Serverless with Spring Cloud, riff and Knative

Self-link: bit.ly/spring-riff-lab

Source and Slides: https://github.com/trisberg/s1p2018-serverless/

Author: Ray Tsang (@saturnism), Thomas Risberg (@trisberg)

Author	Ray Tsang
ID	spring-riff-knative
Summary	In this lab, you'll quickly create several functions with Spring Cloud, and deployed in a riff environment. Riff is a serverless platform built on top of Knative.
Tags	web, kiosk, spring, gcp
Categories	spring, microservices, riff, function, serverless, java, knative, istio

Overview

Duration: 5:00

In this lab, you'll quickly create several functions with Spring Cloud, and deployed in a riff environment. Riff is a serverless platform built on top of Knative.

This lab also uses Spring Cloud GCP, which provides a set of idiomatic Spring Boot Starters to leverage Google Cloud Platform services easily. There is a Spring Cloud GCP Codelab that you can try on your own.

What is your experience level with Google Cloud Platform?

- I have just heard of Google Cloud Platform
- I have played around with Google Cloud Platform
- I use Google Cloud Platform in production

Lab 1 - Initial setup

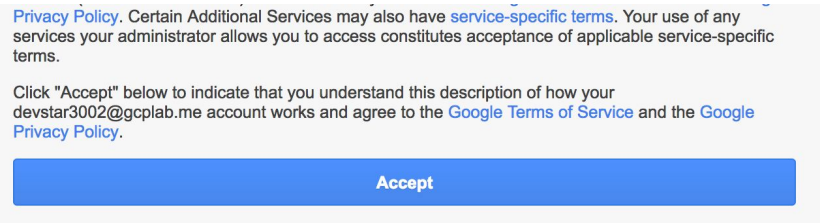
Duration: 20:00

Task 1 - Logging In

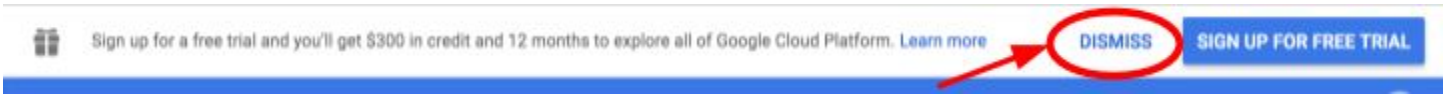
1. The instructor will provide you with a temporary username / password to sign in to the Google Cloud Platform Console.

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

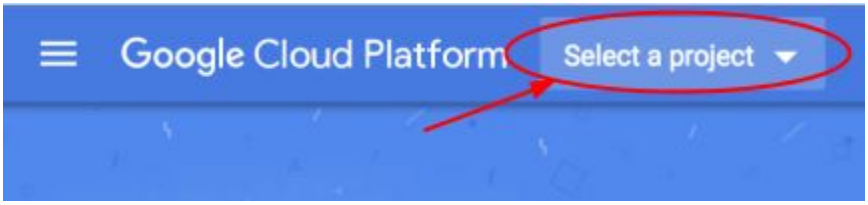
2. Sign in to the Google Cloud Platform Console: https://console.cloud.google.com/ with the provided credentials. In **Welcome to your new account** dialog, click **Accept**.



3. 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.

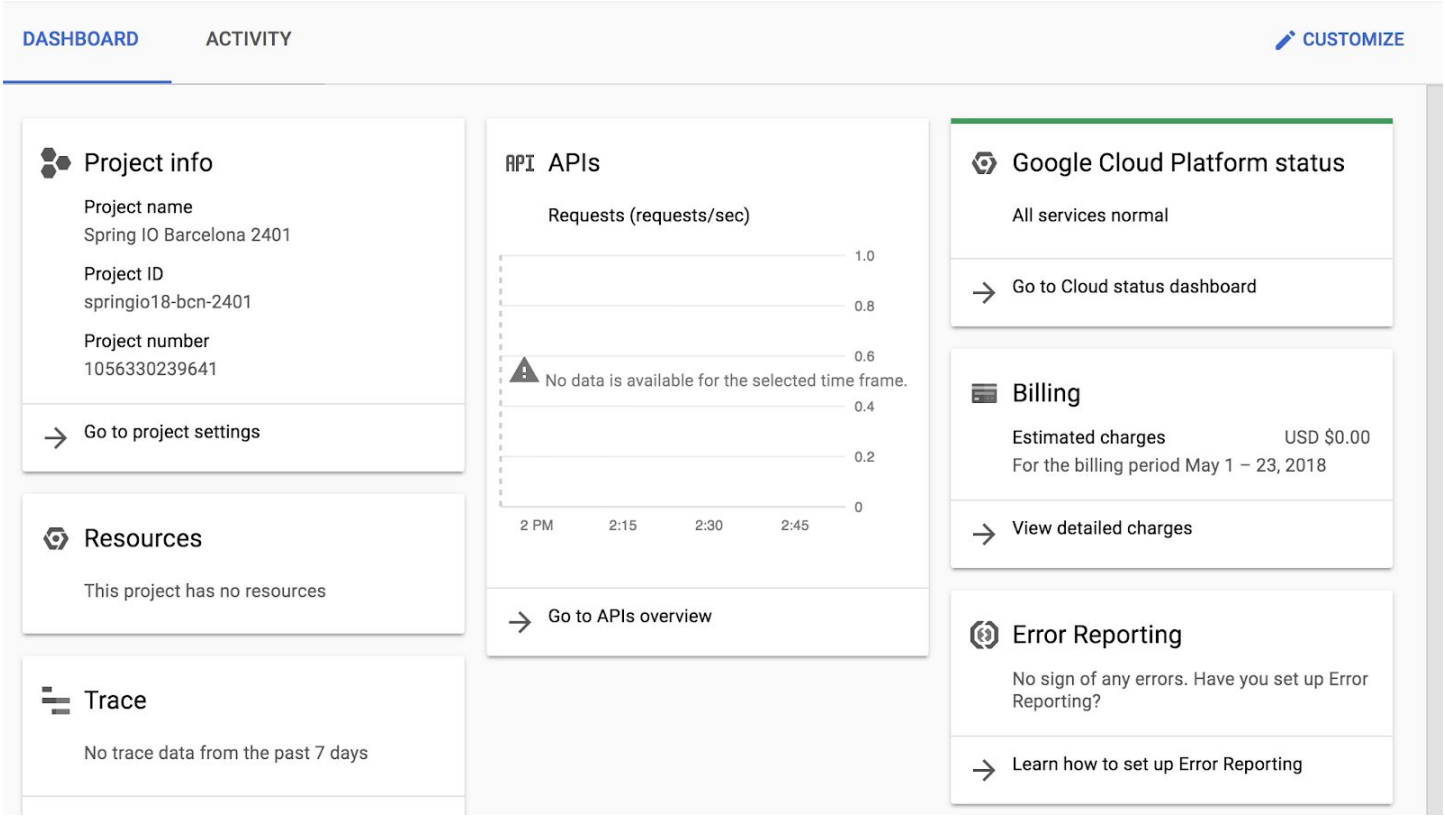


4. Click **Select a project**.



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

You should see the Project Dashboard:

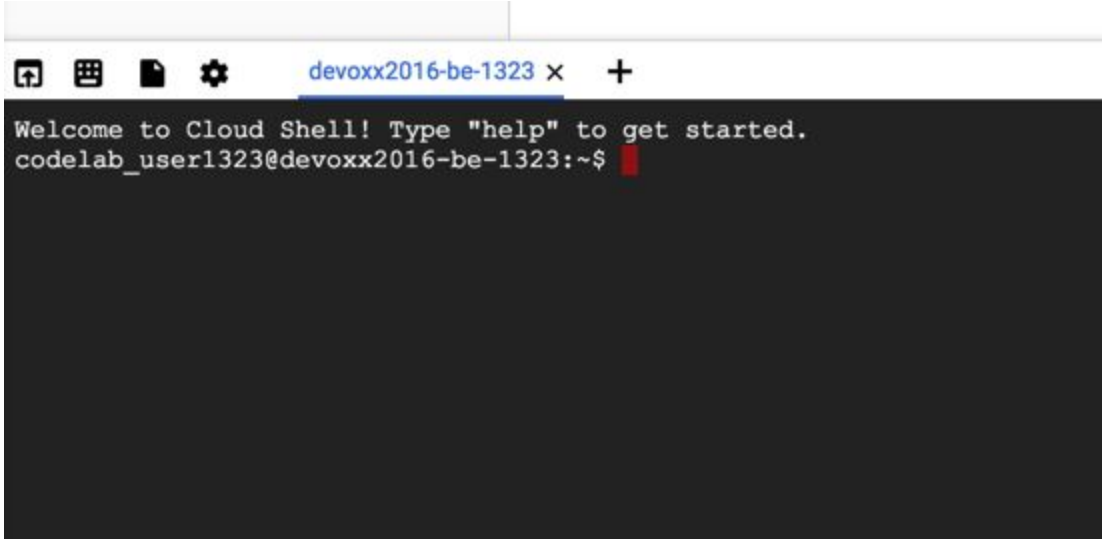


Task 2 - Cloud Shell

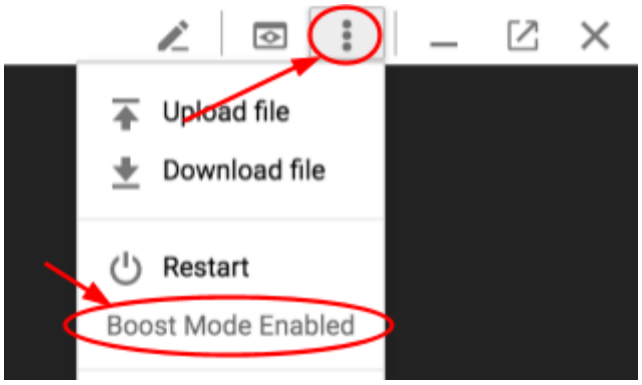
You will do most of the work from the Google Cloud Shell, a command line environment running in the Cloud. This Debian-based virtual machine is loaded with all the development tools you'll need (`docker`, `gcloud`, `kubectl` and others) and offers a persistent 5GB home directory. Open the Google Cloud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

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

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



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 - Bootstrap Kubernetes, Istio, Knative, riff

Duration: 30:00

You should perform all of the lab instructions directly in Cloud Shell. Although there seems to be a lot of moving parts - you don't need to worry. After the initial Kubernetes provisioning, riff will provision everything else.

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
```

If you are having trouble with this command line, try this:

Task 2 - Create a Kubernetes cluster

1. Set the default zone and region.

```
$ gcloud config set compute/zone us-central1-c
$ gcloud config set compute/region us-central1
```

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 container clusters create riff-cluster \
  --cluster-version 1.12 \
  --machine-type=n1-standard-4 \
  --num-nodes 4 \
  --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.

Task 3 - Install Istio, Knative and Riff

The easiest way to install the rest of the environment is using riff command line interface.

1. Install riff CLI.

```
$ curl -Lo riff-linux-amd64.tgz \
```

```
  https://github.com/projectriff/riff/releases/download/v0.3.0/riff-linux-amd64.tgz
$ mkdir -p ~/riff-bin/
$ tar xvzf riff-linux-amd64.tgz -C ~/riff-bin
$ echo export 'PATH=~/riff-bin:$PATH' >> ~/.bashrc
$ source ~/.bashrc
```

Note: You would normally install riff CLI into `/usr/local/bin` directory. In Cloud Shell, only your home directory is persisted across multiple sessions. Thus, in this lab, the instructions installs the riff CLI into your home directory `riff-bin` directory instead.

2 Validate riff CLI was installed properly.

```
$ riff version
Version
  riff cli: ...
```

3. Grant the cluster administrator privileges to yourself.

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

4. Install riff to your Kubernetes cluster.

```
$ riff system install
```

This simple command line will install riff and all of the dependent components (Knative, Istio, etc) into the Kubernetes cluster.

Note: In case something went wrong because of a missing step above, you may need to clean the riff installation before re-installing all the components. To uninstall riff:

```
$ riff system uninstall --istio --force
```

5. Validate that all pods are up and running.

```
$ kubectl get pods --all-namespaces
```

This will list all pods from all the namespaces (e.g., `istio-system`, `knative-serving`, `knative-build`, `knative-eventing`, ...). Make sure all the pods in each of the namespaces has the status of `Running`.

Task 4 - Create a Push Credential

Both Knative and riff has a build pipeline that can build from source to Docker image and stored into a Docker registry. Google Cloud Platform comes with a private registry for every project, known as the Google Container Registry. Knative and riff environment needs a credential to push to this registry.

1. Create a new service account that can push to Google Container Registry.

```
$ gcloud iam service-accounts create push-image
```

2. Assign IAM permission to the service account.

```
$ export GCP_PROJECT=$(gcloud config get-value core/project)
$ gcloud projects add-iam-policy-binding $GCP_PROJECT \
  --member serviceAccount:push-image@$GCP_PROJECT.iam.gserviceaccount.com \
  --role roles/storage.admin
```

3. Create a JSON key.

```
$ gcloud iam service-accounts keys create \
  --iam-account "push-image@$GCP_PROJECT.iam.gserviceaccount.com" \
  ~/push-image.json
```

Warning: Treat the service account JSON key file as your own username/password. Do not share this in the public!

4. Initialize the default namespace with the push credentials you created.

```
$ riff namespace init default --gcr $HOME/push-image.json
```

This creates a Kubernetes service account named `riff-build` that is associated with a `push-credentials` secret that is also created. Later in the lab when riff builds a container from source, it'll create a new Knative build job and use the `riff-build` service account. This allows riff to use the GCP service account key to authenticate and push to Google Container Registry (GCP's private container registry).

5. Inspect the `riff-build` service account and see it references the `push-credentials` secret.

```
$ kubectl get sa riff-build -oyaml
...
secrets:
- name: push-credentials
- name: riff-build-token-...
```

Lab 3 - Deploy a Simple Function

Duration: 10:00

The hard part is done! Once the environment is setup, you can now easily deploy functions and/or containers in the riff/Knative environment! This lab will start with something simple - a Spring Cloud Function.

Task 1 - Deploy a riff function

1. Deploy a simple Java function that will take in a String input and uppercase it.

```
$ export GCP_PROJECT=$(gcloud config get-value core/project)
$ riff function create uppercase \
  --git-repo https://github.com/trisberg/uppercase.git \
  --handler uppercase \
  --image gcr.io/$GCP_PROJECT/uppercase-function \
  --verbose
```

The build can take up to a minute to get started on a new cluster because it takes some time to download the build images. Once they are downloaded you will see some output from the build.

Note: The build features are getting improved and *riff* now includes an option to use Cloud Native Buildpacks (https://buildpacks.io/). This allows you to compile from source rather than having to push a jar file to a repository. In the above example the jar file has been pushed to a branch named *jar* so we specify that with the `--git-revision` option.

A lot of things are happening behind this simple command!

2. Riff will create a Knative service called `uppercase`.

```
$ kubectl get service.serving
NAME         AGE
uppercase    1m
```

3. Invoke the function using riff CLI.

```
$ riff service invoke uppercase --text -- -d'ray' -w'\n'
curl ...
RAY
```

Congratulations! You've just deployed your first *riff* function!

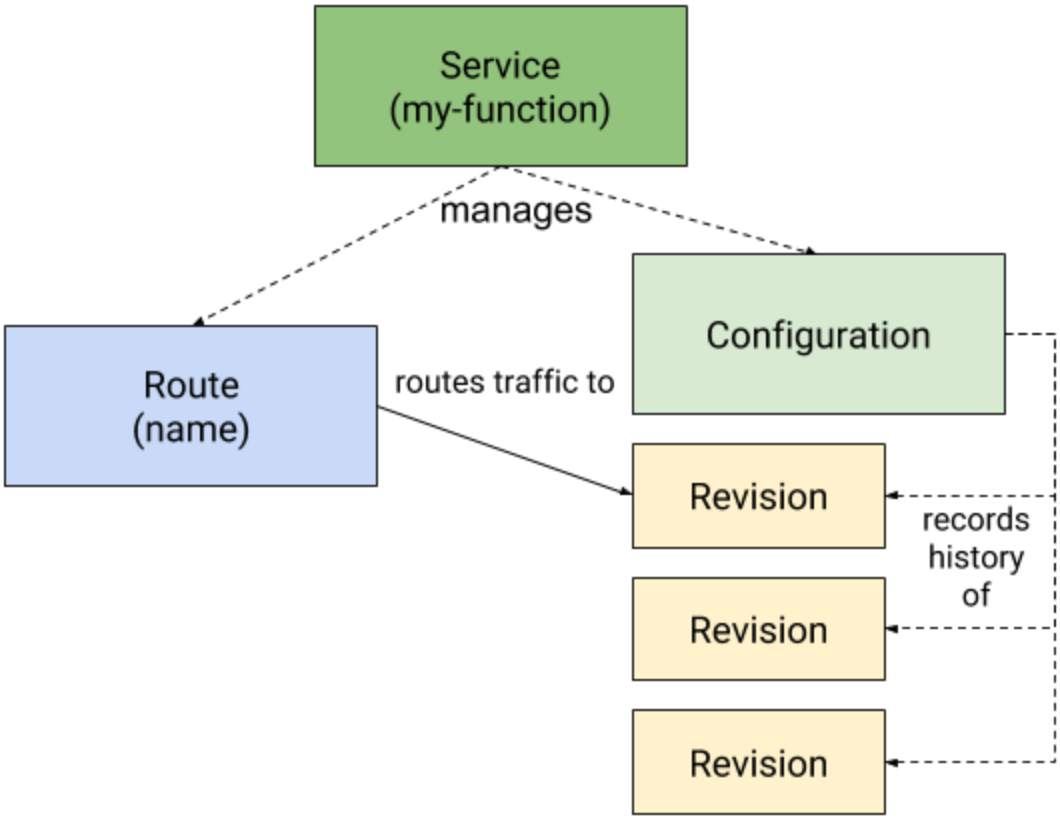
Note: We'll examine what happened behind the scenes during the next lab!

Lab 4 - From riff to Knative

Duration: 30:00

Task 1 - Examine behind the scenes details

A riff Function is actually a Knative Service behind the scene. A Knative Service creates and manages 2 additional resources behind the scenes: Configuration and Route. Details are in the Knative Serving documentation. Examine each component carefully.



1. Examine the Knative service in detail.

```
$ kubectl get service.serving uppercase -oyaml
```

This service has a build configuration (how it's built) and a revision configuration (how it's run):

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  ...
spec:
  # Route the traffic to the latest successfully built function.
  runLatest:
    configuration:
      build:
```

```
    # The service account contains the push-credentials secret
    # Examine with `kubectl get sa riff-build -oyaml`
    serviceAccountName: riff-build

    # The git repository we specified in `--git-repo` argument
    source:
      git:
        revision: master
        url: https://github.com/trisberg/uppercase.git
    # Use `riff` build template and configuration arguments for the build.
    template:
      arguments:
      # The name of the image from `--image` argument
      - name: IMAGE
        value: ...
      - name: FUNCTION_ARTIFACT
        value: ""
      - name: FUNCTION_HANDLER
        value: uppercase
      - name: FUNCTION_LANGUAGE
        value: ""
      # Specify the `riff-cnb` build template.
      # Examine the template with `kubectl get buildtemplates riff -oyaml`
      name: riff-cnb
  revisionTemplate:
    metadata:
      ...
    spec:
      concurrencyModel: Multi
      # The name of the image from `--image` argument
      container:
        image: gcr.io/YOUR_GCP_PROJECT/uppercase-function
```

2. The `configuration` block will be copied into the `Configuration` resource.

```
$ kubectl get configuration uppercase -oyaml
...
```

3. A configuration can have multiple revisions. Everytime you update the configuration, a new revision is created.

```
$ kubectl get revision
NAME                AGE
uppercase-00001     4h
```

4. For each revision, a build is executed if needed.

```
$ kubectl get pods --show-all
NAME                    READY     STATUS      RESTARTS   AGE
uppercase-00001-...     0/1       Completed   0          ...
```

5. If the build status is Completed, then you can see the container image built and stored in Google Container Registry.

```
$ gcloud container images list
NAME
gcr.io/YOUR_GCP_PROJECT/uppercase-function
```

6. A Kubernetes deployment is created and it is automatically scaled based on load. If the function is not being used, then it'll scale to 0 instances.

```
$ kubectl get deployment
NAME                          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
uppercase-00001-deployment    0         0         0            0           20m
```

7. A Knative route was created. Because riff configured the service to use `runLast` configuration, it'll automatically route 100% of the traffic to the latest revision of the build.

```
$ kubectl get route uppercase -oyaml
```

The generated configuration looks like the following:

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  ...
spec:
  # Route 100% of the traffic to the latest `uppercase` configuration
  traffic:
  - configurationName: uppercase
    percent: 100
status:
  conditions:
    ...
  # The fully qualified host header that must be used to invoke this service.
  domain: uppercase.default.example.com
  domainInternal: uppercase.default.svc.cluster.local

  # Route 100% of the traffic to the revision you just built.
  traffic:
  - configurationName: uppercase
    percent: 100
    revisionName: uppercase-00001
```

Task 2 - Invoke the function using riff CLI

1. Earlier, you already used the riff CLI to invoke the function.

```
$ riff service invoke uppercase --text -- -d'thomas' -w'\n'
curl ...
THOMAS
```

What happened behind the scenes is that the riff CLI automatically discovered the public IP address that's fronted by an HTTP load balancer. It also automatically set the `Host` header of the HTTP request to indicate which function to invoke. Behind the scenes, it actually used `curl` to make the request.

Task 3 - Invoke the function using curl

You can follow the steps to understand what riff is doing behind the scenes.

1. Find the public ingress IP address.

```
$ export INGRESS_IP=`kubectl get svc istio-ingressgateway -n istio-system \
  -o jsonpath="{.status.loadBalancer.ingress[*].ip}"`
$ echo $INGRESS_IP
```

2. Find the fully qualified domain name for the function to passed via the header.

```
$ export FUNCTION_HOST=`kubectl get services.serving/uppercase \
  --namespace=default \
  --output=jsonpath="{.status.domain}"`
$ echo $FUNCTION_HOST
```

3. Make the request using curl.

```
$ curl \
  -H "Host: ${FUNCTION_HOST}" \
  -H "Content-Type: text/plain" \
  -d thomas \
  -w'\n' \
  http://${INGRESS_IP}
```

Lab 5 - Create & Deploy the Cat Not Cat Function

Duration: 30:00

Task 1 - Enable Vision API

1. Enable the Vision API so we can use it to analyze the uploaded images.

```
$ gcloud services enable vision.googleapis.com
Waiting for async operation operations/... to complete...
Operation finished successfully. The following command can describe the
Operation details:
 gcloud services operations describe operations/...
```

Task 2 - Create an Application Service Account

1. Create a service account with access to the Vision API.

```
$ gcloud iam service-accounts create catnotcat
```

Warning: This creates a service account with the Project Editor role. In your production environment, you should only assign roles and permissions that the application actually needs.

2. Generate the JSON key file to be used by the application to identify itself using the service account.

```
$ export GCP_PROJECT=$(gcloud config list --format 'value(core.project)')
$ gcloud iam service-accounts keys create \
  --iam-account catnotcat@${GCP_PROJECT}.iam.gserviceaccount.com \
  ~/catnotcat.json
```

Task 3 - Create a Spring Cloud Function

1. Create a new Spring Cloud Function using Spring Initializr.

```
$ cd ~/
$ curl https://start.spring.io/starter.tgz \
  -d bootVersion=2.1.5.RELEASE \
  -d applicationName=CatFunction \
  -d dependencies=cloud-function,webflux,cloud-gcp \
  -d name=catnotcat \
  -d artifactId=catnotcat \
  -d baseDir=catnotcat | tar -xzvf -
$ cd catnotcat
```

2. Add the Google Cloud Vision client library.

catnotcat/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
     ...
```

```
    <dependencies>
        ...
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-gcp-starter-vision</artifactId>
        </dependency>

    </dependencies>

    <dependencyManagement>
        ...
    </dependencyManagement>
    ...
</project>
```

2. Create a Cat Not Cat Function that will take in a Base64 encoded image input and then passing that the to Vision API to find out whether it's a picture containing a cat or not.

catnotcat/src/main/java/com/example/catnotcat/CatFunction.java

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ByteArrayResource;
import org.springframework.cloud.gcp.vision.CloudVisionTemplate;

import com.google.cloud.vision.v1.Feature;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Base64;
import java.util.function.Function;


@SpringBootApplication
public class CatFunction {

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

    @Bean
    Function<Flux<String>, Mono<String>> catnotcat(
        CloudVisionTemplate visionTemplate) {

        return (in) ->
            in.map(Base64.getDecoder()::decode)
                .map(bytes -> new ByteArrayResource(bytes))
```

```
                        .map(resource -> visionTemplate
                             .analyzeImage(resource,
                                  Feature.Type.LABEL_DETECTION))
                        .flatMapIterable(response -> response
                             .getLabelAnnotationsList())
                        .any(label -> label.getDescription()
                                  .equalsIgnoreCase("cat") &&
                             label.getScore() >= 0.90f)
                        .map(cat -> cat ? "cat" : "not cat");
    }

}
```

Note: The client implements AutoCloseable, thus its lifecycle is automatically managed by Spring as well.

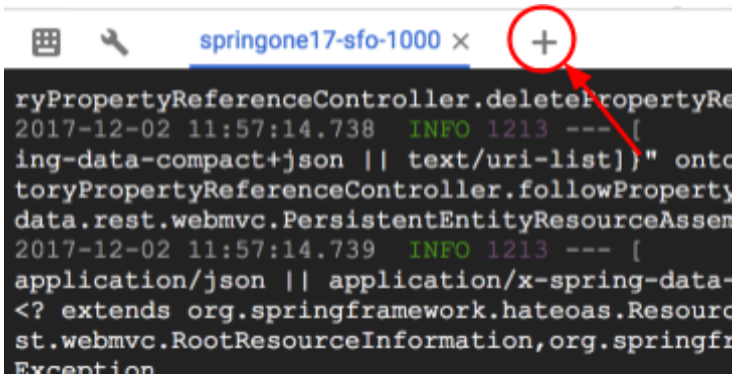
Task 4 - Run it locally

1. Run it locally with the service account credentials.

```
$ cd ~/catnotcat
$ ./mvnw -q -DskipTests spring-boot:run \
  -Dspring.cloud.gcp.credentials.location=file:///$HOME/catnotcat.json
```

Note: This instruction uses `-q` parameter to suppress excessive output from Maven that may cause issues within Cloud Shell. This is not required. But since the output is supressed, please be patient while waiting for the dependencies to download and the application to be built.

2. Open a new Cloud Shell session tab.



3. In the new Cloud Shell tab, download a cat picture from Wikipedia (CCA Share Alike 4.0, Wikipedia)

```
$ cd ~/catnotcat
$ curl -Lo cat.jpg \
https://upload.wikimedia.org/wikipedia/commons/thumb/4/4f/Felis_silvestris_catus
_lying_on_rice_straw.jpg/320px-Felis_silvestris_catus_lying_on_rice_straw.jpg
```

4. Make a `curl` request and send the cat image in Base64 encoded form.

```
$ base64 -w0 cat.jpg | curl -H"Content-Type: text/plain" -w "\n" \
  -d@- localhost:8080
cat
```

5. Similarly, download a dog picture (CCA Share Alike 4.0, Wikipedia), and make the request.

```
$ curl -Lo dog.jpg \
https://upload.wikimedia.org/wikipedia/commons/thumb/b/b5/Golden_Retriever_mediu
m-to-light-coat.jpg/320px-Golden_Retriever_medium-to-light-coat.jpg
$ base64 -w0 dog.jpg | curl -H"Content-Type: text/plain" -w "\n" \
  -d@- localhost:8080
not cat
```

6. Stop the Java process.

Taks 5 - Containerize with Jib

1. Use Jib to build the container.

```
$ cd ~/catnotcat
$ export GCP_PROJECT=$(gcloud config get-value core/project)
$ ./mvnw clean compile com.google.cloud.tools:jib-maven-plugin:build \
  -Dimage=gcr.io/${GCP_PROJECT}/catnotcat-function
...
[INFO]
[INFO] Containerizing application to gcr.io/.../catnotcat-function...
[INFO]
[INFO] Retrieving registry credentials for gcr.io...
[INFO] Getting base image gcr.io/distroless/java...
[INFO] Building dependencies layer...
[INFO] Building resources layer...
[INFO] Building classes layer...
[INFO] Finalizing...
[INFO]
[INFO] Container entrypoint set to [java, -cp,
/app/resources/:/app/classes/:/app/libs/*, com.example.demo.UpperFunction]
[INFO]
[INFO] Built and pushed image as gcr.io/.../catnotcat-function
[INFO]
[INFO]
------------------------------------------------------------------------
...
```

Note: The instruction here took a shortcut to use the latest Jib plugin directly from Maven command line. You can also configure your `pom.xml` to use the Jib plugin with additional configurations.

Task 6 - Enable Egress to Google Cloud Platform APIs

By default, Istio will block all traffic to outside of the cluster. To run this application, you need to configure Istio to allow access to `*.googleapis.com` endpoints in order to call the Vision API. There are 2 different ways to do this:

- Configure an IP range that should be intercepted by Istio. This will require determining the IP range that the Kubernetes cluster uses for Pod IPs. See Knative Network Configuration example.
- Or, add Service Entry to have fine-grained control over which external services you want to allow.

This lab will use the second method.

1. Apply a new Service Entry to Istio so that traffic to `*.googleapis.com` is allowed.

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: googleapis
spec:
  hosts:
  - "*.googleapis.com"
  location: MESH_EXTERNAL
  ports:
  - number: 443
    name: https
    protocol: HTTPS
EOF
```

2. Apply a new Service Entry to Istio so that traffic to `metadata.google.internal` is allowed. This server is used to automatically discover environment configurations such as the GCP Project ID, and also the machine credentials.

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: metadata-server
spec:
  hosts:
  - metadata.google.internal
  - 169.254.169.254
  location: MESH_EXTERNAL
  ports:
  - number: 80
```

```
    name: http
    protocol: HTTP
EOF
```

Task 7 - Deploy the function with service account

1. Deploy the container using riff CLI.

```
$ export GCP_PROJECT=$(gcloud config get-value core/project)
$ riff service create catnotcat \
  --image gcr.io/${GCP_PROJECT}/catnotcat-function
```

Note: Your GKE cluster was created with the `cloud-platform` scope. This means that any process running inside of this cluster may have access to all of the Google Cloud Platform APIs. This can be further restricted by the GCP service account associated with the GKE cluster.

However, in a production environment, your cluster is likely restricted in terms of the cluster scope. In which case, you may need to start the process with a GCP service account. This can be done via the environmental variable, e.g.:

```
  riff service create catnotcat \
    --env SPRING_CLOUD_GCP_PROJECT_ID=${GCP_PROJECT} \
    --env SPRING_CLOUD_GCP_CREDENTIALS_ENCODED_KEY="$(base64 -w0 \
       $HOME/catnotcat.json)" \
    --image gcr.io/${GCP_PROJECT}/catnotcat-function
```

Or, store the credential in Kubernetes Secret and use the secret value as an environmental variable.

2. Validate that the function was successfully deployed.

```
$ kubectl get deployment
catnotcat-00001-deployment   1         1         1            1           19s
uppercase-00001-deployment   0         0         0            0           1h
```

3. Invoke the function.

```
$ cd ~/catnotcat
$ base64 -w0 cat.jpg | riff service invoke catnotcat --text -- -d@- -w '\n'
cat
```

4. Riff generated a Knative service resource in Kubernetes. You can see what riff generated:

```
$ riff service create catnotcat \
  --dry-run \
  --image gcr.io/${GCP_PROJECT}/catnotcat-function
```

In fact, you can deploy the same YAML to another Knative cluster that doesn't have riff installed!

Lab 6 - Configure Domain Name

Duration: 20:00

The default domain name is `example.com`. The fully qualified domain name is constructed based on: `${service_name}.${namespace}.${domain}`. You can change the default domain name by updating a Knative configuration.

1. See the default configuration.

```
$ kubectl -n knative-serving get cm config-domain -oyaml
apiVersion: v1
data:
  example.com: ""   ← This is where the default domain name is configured.
kind: ConfigMap
...
```

2. Construct the nip.io address based on your Ingress IP. Nip.io is a "Dead simple wildcard DNS for any IP Address". E.g., if you have IP address 1.2.3.4, you can use the domain name 1.2.3.4.nip.io, or any subdomain of that - they will all resolve to the IP address.

```
$ export INGRESS_IP=`kubectl get svc istio-ingressgateway -n istio-system \
  -o jsonpath="{.status.loadBalancer.ingress[*].ip}"`
$ export NIP_DOMAIN="${INGRESS_IP}.nip.io"
$ echo $NIP_DOMAIN
```

3. Update the default configuration to use `nip.io`.

```
$ cat << EOF | kubectl apply -n knative-serving -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-domain
data:
  ${NIP_DOMAIN}: ""
EOF
```

4. Check that the domain name has been updated. Wait until that your `nip.io` domain name has been updated.

```
$ watch kubectl get ksvc catnotcat \
  --namespace=default \
  --output=jsonpath="{.status.domain}"
```

5. Exit the watch loop (Ctrl+C)

6. Make a request via the new domain name.

```
$ cd ~/catnotcat
$ base64 -w0 dog.jpg | curl -d@- -H"Content-Type: text/plain" -w"\n" \
  http://catnotcat.default.${NIP_DOMAIN}
not cat
```

Lab 6 - Blue / Green Deployment

1. Check out Knative demo repository.

```
$ cd ~/
$ git clone https://github.com/saturnism/knative-demos.git
$ cd knative-demos
```

The original files were written by Mark Chmarny. Ray customized some of the files to show Spring Boot and Java applications running in Knative.

2. Deploy the Blue version.

```
$ kubectl apply -f kubectl apply -f blue-green-deploy/stage1.yaml
```

3. Get the URL.

```
$ echo "http://bg.default.${NIP_DOMAIN}"
```

4. Open the URL in the browser.

5. Deploy the Green version, but keep traffic to Blue version.

```
$ kubectl apply -f blue-green-deploy/stage2.yaml
```

6. Go back to the URL and refresh, you'll see that the traffic is still routed to the Blue version.

7. However, you can check the new version with a special URL:

```
$ echo "http://v2.bg.default.${NIP_DOMAIN}"
```

8. Go to that URL, and see the new version before you shift all the traffic over.

9. Ramp up the traffic to 50/50.

```
$ kubectl apply -f blue-green-deploy/stage3.yaml
```

10. Reload the page a couple of times to see the traffic splitting.

```
$ curl http://bg.default.${NIP_DOMAIN}     # do this a few times!
```

11. Ramp up the Green version traffic to 100%.

```
$ kubectl apply -f blue-green-deploy/stage4.yaml
```

Lab 8 - Deploy to Cloud Run

1. Enable Cloud Run API

```
$ gcloud services enable run.googleapis.com
```

2. Deploy the same image to Cloud Run

```
$ gcloud beta run deploy catnotcat \
  --image=gcr.io/${GCP_PROJECT}/catnotcat-function \
  --allow-unauthenticated \
  --memory=512Mi

Deploying container to Cloud Run service [catnotcat] in project [...] region
[us-central1]
✓ Deploying new service... Done.
  ✓ Creating Revision...
  ✓ Routing traffic...
  ✓ Setting IAM Policy...
Done.
Service [catnotcat] revision [catnotcat-00001] has been deployed and is
serving traffic at https://catnotcat-....a.run.app
```

3. Get the URL and make a request!

```
$ export CATNOTCAT_URL=$(gcloud beta run services describe catnotcat \
    --format="value(status.domain)")
$ echo $CATNOTCAT_URL
```

4. Make a request!

```
$ cd ~/catnotcat
$ base64 -w0 cat.jpg | curl -XPOST -H"Content-Type: text/plain" \
    -d@- -w '\n' $CATNOTCAT_URL
```

5. Check out the Knative configuration used by Cloud Run

```
$ gcloud beta run services describe catnotcat
```

In the previous labs, you manually installed Knative on GKE using riff. Cloud Run not only can run as a managed service, but it can also be installed on GKE cluster. Essentially, Cloud Run on GKE is a GKE cluster with Knative pre-installed. See Cloud Run on GKE documentation.

~~Lab 7 - Create an Eventing Bus~~

~~Duration: 10:00~~

~~In the next lab, we'll ultimately connect the `catnotcat` function and `uppercase` function together via a channel over an event bus. So, if you send a picture into `catnotcat` function, the result string will be passed to the `uppercase` function. This lab will configure the event bus.~~

~~Task 1 - Enable Pub/Sub API~~

~~1. Enable Pub/Sub API~~

```
$ gcloud services enable pubsub.googleapis.com
Waiting for async operation operations/... to complete...
Operation finished successfully. The following command can describe the
Operation details:
 gcloud services operations describe operations/...
```

~~Task 2 - Create a Pub/Sub Service Account~~

~~1. Create a new service account that can push to Google Container Registry.~~

```
$ gcloud iam service-accounts create pubsub-bus
```

2. Assign IAM permission to the service account.

```
$ export GCP_PROJECT=$(gcloud config get-value core/project)
$ gcloud projects add-iam-policy-binding $GCP_PROJECT \
  --member serviceAccount:pubsub-bus@$GCP_PROJECT.iam.gserviceaccount.com \
  --role roles/pubsub.editor
```

3. Create a JSON key.

```
$ gcloud iam service-accounts keys create \
  --iam-account "pubsub-bus@$GCP_PROJECT.iam.gserviceaccount.com" \
  ~/pubsub-bus.json
```

Warning: Treat the service account JSON key file as your own username/password. Do not share this in the public!

Task 3 - Configure Bus

Duration: 30:00

1. Store the Service Account as a Secret

```
$ kubectl create secret generic gcppubsub-bus-key \
  -n knative-eventing \
  --from-file=key.json=$HOME/pubsub-bus.json
```

2. Configure Knative Cloud Pub/Sub Event Bus to use your project so that it can create Cloud Pub/Sub topics and subscriptions in the project.

```
$ export GCP_PROJECT=$(gcloud config get-value core/project)
$ kubectl create configmap gcppubsub-bus-config \
  -n knative-eventing \
  --from-literal=GOOGLE_CLOUD_PROJECT=$GCP_PROJECT
```

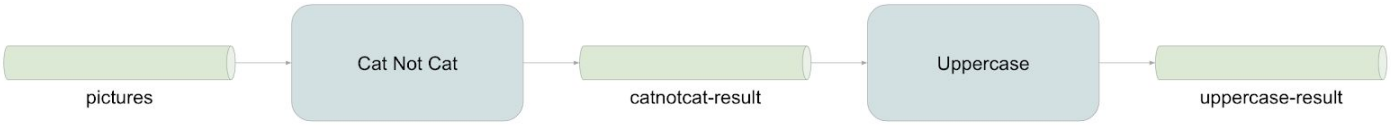
3. Install the Pub/Sub Bus.

```
$ kubectl apply -f \
https://github.com/knative/eventing-sources/releases/download/v0.5.0/gcppubsub.yaml
```

Lab 7 - Uppercat Lite

Duration: 20:00

In this lab, you'll connect the `catnotcat` function and `uppercase` function together via a channel over an event bus. So, if you send a picture into `catnotcat` function, the result string will be passed to the `uppercase` function.



Task 1 - Create Input and Output Channels

1. Create channels.

```
$ riff channel create pictures --cluster-bus gcppubsub
$ riff channel create catnotcat-result --cluster-bus gcppubsub
$ riff channel create uppercase-result --cluster-bus gcppubsub
```

2. Behind the scenes, Knative Eventing's Cloud Pub/Sub integration will automatically create Cloud Pub/Sub topics. List all of the Pub/Sub topics automatically provisioned by the Knative Eventing Bus.

```
$ gcloud pubsub topics list --format="value(name)"
projects/.../topics/channel-gcppubsub-default-pictures
projects/.../topics/channel-gcppubsub-default-catnotcat-result
projects/.../topics/channel-gcppubsub-default-uppercase-result
```

3. Tie up the channels with the services.

```
$ riff subscription create --channel pictures \
  --subscriber catnotcat --reply-to catnotcat-result
$ riff subscription create --channel catnotcat-result \
  --subscriber uppercase --reply-to uppercase-result
```

4. Behind the scenes, Knative Eventing will also automatically create subscriptions to the topics. List all of the Pub/Sub subscriptions automatically provisioned by the Knative Eventing Bus.

```
$ gcloud pubsub subscriptions list --format="value(name)"
projects/.../subscriptions/subscription-gcppubsub-default-catnotcat
projects/.../subscriptions/subscription-gcppubsub-default-uppercase
```

5. To get the results from `uppercase-result` channel, create a new subscription so you can subscribe to validate the output. This is done so that you can quickly validate messages are flowing through before we move to the next lab.

```
$ gcloud pubsub subscriptions create uppercase-result-sub \
  --topic=channel-gcppubsub-default-uppercase-result
```

Warning: You typically shouldn't access/consume Knative-managed topics. The lab created the subscription here so that you can quickly understand what's happening behind the scenes and peek into the data to make sure everything is working before deploying a fully connected examples.

Task 5 - Publish to the Channel

1. To quickly test that your event flows are correct, publish a picture to the topic.

```
$ cd ~/catnotcat
$ gcloud pubsub topics publish channel-gcppubsub-default-pictures \
  --attribute "Content-Type=text/plain" --message="$(base64 -w0 cat.jpg)"
```

Warning: Similar to the previous task, you typically shouldn't access/consume Knative-managed topics. This task is publishing direct a topic so that you can quickly understand what's happening behind the scenes and make sure everything is working before deploying a fully connected examples.

2. Observe that a `catnotcat` function instance is running. If it was previously scaled to 0, a new instance should be starting.

```
$ kubectl get pod
NAME                                    READY     STATUS    RESTARTS   AGE
catnotcat-00001-deployment-...   3/3       Running   0          5m
```

3. When the instance status is RUNNING, you can tail the log and you should observe that it processed one input and printed the response output.

If there is only one `catnotcat-00001-*` pod instance, then you can use a shortcut command to see the logs.

```
$ kubectl logs -f -c user-container \
  $(kubectl get pod -l app=catnotcat-00001 -oname)
...
responses {
  label_annotations {
    mid: "/m/01yrx"
    description: "cat"
```

```
    score: 0.98402584
    topicality: 0.98402584
  }
...
```

Otherwise, you'll need to view each instance's log separately by providing the specific Pod name.

```
$ kubectl logs -f catnotcat-00001-deployment-... -c user-container
...
responses {
  label_annotations {
    mid: "/m/01yrx"
    description: "cat"
    score: 0.98402584
    topicality: 0.98402584
  }
...
```

Note: In a production deployment, you may have multiple instances of the application/function running. Tailing each pod individually is not viable.

There are other command line utilities that you can use to tail the logs for a group of pods (e.g., all `catnotcat` pod instances). For example: kail, stern, kubetail, etc.

In Google Cloud Platform, all of the logs are automatically streamed to and stored in Stackdriver Logging. You can use Stackdriver Logging console to browse/search/export logs, and even alert on them.

4. Observe that an `uppercase` function instance is starting/running too.

```
$ kubectl get pod
NAME                                    READY     STATUS    RESTARTS   AGE
catnotcat-00001-deployment-...   3/3       Running   0          5m
uppercase-00001-deployment-...   3/3       Running   0          5m
```

5. Inspect the `uppercase-result` channel to see the result. The channel is mapped to a Cloud Pub/Sub topic. You created a subscription to the topic earlier. Pull a message from the subscription to see the result.

```
$ gcloud pubsub subscriptions pull uppercase-result-sub --auto-ack
┌──────┬─────────────────┬──────────────────────────────────────────────────────────────────────────┐
│ DATA │    MESSAGE_ID   │                                ATTRIBUTES                                │
├──────┼─────────────────┼──────────────────────────────────────────────────────────────────────────┤
│ CAT  │ ...             │ Content-Type=text/plain;charset=UTF-8                                    │
│      │                 │ X-B3-Sampled=1                                                           │
│      │                 │ X-B3-Spanid=3f482930bbeba988                                             │
│      │                 │ X-B3-Traceid=3f482930bbeba988                                            │
│      │                 │ X-Request-Id=b1c16a46-38da-9ed1-abcc-a5a6fef7610d                        │
```

```
└──────┴──────────────────┴──────────────────────────────────────────────────────────┘
```

6. Try the same thing with the dog picture!

```
$ cd ~/catnotcat
$ gcloud pubsub topics publish channel-gcppubsub-default-pictures \
  --attribute "Content-Type=text/plain" --message="$(base64 -w0 dog.jpg)"
```

Lab 8 - Uppercat End-to-End

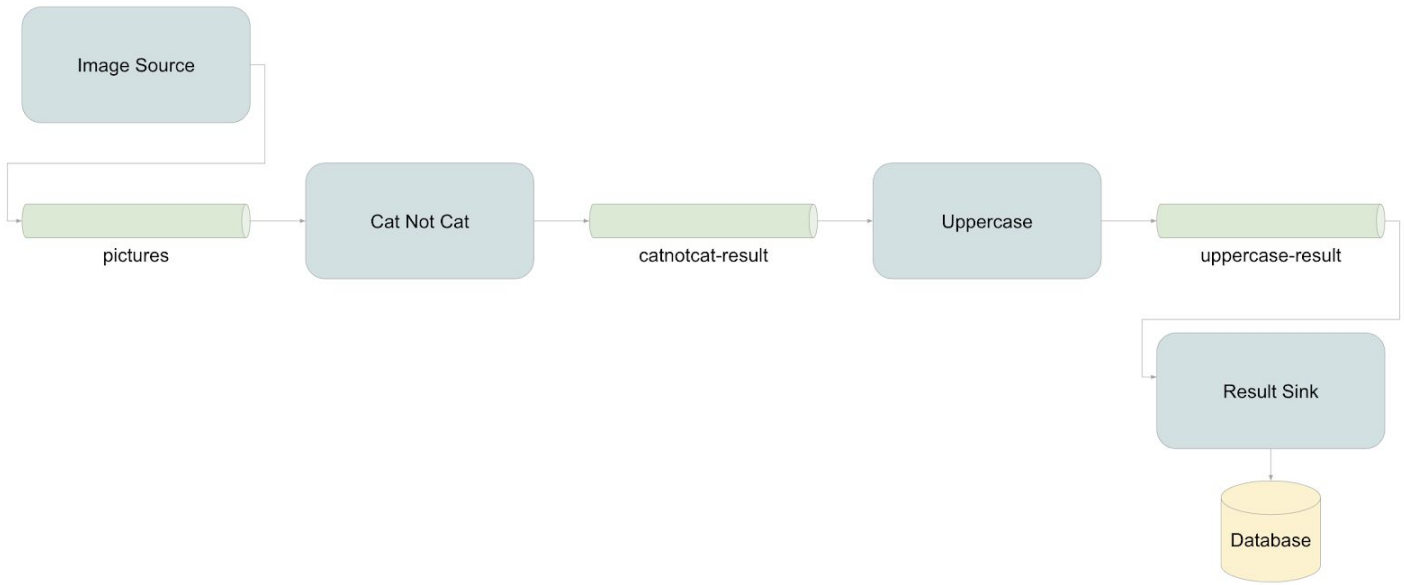
Duration: 20:00

So far, you've learned:

1. Creating a Kubernetes Cluster
2. Installing riff, Knative, and Istio
3. Deploying Function from source
4. Deploying Function from a pre-built container.
5. Creating Spring Cloud Functions from scratch.
6. Using Google Cloud Platform APIs with Spring Cloud GCP.
7. Configuring a Knative event bus.
8. Tying functions together using channels over the event bus.

Let's tie up the example application end to end, so that you can:

1. Send in a URL of a picture to an image source function that'll extract the content of the image from the URL.
2. The content of the image is then sent to the catnotcat function via a channel.
3. The result is uppercased.
4. The uppercased result is stored into a database.



Task 1 - Image Source Function

The Image Source Function is located here:
https://github.com/trisberg/s1p2018-serverless/tree/master/image-source

1. Examine the function source code.

It simply takes in an image URL and then posts the base64 encoded content to the channel.

2. It takes advantage of message headers to propagate the name of the image via the header named `ce-image-name`.

3. Deploy this function using a prebuilt container.

```
$ riff service create imagesource --image=trisberg/imagesource-function
```

4. Once deployed, give it a try!

```
$ riff service invoke imagesource --text -- -w"\n" \
-d"https://upload.wikimedia.org/wikipedia/commons/thumb/4/4f/Felis_silvestris_catus_lying_on_rice_straw.jpg/320px-Felis_silvestris_catus_lying_on_rice_straw.jpg
"
Publishing: 320px-Felis_silvestris_catus_lying_on_rice_straw.jpg [39776]
```

Task 2 - Results Sink

The Results Sink Function is located here:
https://github.com/trisberg/s1p2018-serverless/tree/master/results-sink

1. Examine the pom.xml.

This function connects to fully managed Cloud Spanner instance using the Spring Cloud GCP's Data Spanner Starter.

2. Examine the function source code.

It simply takes in a Message that has the payload type of String (which will be "cat" or "not cat"), and inserts the string into a database table, with the associated image name extracted from the Message header. Since this component does not produce any output it is implemented as a `java.util.function.Consumer`.

3. Examine the application.properties.

This configures the instance and the name of the database the application will connect to.

4. Create a new Cloud Spanner instance.

```
$ gcloud services enable spanner.googleapis.com
$ gcloud spanner instances create workshop --config=regional-us-central1 \
  --nodes=1 --description="SpringOne Platform 2018"
```

5. Create a new database within the Cloud Spanner instance.

```
$ gcloud spanner databases create catnotcat --instance=workshop
```

6. Create the schema file that will create the database table to store the results.

```
$ cat <<EOF > schema.ddl
CREATE TABLE results (
  id          STRING (36) NOT NULL,
  name        STRING (255) NOT NULL,
  catnotcat STRING (10) NOT NULL
) PRIMARY KEY (id );
EOF
```

7. Create the tables in Spanner using the schema file.

```
$ gcloud spanner databases ddl update catnotcat \
  --instance=workshop --ddl="$(cat schema.ddl)"
```

8. Deploy the function. You also need to configure an environmental variable to provide your Project ID to reference to your Cloud Spanner instance.

```
$ export GCP_PROJECT=$(gcloud config get-value core/project)
$ riff service create resultssink \
  --env SPRING_CLOUD_GCP_PROJECT_ID=${GCP_PROJECT} \
  --image=trisberg/resultssink-function
```

9. Subscribe the function to listen to the `uppercase-result` channel.

```
$ riff subscription create --channel uppercase-result \
  --subscriber resultssink
```

Task 3 - Try It Out!

1. Send a URL to the Image Source function.

```
$ riff service invoke imagesource --text --- -w"\n" \
```

```
-d"https://upload.wikimedia.org/wikipedia/commons/thumb/4/4f/Felis_silvestris_catus_lying_on_rice_straw.jpg/320px-Felis_silvestris_catus_lying_on_rice_straw.jpg"
Publishing: 320px-Felis_silvestris_catus_lying_on_rice_straw.jpg [39776]
```

2. Wait for the function pipeline to execute. Observe that individual functions are scaled to at least one instance.

```
$ kubectl get pods
NAME                                    READY     STATUS    RESTARTS   AGE
catnotcat-00001-deployment-...          3/3       Running   0          7s
imagesource-00001-deployment-...        3/3       Running   0          46s
results-sink-00001-deployment-...       3/3       Running   0          2m
uppercase-00001-deployment-...          3/3       Running   0          35s
```

3. When everything's finished, connect to the Cloud Spanner instance to read the results.

```
$ gcloud spanner databases execute-sql --instance workshop catnotcat \
  --sql "select * from results"
```

4. To see the current subscriptions and what channels they listen to run the following command:

```
$ riff subscription list
NAME         CHANNEL           SUBSCRIBER   REPLY-TO
catnotcat    pictures          catnotcat    catnotcat-result-channel
resultssink  uppercase-result  resultssink
uppercase    catnotcat-result  uppercase    uppercase-result-channel
```

5. Try it with a dog picture.

```
$ riff service invoke imagesource --text -- -w"\n" \
-d"https://upload.wikimedia.org/wikipedia/commons/thumb/b/b5/Golden_Retriever_medium-to-light-coat.jpg/320px-Golden_Retriever_medium-to-light-coat.jpg"
Publishing: 320px-Golden_Retriever_medium-to-light-coat.jpg [64836]
```

Lab 10 - Autoscaling

Serverless platform is designed to automatically scale out when you are receiving more requests/traffic to your function.

1. See that there is 0 or only 1 instances of uppercase function currently running.

```
$ kubectl get pods -lserving.knative.dev/configuration=uppercase
```

2. Generate some traffic to see the auto scaling capability.

```
$ export INGRESS_IP=`kubectl get svc knative-ingressgateway -n istio-system \
  -o jsonpath="{.status.loadBalancer.ingress[*].ip}"`
$ export NIP_DOMAIN="${INGRESS_IP}.nip.io"
$ docker run -ti --rm saturnism/hey -H "Content-Type: text/plain" \
  -m POST \
  -d "ray" \
  http://uppercase.default.${NIP_DOMAIN}
...
Status code distribution:
  [200] 200 responses
```

Note: hey is a command line utility to generate load.

There are many other load generators you can use. But the popular Apache Bench (`ab`) may not work with Knative. This is because Knative uses Istio underneath, and the underlying HTTP proxy (`envoy`) requires HTTP/1.1 connections. Apache Bench only supports HTTP/1.0.

2. See that there are multiple instances of uppercase function currently running.

```
$ kubectl get pods -lserving.knative.dev/configuration=uppercase
NAME                                                  READY     STATUS    RESTARTS
AGE
uppercase-00001-deployment-5c9cfb5fbc-drfmw   3/3       Running   0
17s
uppercase-00001-deployment-5c9cfb5fbc-hmsqc   3/3       Running   0
17s
uppercase-00001-deployment-5c9cfb5fbc-kh2zx   3/3       Running   0
17s
uppercase-00001-deployment-5c9cfb5fbc-kk6lq   3/3       Running   0
17s
uppercase-00001-deployment-5c9cfb5fbc-klxbr   3/3       Running   0
5m
uppercase-00001-deployment-5c9cfb5fbc-njxdr   3/3       Running   0
17s
uppercase-00001-deployment-5c9cfb5fbc-pks7b   3/3       Running   0
19s
uppercase-00001-deployment-5c9cfb5fbc-px4ht   3/3       Running   0
19s
uppercase-00001-deployment-5c9cfb5fbc-qmt8g   3/3       Running   0
19s
uppercase-00001-deployment-5c9cfb5fbc-vmf2t   3/3       Running   0
19s
```

Lab 11 - Knative native

Duration: 15:00

So far you've deployed Spring Cloud Function using riff which was super easy to do. Since riff is built atop of Knative which is built atop of Kubernetes - it means you can:

- Deploy any Kubernetes workload into the cluster.
- Deploy any Knative applications into the cluster

In this lab, you'll deploy a Spring Boot application using Knative build template.

Taks 1 - Jib Build Template

There are many different Knative build templates that you can use to build from source to URL (i.e., build source, build container, and deployed into Kubernetes cluster so that you can access via a URL!).

You've already learned to use Jib to build an image from existing Java application. You can use the Jib build template and build/deploy from source.

1. Install Jib build template

```
$ kubectl apply -f \
https://raw.githubusercontent.com/knative/build-templates/master/jib/jib-maven.yaml
```

2. Java applications tend to download quite a bit of dependencies. Jib build template supports local Maven caches. Create a Persistent Volume Claim to store the Maven cache.

```
$ cat <<EOF | kubectl apply -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: build-cache-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
EOF
```

Task 2 - Deploy an Existing Java Application from Source

Let's use an existing Spring Cloud GCP's Vision sample.

1. Deploy a Knative service that will build from source using the Jib build template.

```
$ export GCP_PROJECT=$(gcloud config get-value core/project)
$ cat <<EOF | kubectl apply -f -
apiVersion: serving.knative.dev/v1alpha1
kind: Service
```

```
metadata:
  name: vision
  namespace: default
spec:
  runLatest:
    configuration:
      build:
        # Use riff-build service account that has the push credential to GCR.
        serviceAccountName: riff-build

        # Set a build timeout duration of 10 minutes
        timeout: 10m

        # Clone the repository and checkout the branch/tag.
        source:
          git:
            url: https://github.com/spring-cloud/spring-cloud-gcp.git
            revision: v1.0.0.RELEASE

        # Use Jib Maven build template that you installed early.
        # Then configure the image name, use a Maven cache, and the submodule
        # to build.
        template:
          name: jib-maven
          arguments:
          - name: IMAGE
            value: &image gcr.io/${GCP_PROJECT}/vision-sample
          - name: CACHE
            value: build-cache
          - name: DIRECTORY
            value: spring-cloud-gcp-samples/spring-cloud-gcp-vision-api-sample
        volumes:
          - name: build-cache
            persistentVolumeClaim:
              claimName: build-cache-claim
      revisionTemplate:
        metadata:
          labels:
            knative.dev/type: app
        spec:
          container:
            image: *image
EOF
```

Note: You can and should store these YAML configurations in a file and versioned controlled for your production application.

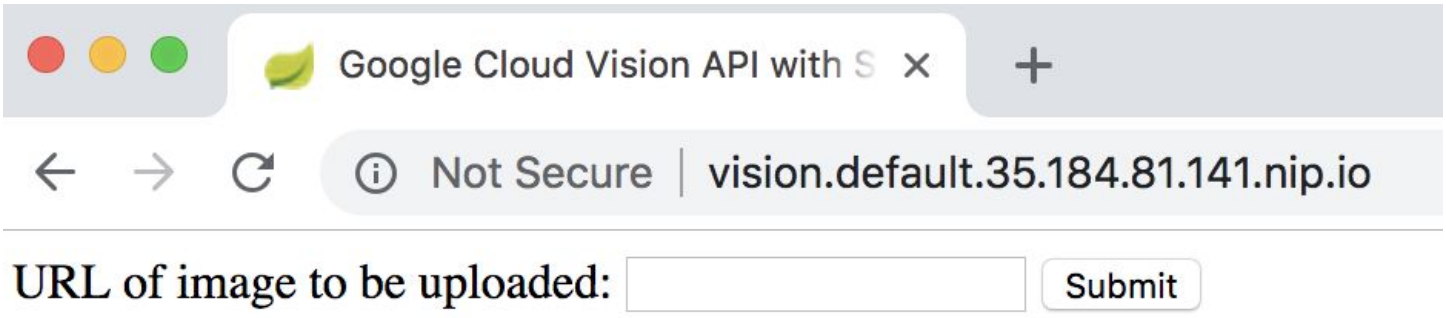
2. This will kick off a Knative build that will automatically checkout the source, build the application, and containerize with Jib.

```
$ watch kubectl get pods
```

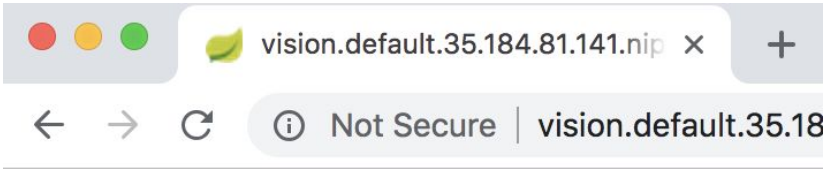
3. Assuming you've completed Configure Domain Name lab, you are able to access this application directly from the browser. Find the domain name to this application.

```
$ kubectl get services.serving/vision \
  --namespace=default \
  --output=jsonpath="{.status.domain}" && echo
vision.default...nip.io
```

4. In a browser, navigate to the domain name, you should see the application:



4. Find an image URL and see what it returns! E.g., the Cat picture you've been using:
https://upload.wikimedia.org/wikipedia/commons/thumb/4/4f/Felis_silvestris_catus_lying_on_rice_straw.jpg/320px-Felis_silvestris_catus_lying_on_rice_straw.jpg



description	score
cat	0.98402584
fauna	0.93889964
mammal	0.924809
small to medium sized cats	0.9135055
cat like mammal	0.85424644
wildlife	0.8493201
whiskers	0.81929225
wild cat	0.8057926
terrestrial animal	0.7708989
organism	0.6898634

~~Lab 12 - Clean Up~~

~~Before you leave, please delete all the resources that you created:~~

- ~~Kubernetes~~
- ~~Spanner~~

```
~~$ riff system uninstall --force~~
~~$ gcloud spanner instances delete workshop -q~~
~~$ gcloud container clusters delete riff-cluster -q~~
```