Knative Workshop

Self-link: bit.ly/knativeworkshop

Slides

Author: Ray Tsang (@saturnism), James Ward (@_JamesWard)

Lab 1 - Initial setup

Duration: 20:00

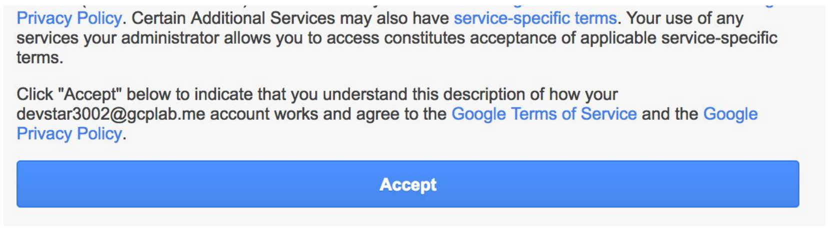
You will get setup on GCP and get Cloud Shell working.

Task 1 - Logging In

1. The instructor will provide you with a temporary username / password to sign in to the Google Cloud Platform Console.

Important: To avoid conflicts with your personal account, please open a new incognito window for the rest of this lab.

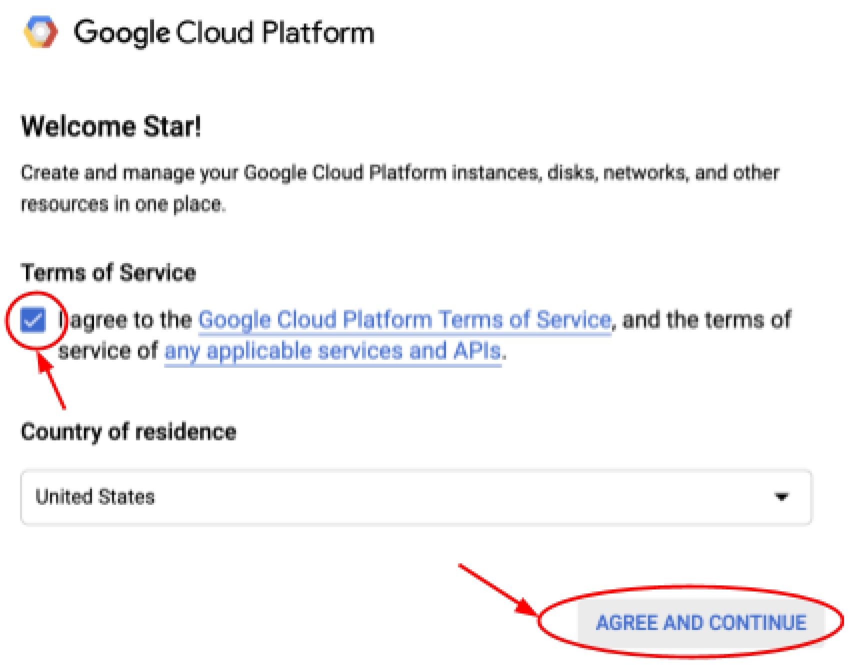
2. Sign in to the Google Cloud Platform Console: https://console.cloud.google.com/ with the provided credentials. In **Welcome to your new account** dialog, click **Accept**.



Privacy Policy. Certain Additional Services may also have service-specific terms. Your use of any services your administrator allows you to access constitutes acceptance of applicable service-specific terms.

Click "Accept" below to indicate that you understand this description of how your devstar3002@gcplab.me account works and agree to the Google Terms of Service and the Google Privacy Policy.

3. Check **I agree** and click **Agree and Continue**


Google Cloud Platform

Welcome Star!

Create and manage your Google Cloud Platform instances, disks, networks, and other resources in one place.

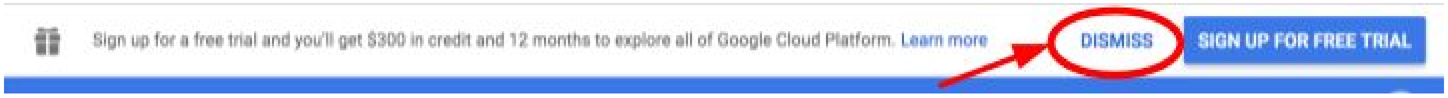
Terms of Service

☑ I agree to the Google Cloud Platform Terms of Service, and the terms of service of any applicable services and APIs.

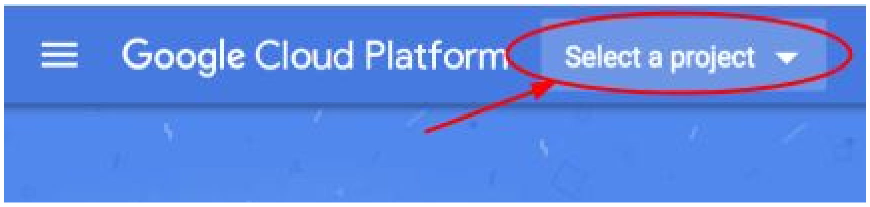
Country of residence

United States

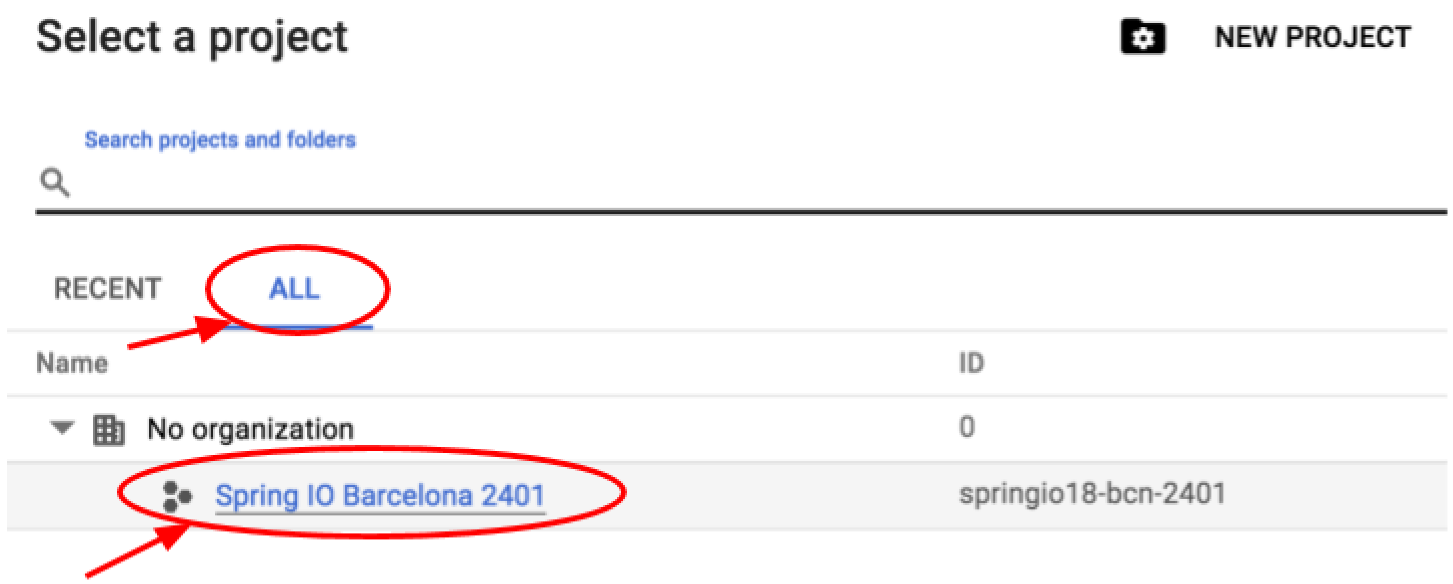
4. If you see a top bar with **Sign Up for Free Trial** - DO NOT SIGN UP FOR THE FREE TRIAL. Click **Dismiss** since you'll be using a pre-provisioned lab account. If you are doing this on your own account, then you may want the free trial.


Sign up for a free trial and you'll get $300 in credit and 12 months to explore all of Google Cloud Platform. Learn more

5. Click **Select a project**.



6. In the **All** tab, click on your project:


Select a project

NEW PROJECT

Search projects and folders

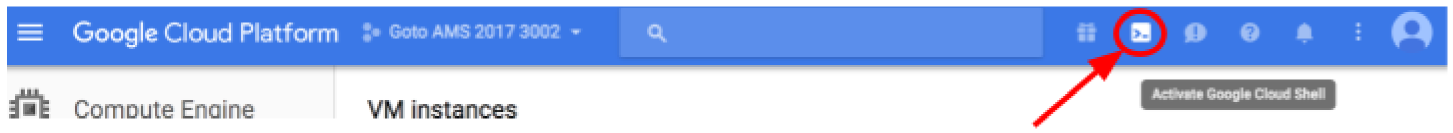
RECENT ALL

Name	ID
No organization	0
Spring IO Barcelona 2401	springio18-bcn-2401

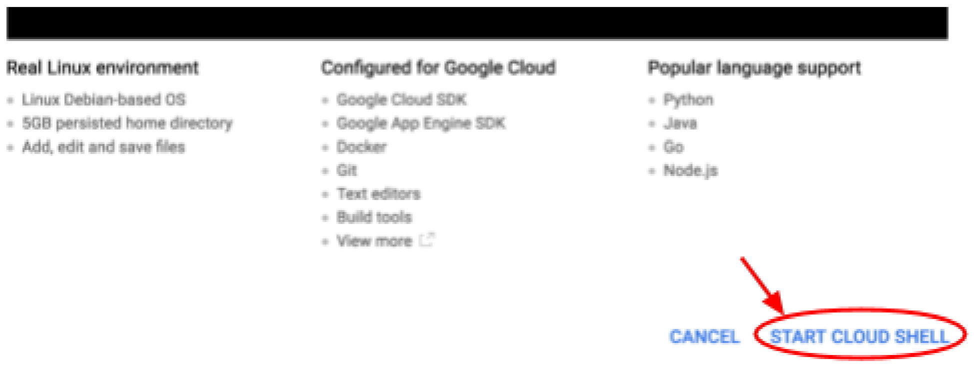
7. You should see the project name now reflected on the top bar.

Task 2 - Cloud Shell

You will do most of the work from the Google Cloud Shell, a command line environment running in the Cloud. This Debian-based virtual machine is loaded with all the development tools you'll need (`docker`, `gcloud`, `kubectl` and others) and offers a persistent 5GB home directory. Open the Google Cloud Shell by clicking on the icon on the top right of the screen:



1. When prompted, click **Start Cloud Shell**:


[redacted]

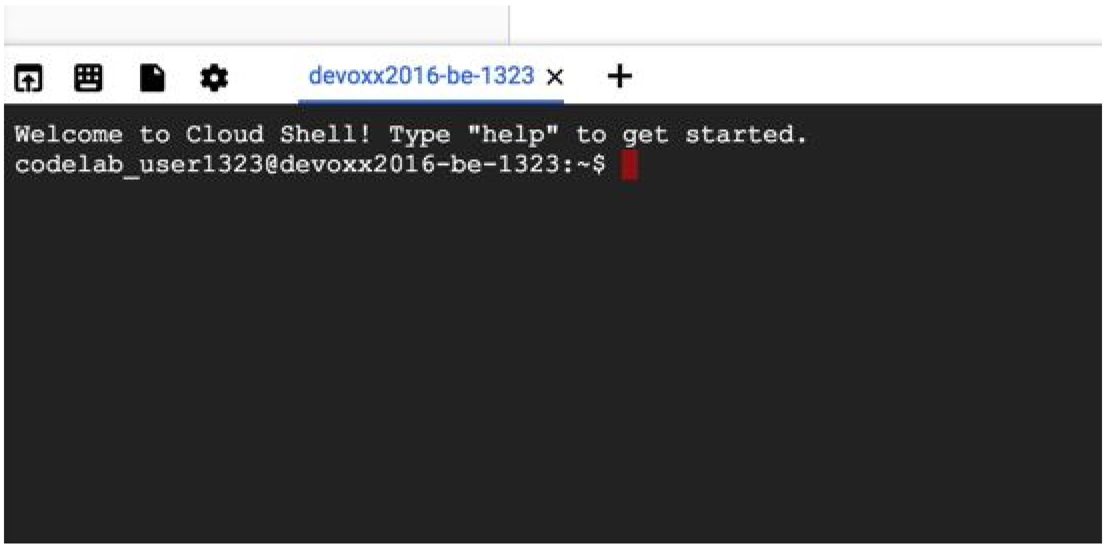
Real Linux environment
- Linux Debian-based OS
- 5GB persisted home directory
- Add, edit and save files

Configured for Google Cloud
- Google Cloud SDK
- Google App Engine SDK
- Docker
- Git
- Text editors
- Build tools
- View more

Popular language support
- Python
- Java
- Go
- Node.js

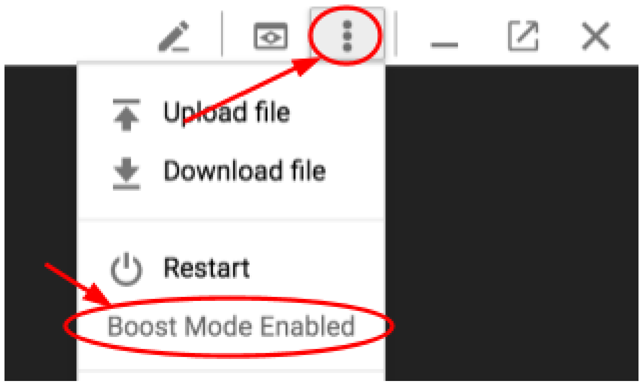
CANCEL START CLOUD SHELL

You should see the shell prompt at the bottom of the window:



```
Welcome to Cloud Shell! Type "help" to get started.
codelab_user1323@devoxx2016-be-1323:~$
```

2. Check to see if Cloud Shell's Boost mode is Enabled.



3. If not, then enable **Boost Mode** for Cloud Shell.

Note: When you run `gcloud` on your own machine, the config settings will be persisted across sessions. But in Cloud Shell, you will need to set this for every new session / reconnection.

Lab 2 - Create GKE Cluster

Duration: 15:00

You should perform all of the lab instructions directly in Cloud Shell.

Task 1 - Enable APIs

1. Enable Google Cloud APIs that you'll be using from our Kubernetes cluster.

```
gcloud services enable \
  cloudapis.googleapis.com \
  container.googleapis.com \
  containerregistry.googleapis.com
```

Task 2 - Create a Kubernetes cluster

1. Set the default zone and region.

```
gcloud config set compute/zone europe-west1-c
gcloud config set compute/region europe-west1
```

Note: For the lab, use the region/zone recommended by the instructor. Learn more about different zones and regions in Regions & Zones documentation.

2. Create a Kubernetes cluster in Google Cloud Platform is very easy! Use Kubernetes Engine to create a cluster:

```
gcloud beta container clusters create knative \
  --addons=HorizontalPodAutoscaling,HttpLoadBalancing \
  --machine-type=n1-standard-4 \
  --cluster-version=1.14 \
  --enable-stackdriver-kubernetes --enable-ip-alias \
  --enable-autoscaling --min-nodes=1 --max-nodes=10 \
  --enable-autorepair \
  --scopes cloud-platform
```

Note: The `scopes` parameter is important for this lab. Scopes determine what Google Cloud Platform resources these newly created instances can access. By default, instances are able to read from Google Cloud Storage, write metrics to Google Cloud Monitoring, etc. For our lab, we add the cloud-platform scope to give us more privileges, such as writing to Cloud Storage as well.

This will take a few minutes to run. Behind the scenes, it will create Google Compute Engine instances, and configure each instance as a Kubernetes node. These instances don't include the Kubernetes Master node. In Google Kubernetes Engine, the Kubernetes Master node is managed service so that you don't have to worry about it!

You can see the newly created instances in the **Compute Engine** → **VM Instances** page.

3. Grant cluster-admin permissions to the current user:

```
kubectl create clusterrolebinding cluster-admin-binding \
  --clusterrole=cluster-admin \
  --user=$(gcloud config get-value core/account)
```

Admin permissions are required to create the necessary RBAC rules for Knative.

Lab 3 - Install Istio

We need to manually install Istio and a Cluster Local Gateway:

```
kubectl apply -f
https://raw.githubusercontent.com/knative/serving/master/third_party/istio-1.3.3
/istio-crds.yaml

while [[ $(kubectl get crd gateways.networking.istio.io -o
jsonpath='{.status.conditions[?(@.type=="Established")].status}') != 'True' ]];
do
  echo "Waiting on Istio CRDs"; sleep 1
done

kubectl apply -f
https://raw.githubusercontent.com/knative/serving/master/third_party/istio-1.3.3
/istio.yaml

kubectl apply -f
https://raw.githubusercontent.com/knative/serving/master/third_party/istio-1.3.3
/istio-knative-extras.yaml
```

Lab 4 - Install Knative

Duration: 10:00

Task 1 - To install Knative, install the CRDs

```
kubectl apply --selector knative.dev/crd-install=true \
 -f https://github.com/knative/serving/releases/download/v0.10.0/serving.yaml \
 -f https://github.com/knative/eventing/releases/download/v0.10.0/release.yaml \
 -f https://github.com/knative/serving/releases/download/v0.10.0/monitoring.yaml
```

Task 2 - Now install the Knative components

```
kubectl apply \
 -f https://github.com/knative/serving/releases/download/v0.10.0/serving.yaml \
 -f https://github.com/knative/eventing/releases/download/v0.10.0/release.yaml \
 -f https://github.com/knative/serving/releases/download/v0.10.0/monitoring.yaml
```

Task 3 - Monitor the Knative components until all of the components show a `STATUS` of `Running`:

```
kubectl get pods --namespace knative-serving
kubectl get pods --namespace knative-eventing
kubectl get pods --namespace knative-monitoring
```

Lab 5 - Deploy a Simple Spring Boot App

Duration: 10:00

Task 1 - Create `hello-service.yaml` from the shell

```
cat << EOF > hello-service.yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: hello-springboot
spec:
  template:
    spec:
      containers:
        - image: gcr.io/cr-demo-235923/hello-springboot-mvn
EOF
```

Task 2 - Apply the CRD to deploy the image as a service

```
kubectl apply -f hello-service.yaml
```

Task 3 - Check the service

Get the external IP for the service:

```
export IP_ADDRESS=$(kubectl get svc istio-ingressgateway -n istio-system \
  -o 'jsonpath={.status.loadBalancer.ingress[0].ip}')
echo $IP_ADDRESS
```

Get the hostname for the service:

```
export HOST_NAME=$(kubectl get route hello-springboot \
  -o 'jsonpath={.status.url}' | cut -c8-)
echo $HOST_NAME
```

Curl the service:

```
curl -w'\n' -H "Host: $HOST_NAME" http://${IP_ADDRESS}
```

Lab 6 - Deploy a Service from Source

Duration: 15:00

Task 1 - Install Tekton & Buildpack support:

```
kubectl apply -f https://github.com/tektoncd/pipeline/releases/download/v0.8.0/release.yaml
```

Validate it is running:

```
kubectl get pods -n tekton-pipelines
```

Install buildpack tasks:

```
kubectl apply \
 -f https://raw.githubusercontent.com/tektoncd/catalog/master/buildpacks/buildpacks-v3.yaml
```

Task 2 - Create a service that builds and deploys a service from source:

1. Use Tekton and the Cloud Native Buildpack to build an image from source.

```
export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
echo $PROJECT_ID

cat << EOF > hello-service-cnb.yaml
apiVersion: tekton.dev/v1alpha1
kind: TaskRun
metadata:
  name: hello-springboot-mvn-build
spec:
  taskRef:
    name: buildpacks-v3
  podTemplate:
    volumes:
    - name: my-cache
  inputs:
    resources:
    - name: source
      resourceSpec:
        type: git
        params:
        - name: url
```

```
          value: https://github.com/jamesward/hello-springboot-mvn.git
    params:
    - name: BUILDER_IMAGE
      value: heroku/buildpacks
    - name: USE_CRED_HELPERS
      value: "true"
    - name: CACHE
      value: my-cache
  outputs:
    resources:
    - name: image
      resourceSpec:
        type: image
        params:
        - name: url
          value: gcr.io/$PROJECT_ID/hello-springboot-mvn-knative
EOF
```

3. Create the build

```
kubectl apply -f hello-service-cnb.yaml
```

Task 3 - Get the build status and logs:

```
kubectl get taskruns/hello-springboot-mvn-build -o yaml
```

This will kick off the build. See that the build is actually taking place inside of a Kubernetes Pod.

```
kubectl get pods
```

You should see one of the pod that's named `hello-springboot-mvn-build-pod-....` Each step in the build step is converted into an initialization container of the pod and executed in sequence.

Check the build output with:

```
kubectl logs -f $(kubectl get taskruns/hello-springboot-mvn-build \
  -ojsonpath={.status.podName}) -c step-build
```

Note: If the build hasn't reached the actual build step, then you'll see an error from the command line. Retry the command again.

Task 4 - Once the build is complete deploy a new revision:

```
cat << EOF > hello-service.yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: hello-springboot
spec:
  template:
    metadata:
      name: hello-springboot-2
    spec:
      containers:
        - image: gcr.io/$PROJECT_ID/hello-springboot-mvn-knative
EOF
```

Apply the config:

```
kubectl apply -f hello-service.yaml
```

Task 5 - View the revisions (which you should have 2):

```
kubectl get revisions
```

Task 6 - Check the service:

```
curl -w'\n' -H "Host: $HOST_NAME" http://${IP_ADDRESS}
```

Lab 7 - Custom Domain

Task 1 - Set the config

Echo your ingress IP:

```
export IP_ADDRESS=$(kubectl get svc istio-ingressgateway -n istio-system \
  -o 'jsonpath={.status.loadBalancer.ingress[0].ip}')
echo $IP_ADDRESS
```

Create a file named `custom-domain.yaml` containing the following and replacing IP_ADDRESS with the output of the above:

```
cat <<EOF > custom-domain.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-domain
  namespace: knative-serving
data:
  $IP_ADDRESS.nip.io: ""
EOF
```

Apply the config:

```
kubectl apply -f custom-domain.yaml
```

Task 2 - Test it in your browser by loading the URL from:

```
echo "http://hello-springboot.default.$IP_ADDRESS.nip.io"
```

Lab 8 - Cold Starts / Min Instances

Task 1 - Check current number of pods

```
kubectl get pods
```

Look for pod names containing "deployment" - there may be none.

Task 2 - Create a file named `autoscale.yaml` containing

```
cat << EOF > autoscale.yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: hello-springboot
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/class: hpa.autoscaling.knative.dev
        autoscaling.knative.dev/metric: cpu
        autoscaling.knative.dev/target: "15"
        autoscaling.knative.dev/minScale: "5"
        autoscaling.knative.dev/maxScale: "10"
    spec:
      containers:
        - image: gcr.io/cr-demo-235923/hello-springboot-mvn
EOF
```

Task 3 - Apply this change

```
kubectl apply -f autoscale.yaml
```

Task 4 - Check the number of pods again and you should see at least 5 pods

```
kubectl get pods
```

Task 5 - Install "hey" a load testing tool

```
wget -O hey https://storage.googleapis.com/hey-release/hey_linux_amd64
chmod a+x hey
```

Task 6 - In a new Cloud Shell tab, watch the pods

```
watch kubectl get pods
```

Task 7 - Do a 60 second load test with 50 concurrent requests

```
export IP_ADDRESS=$(kubectl get svc istio-ingressgateway -n istio-system \
  -o 'jsonpath={.status.loadBalancer.ingress[0].ip}')
echo $IP_ADDRESS

./hey -z 60s -c 50 -disable-keepalive \
  http://hello-springboot.default.$IP_ADDRESS.nip.io
```

Watch the number of pods go up to handle the load.

Note: It may take 30 seconds to a minute before new instances are spun up.

Task 8 - Scale Back Down

Edit the `autoscale.yaml` file and set the minScale to 0, then re-apply it:

```
kubectl apply -f autoscale.yaml
```

Lab 9 - Traffic Splitting

Task 1 - Deploy a new revision with 0% traffic

Create a new file named `bluegreen.yaml`:

```
export CURRENT_REV=$(kubectl get configurations hello-springboot \
  -o=jsonpath='{.status.latestCreatedRevisionName}')
echo $CURRENT_REV

cat << EOF > bluegreen.yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: hello-springboot
spec:
  template:
    metadata:
      name: hello-springboot-canary
    spec:
      containers:
        - image: gcr.io/cr-demo-235923/hello-springboot-mvn
  traffic:
  - tag: current
    revisionName: $CURRENT_REV
    percent: 100
  - tag: canary
    revisionName: hello-springboot-canary
    percent: 0
EOF
```

Apply the change:

```
kubectl apply -f bluegreen.yaml
```

Check out the traffic split:

```
kubectl get ksvc hello-springboot -o yaml
```

You will see that each revision has a unique URL which you can access.

Check out the new canary revision that currently has 0% traffic:

```
export IP_ADDRESS=$(kubectl get svc istio-ingressgateway -n istio-system \
  -o 'jsonpath={.status.loadBalancer.ingress[0].ip}')

curl -w'\n' http://canary-hello-springboot.default.$IP_ADDRESS.nip.io
```

Task 2 - Route 20% of traffic to the new version

Update the `bluegreen.yaml` file and change the current percent to 80 and the latest to 20. Do not copy and paste this section - just update the percent in your existing file. You need to keep the revisionName set in the previous step.

```
  traffic:
  - tag: current
    revisionName: CURRENT_REV
    percent: 80
  - tag: canary
    revisionName: hello-springboot-canary
    percent: 20
```

Apply the change:

```
kubectl apply -f bluegreen.yaml
```

Make a few requests in your browser to see some of the requests being routed to the new revision:

```
export IP_ADDRESS=$(kubectl get svc istio-ingressgateway -n istio-system \
  -o 'jsonpath={.status.loadBalancer.ingress[0].ip}')

while true; do curl -w'\n' \
  http://hello-springboot.default.$IP_ADDRESS.nip.io; done
```

Lab 10 - Internal Service

By default, all services are accessible from outside of the cluster. You can make a service accessible from within a cluster (cluster local) by assigning the visibility label to the Route or the Knative Service.

Task 1 - Redeploy the service

```
kubectl apply -f hello-service.yaml
```

Task 2 - Make the Helloworld service internal

```
kubectl label ksvc hello-springboot \
  serving.knative.dev/visibility=cluster-local
```

Task 3 - Check the Route

```
kubectl get ksvc hello-springboot
```

Notice that the URL for `hello-springboot` is now suffixed with `svc.cluster.local`. It's no longer accessible via the public ingress.

```
curl -v http://hello-springboot.default.$IP_ADDRESS.nip.io
```

This returns a 404.

Similarly, you can't invoke the service from the public ingress even if you specify the hostname.

```
export IP_ADDRESS=$(kubectl get svc istio-ingressgateway -n istio-system \
  -o 'jsonpath={.status.loadBalancer.ingress[0].ip}')

export HOST_NAME=$(kubectl get route hello-springboot \
  -o 'jsonpath={.status.url}' | cut -c8-)

curl -v -H "Host: $HOST_NAME" http://${IP_ADDRESS}
```

However, the service is accessible within the Kubernetes cluster by pods running inside of the cluster.

Lab 11 - Events

Task 1 - Enable a default broker

```
kubectl label namespace default knative-eventing-injection=enabled
```

Task 2 - Create a new Spring Boot project

```
curl https://start.spring.io/starter.tgz \
  -d bootVersion=2.2.0.RELEASE \
  -d applicationName=UpperFunction \
  -d dependencies=webflux \
  -d name=spring-upper \
  -d artifactId=spring-upper \
  -d baseDir=upper | tar -xzvf -
```

Switch to the new `upper` dir

```
cd upper
```

Task 3 - Add a CloudEvent function that uppercases the message

Edit the pom.xml file and add the cloudevents-api library:

```
<dependency>
    <groupId>io.cloudevents</groupId>
    <artifactId>cloudevents-api</artifactId>
    <version>0.3.1</version>
</dependency>
```

Update the code to have a request handler that parses a CloudEvent and returns a modified one:

```
cat << EOF > src/main/java/com/example/springupper/UpperFunction.java
package com.example.springupper;

import io.cloudevents.CloudEvent;
import io.cloudevents.format.Wire;
import io.cloudevents.v03.AttributesImpl;
import io.cloudevents.v03.CloudEventBuilder;
import io.cloudevents.v03.http.Marshallers;
import io.cloudevents.v03.http.Unmarshallers;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.util.UriComponentsBuilder;

import java.time.ZonedDateTime;
import java.util.HashMap;
import java.util.Map;
import java.util.UUID;

@SpringBootApplication
@RestController
public class UpperFunction {

    public static void main(String[] args) {
        SpringApplication.run(UpperFunction.class, args);
    }

    @RequestMapping
    public ResponseEntity<String> upper(@RequestHeader HttpHeaders headers,
UriComponentsBuilder uriComponentsBuilder, @RequestBody String body) {
        Map<String, Object> reqHeaders = new
HashMap<>(headers.toSingleValueMap());
        reqHeaders.put(HttpHeaders.CONTENT_TYPE, "application/json");

        CloudEvent<AttributesImpl, Data> event =
                Unmarshallers.binary(Data.class)
                        .withHeaders(() -> reqHeaders)
                        .withPayload(() -> body)
                        .unmarshal();

        Data respData = event.getData().get();
        respData.message = respData.message.toUpperCase();

        CloudEvent<AttributesImpl, Data> respEvent =
                CloudEventBuilder.<Data>builder()
                        .withType("spring-upper")
                        .withSource(uriComponentsBuilder.build().toUri())
                        .withId(UUID.randomUUID().toString())
                        .withTime(ZonedDateTime.now())
                        .withDatacontenttype("application/json")
                        .withData(respData)
                        .build();

        Wire<String, String, String> wire = Marshallers.<Data>binary()
                .withEvent(() -> respEvent)
                .marshal();
        MultiValueMap<String, String> respHeaders = new LinkedMultiValueMap<>();
        respHeaders.setAll(wire.getHeaders());

        return new ResponseEntity<>(wire.getPayload().get(), respHeaders,
HttpStatus.OK);
    }

    public static class Data {
        public String message;
    }

}
EOF
```

Task 4 - Run the application in Cloud Shell

```
./mvnw -q spring-boot:run
```

Note: This command runs Maven in quiet mode. This suppresses unnecessary Maven download messages. Be patient as the build is actually taking place. Wait for the application to start.

Task 5 - Test the service

In a different Cloud Shell session tab:

```
curl -v -H 'Content-Type:' -X POST \
  -H 'Ce-Id: 1234' \
  -H 'Ce-Source: cronjob-spring-upper' \
  -H 'Ce-Specversion: 0.3' \
  -H 'Ce-Type: dev.knative.cronjob.event' \
  -d '{"message": "hello, world"}' \
  http://localhost:8080/
```

Task 6 - Create the container image and store it in GCR

```
export PROJECT_ID=$(gcloud config get-value core/project)

./mvnw clean compile com.google.cloud.tools:jib-maven-plugin:build \
  -Dimage=gcr.io/${PROJECT_ID}/upper-function
```

Task 7 - Create an events.yaml file

```
export PROJECT_ID=$(gcloud config get-value core/project)

cat << EOF > events.yaml
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: CronJobSource
metadata:
  name: cronjob-spring-upper
spec:
  schedule: "* * * * *"
  data: '{"message": "hello, world"}'
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: Broker
      name: default
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: spring-upper
  labels:
   serving.knative.dev/visibility: cluster-local
spec:
  template:
    spec:
      containers:
        - image: gcr.io/$PROJECT_ID/upper-function
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: event-display
  labels:
   serving.knative.dev/visibility: cluster-local
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-releases/github.com/knative/eventing-sources/cmd/event_display
---
apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: upper-trigger
spec:
  broker: default
  filter:
    sourceAndType:
      type: dev.knative.cronjob.event
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1alpha1
      kind: Service
      name: spring-upper
---
apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: event-display-trigger
spec:
  broker: default
  filter:
    sourceAndType:
      type: spring-upper
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1alpha1
      kind: Service
      name: event-display
EOF
```

Task 8 - Apply the config

```
kubectl apply -f events.yaml
```

Task 9 - Check that the events are being sent and received (which might take up to a minute to see the first one)

```
kubectl logs -l serving.knative.dev/service=event-display -c user-container
```

Optionally, use kail to tail the log and see the new log entries streamed.

```
cd $HOME

wget \
https://github.com/boz/kail/releases/download/v0.10.1/kail_0.10.1_linux_amd64.tar.gz

tar -xzvf kail_0.10.1_linux_amd64.tar.gz

./kail -l serving.knative.dev/service=event-display
```

More Code Labs

1. Kubernetes Lab - bit.ly/k8s-lab
2. Istio Lab - bit.ly/istio-lab
3. Google Cloud Native with Spring Boot, App Engine and Kubernetes - bit.ly/spring-gcp-lab
4. Serverless with Spring Cloud Function, Riff, and Knative - bit.ly/spring-riff-lab
5. Knative - bit.ly/knativeworkshop
6. gRPC Microservices with Java - bit.ly/grpc-java-lab-doc