateway
fi

INGRESSGATEWAY=istio-ingressgateway
IP_ADDRESS="$(minishift ip):$(kubectl get svc $INGRESSGATEWAY --namespace istio-system
--output 'jsonpath={.spec.ports[?(@.port==80)].nodePort}')"

# calling a knative service named greeter
curl -H "Host: greeter.myproject.example.com" $IP_ADDRESS
```

## Why `dev.local` suffixes for container images?

> Docker images wit v2 or later format has content addressable identifier called digest. The digest remains unchanged as long the underlying image content remains unchanged.

***Source:*** *https://docs.docker.com/engine/reference/commandline/images/#list-image-digests*

Let say you have a deployment like the following (note that resource definition have been trimmed for brevity):

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: helloworld
spec:
  template:
    spec:
       containers:
         - name: my-container
           image: gcr.io/knative-samples/helloworld-go

. . .
```

When you deploy this application in Kubernetes, the deployment will look like:

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: helloworld
spec:
  template:
    spec:
       containers:
          - name: my-container
            image: >-
                   gcr.io/knative-samples/helloworld-
go@sha256:98af362ceca8191277206b3b3854220ce125924b28a1166126296982e33882d0
. . .
```

In the above example the container image name of the deployment i.e. gcr.io/knative-samples/helloworld-go was resolved to its digest gcr.io/knative-samples/helloworld-go@sha256:98af362ceca8191277206b3b3854220ce125924b28a1166126296982e33882d0. Kubernetes extracts the digest from the image manifest. This process of resolving image tag to its digest is informally called as “Tag to Digest”.

Knative Serving deployments by default resolve the container images to digest during the deployment process. Knative Serving has been configured to skip the resolution of image name/tag to image digest for registries ko.local and dev.local, which can be used for local development builds, as the underlying image content are subject to changes during the development process.

You can also add any of the other image repo suffixes from skipping to digest by editing the configmap **config-controller** of **knative-serving** namespace and appending your repo to `registriesSkippingTagResolving` attribute of the configmap.

The following command shows an example how to add a registry **my.docker.registry** to the be skipped:

```
val=$(oc -n knative-serving get cm config-controller -oyaml | yq r -
data.registriesSkippingTagResolving | awk '{print
$1",my.docker.registry"}')

oc -n knative-serving get cm config-controller -oyaml | yq w -
data.registriesSkippingTagResolving $val | oc apply -f -
```

## What is revision simpler terms?

In Knative serving, things are driven via a Configuration to make a separation between code (container images) and config. One of the good practices in configuration management is that we should always be able to rollback the application state to any “last known good configuration”, to be able to allow this type of rollback Knative creates an unique revision for each and every configuration change.