Waterford Institute *of* Technology

Cloud Computing 2

Bachelor of Science (Hons) Applied Computing

Knative Deployments

Ciaran Roche - 20037160

April 16, 2019

Plagiarism Declaration

Unless otherwise stated all the work contained in this report is my own. I also state that I have not submitted this work for any other course of study leading to an academic award.

Contents

1 Introduction

Modern cloud centric computing has seen a number of technologies and approaches emerge as candidates for auto inclusion in development stacks. In particular, Kubernetes and developing for a serverless world are fast becoming the de facto way of working as their benefits are clear. Simple, yet clever configurations can bring unrivalled scalability, coupled with huge cost saving over the lifecycle of a service. These are among the primary drivers for attracting developers towards this paradigm. Through this trend a number of Frameworks and Models have emerged, which allow developers build modern applications. In paticular and the focus of this report is Knative.
Knative is comprised of middleware components which are essential to build modern, source-centric, and container-based applications. By abstracting away the "boring" bits Knative frees developers to focus on writing interesting code.

1.1 Outline

This paper will be broken into two sections, Theory and Practical. Throughout the Theory section we will answer numerous common questions around Knative, gaining an understanding of the framework. Finally in the Practical section we will get our hands dirty with Knative, by exploring their documentation, installing and deploying example applications and digging deeper into what Knative have to offer. We will finish with taking the Knative approach to building a modern serverless web application. The paper will be wrapped up with a Summary and my thoughts on Knative as a solution to serverless deployments within Kubernetes.

2 Theory

This section will look to answer a number of questions so that we can gain a greater understanding of what Knative is and how it aids a developer working with Kubernetes. It should also be noted that since this technology is only a year old, at time of writing this paper, the amount of resources on Knative are limited, so with that I have decided to list the resources I used during research of this term paper in an Appendix, over using a Bibliography, while answering the questions below based on my findings and what I have learned from my time working with Knative.

2.1 What is Knative?

As mentioned above Knative is a new project in the Kubernetes landscape, it is an open source platform that is being developed by engineers from Google, IBM, Red Hat, SAP and many other major players.
Knative follows a mantra *"boring but difficult"*, in that the frame work is a solution to the boring and difficult aspects to modern application development. So if we refer to Appendix A we can clearly see this mantra littered across the frame works's promotional website.
It tells us that Knative is a set of middleware components that are essential to building modern applications. In a sense it abstracts the *"boring"* bits from the developer, freeing them to focus their energy on writing the business logic for their application. If we dig a little deeper we can get a number of examples of these *"boring"* bits, such as orchestrating source-to-container workflows, routing and managing traffic during deployment, auto-scaling the workloads and also binding running services to eventing ecosystems. All this while being interoperable with many idioms, languages and frameworks.
To avoid this becoming very sales-pitchy, we shall dive a bit deeper and break down the components that make up Knative.

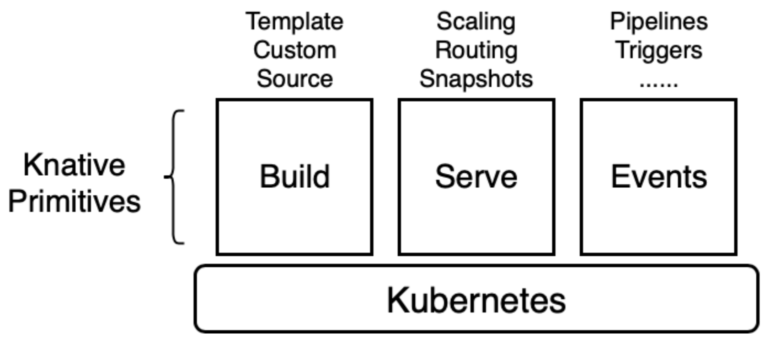

Figure 1: *Knative Overview*

If we refer to Figure 1, it paints a clear picture of what Knative is and how it looks. It is essentially a platform that sits on top of Kubernetes and brings the capabilities of serverless workloads to Kubernetes. It is made up of a number of utilities, that try to make working with cloud native apps feel, native. These utilities/components, or as they are referred to in the documentation, Knative Primitaves, are as follows – Build, Serve and Event

2.1.1 Build

Figure 2 tries to paint a picture of the process in which we would go through in deploying our code in Kubernetes. As we can see we start out with our code for the application, which can be hosted on GitHub or some other repository. Our next step is to take our code and turn it into an image, something in which Kubernetes, Docker or what ever container runtime we wish to use can recognise. This is often a simple Docker Build, but for more complex application a number of different steps may be needed. The end result will always be the same in that we take our code and turn it into a image. Once we have built our image the next step is to push this image to the cloud, so that it is accessable, in a repository or hub of our choice. The final step which we need is a manifest yaml file so that we can deploy our image to Kubernetes. In some cases these yaml files can stretch far beyond a single file depending on the complexity of the application.

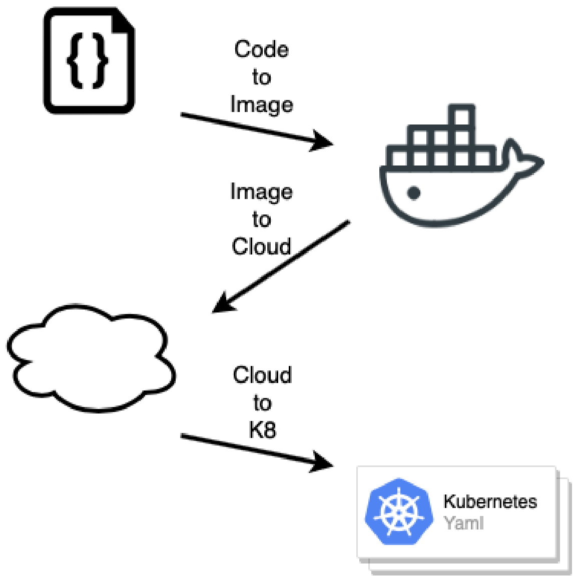

Figure 2: *Knative Build*

Looking at the steps above, and if we accept the fact we are working in an iterative process, pushing and deploying changes regularly, the whole process can become quite tedious. This is where Knative Build comes in. It takes the entire process and brings it directly into our Kubernetes Cluster,

handling the source code management. As well as custom , complex or standard builds. Or if we prefer we can use templates. An example of a template would be Cloud Foundry Build Packs. Diving into build packs is out of the scope of this paper but to give a high level view understanding, they basically examine our application to determine what dependencies are needed and the auto configure the app to communicate with bound services. On push to Cloud Foundry a build pack is applied, the app compiled and ready for launch.
Knative takes all these steps, and automates them through a single manifest file deploy, this makes it a lot easier and I guess more agile, bringing faster development times by abstracting away the steps needed to bring our code from source to deployment within out cluster.
It is also worth nothing that this component is a work-in-progress, as of writing this report, Knative build does not provide a complete standalone CI/CD solution. What it does provide is a lower-level building block that is designed to enable integration and utilization into larger systems.
Under the hood, Knative build is a custom resource. It is here that we can define a process and run it to completion. For example, our custom resource can fetch, build and package our code and respond back if the process has succeeded. All this runs on-cluster and implemented by a Kubernetes Custom Resource Definition.
Like Build Packs, Kubernetes Custom Resource Definitions (CRD) are out of scope to this term paper, but a high level understanding of a CRD is an extension to the Kubernetes API that is not available in a default Kubernetes installation. We are able to spin up and tear down CRD's through dynamic registration within our clusters and we can interact with CRD's as we do with a regular Pod through kubectl.

2.1.2 Serve

The serve component to Knative comes with Istio components built in. Istio will be discussed in Section 2.4. So if we look at Figure 3, we get a snapshot of an average serve looks like. Serve deals with our service or application which we want to deploy in our cluster. This can be a traditional micro service or a simple function. The service is then pointing to two different things – a route and a config.
Now one of the cool aspects to serve is when we do a push of our service to our cluster, our config within the serve stores that version. So if we refer back to Figure 3 we can see we have two different versions of our service served to our cluster. These two versions are managed by the config.
Route on the other hand manages the traffic to our service, and we can route traffic to one or more of our version stored under the config. This is utilized by the Istio traffic management capabilities. So if we look at Figure 3 we can see we have 10% of our traffic directed to version 2 and the rest being forwarded to version 1. This allows for simple staged roll outs of services, or even AB/Canary testing. AB testing is a method that shows a small number of our users a newer version of the service, this allows us to see if the latest version of the service has a positive impact on UX.

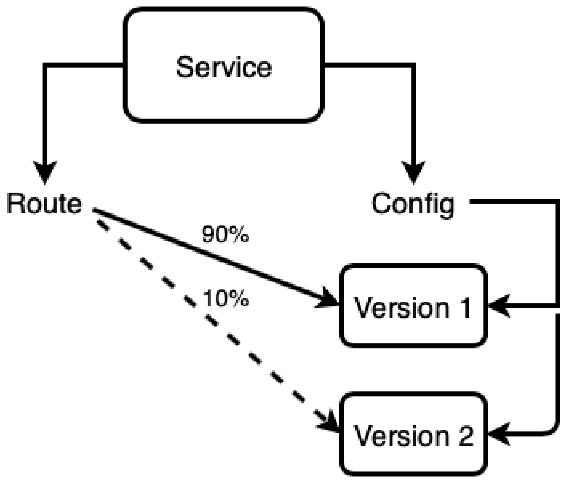

Figure 3: *Knative Serve*

Lets break down the resources that make up a serving within Knative. Like the build, Knative handles a serving as a Kubernetes Custom Resource Definition. The service is the workhorse and manages the whole lifecycle of our application. It handles the creation of other objects to ensure that the app has a route and a configuration along with a new revision for the latest version or update of the service.
The route resource maps a network endpoint to one or more revisions. By utilizing Istio the route can manage the traffic in several ways, for instance fractional traffic and named routes.
The configuration resource maintains the desired state of our application. It provides a separation between our code and our configuration. Any modification to the code and configuration results in a new revision number.
The revision resource is a snapshot of our applications code and configuration. These revisions are immutable objects and can be retained for as long as we need.

2.1.3 Event

So before we begin on eventing it is worth noting that this is a heavy work in progress but they are a number of capabilities which we will discuss below. So I feel it best to start with an example scenario to try paint a picture of the type of use case in which eventing could be used. So eventing in my opinion is the bread and butter of any serverless platform, in that you want to create some sort event which will trigger a serverless function. So lets take for example if you have a webhook on your GitHub repo, and you want to trigger a service which in turn will execute a Knative Build on your latest code. This can be handled through either the use of triggers within in Knative Eventing or else through a pipeline configured with Knative Eventing.
Figure 4 shows the above example of a Knative Data Plane Forward Event. If we break it down

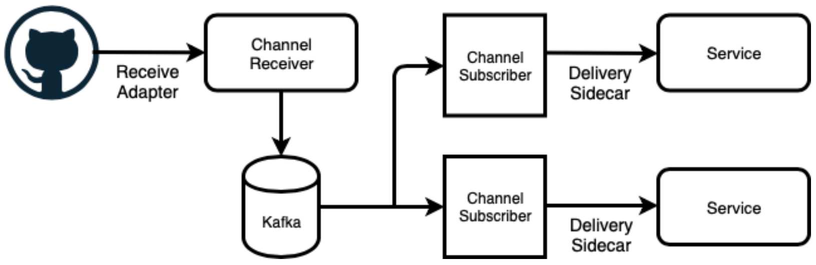

Figure 4: *Knative Data Plane Forward Eventing*

the GitHub repo is outside the pod boundary. The channel receiver is accepting CloudEvents over HTTP POST, feeding it to a Kafka pipeline which is a protocol specific implementation for our services.
The objecs that make up a Knative Event are – consumers, channels and subscriptions, and finally sources.
Event Consumers
We can think of consumers as our services into which we want events to trigger. To allow for the delivery to our services Knative defines two generic interfaces. These interfaces are implemented as a Kubernetes resource. The first being *addressable* – these objects are able to receive and acknowledge an event delivered over HTTP to an address defined in their hostname field. The second interface is *callable* – which are objects that are able to receive an event delivered over HTTP and transform the event, and returning either 0 or 1 new events in the HTTP response. These responses can be further processed in the same way that the original event from the external source is processed.
Event Channels and Subscriptions
We can define a single event forwarding and persistence layer. This is known as a Channel within Knative eventing. These events are forwarded to our services or to other channels using subscriptions. This allows message delivery in our cluster to vary based on our requirements. To put into other words we can handle some events by an in-memory implementation, event channels and subscriptions, or we can be persisted an use a stream processing software such as Kafka.
Event Sources
We can think of a source as the creation of an event, each source is a separate Kubernetes custom resource. Which allows us to define the arguments and parameters needed to instantiate it. It seems to be common practice to define a source type in golang format but they can also be expressed in lists, such as YAML or JSON.

2.2 Who is Knative designed for?

The Kubernetes landscape has a varied audience, in that it is littered with people from end users to developers and everyone in between. So who is Knative aimed for?

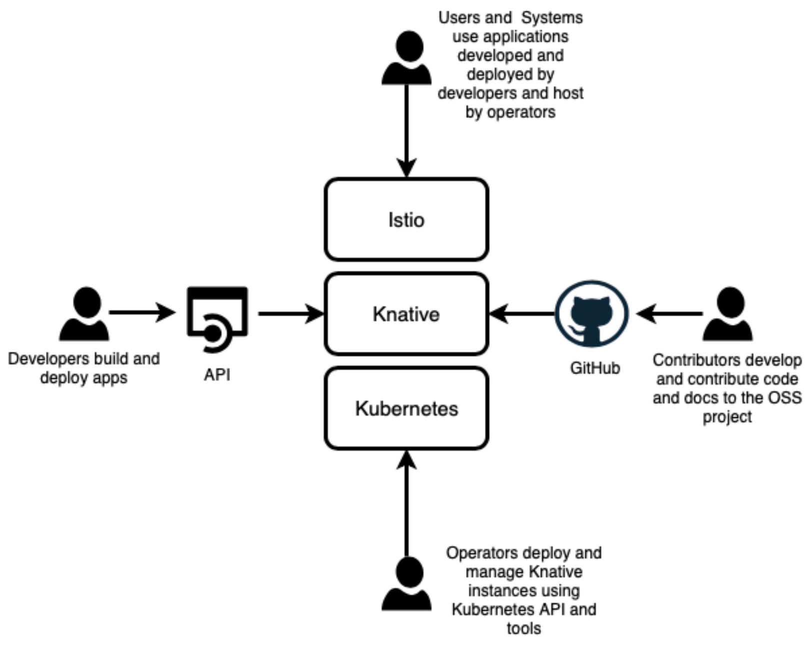
Users and Systems use applications developed and deployed by developers and host by operators
Istio
Developers build and deploy apps
API
Knative
GitHub
Contributors develop and contribute code and docs to the OSS project
Kubernetes
Operators deploy and manage Knative instances using Kubernetes API and tools

Figure 5: *Knative Audience*

Developers – have the ability to deploy serverless-style functions, applications and containers to an auto-scaling runtime through Knative's components which act as Kubernetes-native API's
Operators – Knative components are intended to be integrated into more polished products. It is with this operators can begin to combine the power of Knative primitives and their existing cloud service providers.
Contributors – As Knative is an Open Source Project, it has a clear project scope which allows for an efficient contributor workflow.

2.3 Where can Knative be deployed?

The only dependency that is required to deploy Knative is a Kubernetes cluster. This gives us the freedom to choose the right solution to fit our use case. We have the ability to choose one of the many hosted Kubernetes services available or to even run locally within MiniKube. For this term paper we look a using MiniKube, Google Kubernetes Engine and Amazons EKS. The results can be found in Section 3.2.

2.4 What is Istio?

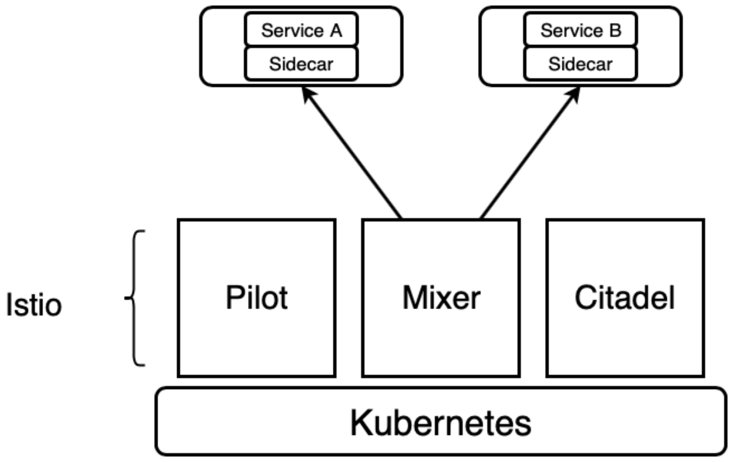

Figure 6: *Istio Overview*

Istio is an open platform independent service mesh, that gives us traffic management, policy enforcement and telemetry collection. So what is a service mesh? If we put it simply it is a network of micro services, and in terms of serverless, we can accept serverless is an extreme case of micro services. If we refer to Figure 6, a service mesh gives Service A the ability to contact Service B, along with giving us routes to each service. As the number of these services grow, so does the complexity of the mesh. This is where Istio comes it, it gives us a way to control connections to Services.
If we have a look at the features of Istio, right away we get out of the box a load balancer, this allows us make connections via the likes of HTTP, TCP and websockets between services as well as bringing traffic from outside the mesh in. The next feature is fine grain control, this allows us to implement rules within our mesh. Giving us the likes of fail overs when a service drops. We also get access control, similar to the fine grain control, but here we enforce the policies which we can set for our application are correct. Finally we get visibility, this gives us logging, stats, graphing and metrics of our mesh.
We get all these features for free from using Istio. Now if we go back to Figure 6 we can have a look at the components of Istio. We can see Istio is made up of – Pilot, Mixer and Citadel. The pilot is the driver of our service mesh, it has the AB testing, it can control the likes of canary deployments. The pilot has the intelligence of how our mesh works. The citadel on the other hand is the security aspect to our service mesh, under the hood it contains a certificate authority. We can route traffic through citadel to ensure traffic within our mesh is encrypted if needs be. This is extremely valuable if we want to extend our Kubernetes cluster to multiple clusters and we extend our service mesh across these clusters, being able to ensure all traffic is encrypted is quite usefull. Finally the mixer is the central point to Istio, it is where the services and the sidecars, along with the Istio components come together. On a side note if we refer to Figure 7 we can get an understanding of what a sidecar

is in terms of microservices. We can deploy components of an application or a service into a separate process or container to provide better isolation and encapsulation. It is refered to as Sidecar as it resembles a sidecar attached to a motorcycle. It is worth noting that a sidecar shares the same lifecycle as its parent service. Back to the mixer and it is here the telemetry from our mesh is built which enables the pilot to build the metrics and produce the graphing of our mesh. The mixer is also plug-able which allows us to build our own components to help manage and control our service mesh.

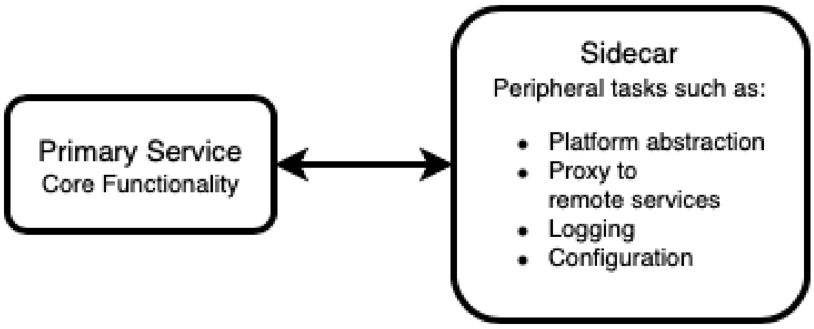

Figure 7: *Sidecar Overview*

So where does Istio fit into Knative, well taking the features mentioned above which we get out of the box from Istio, the likes of traffic management, auto routing and scaling it allows us to abstract these features and wrap them up within our Knative deployments. But something which I find really cool, is Istio gives us the ability to scale to zero. Now this concept allows us to deploy serverless applications within a Kubernetes cluster. So for example if our service is under heavy load we can scale up to as many pods as required, but when no one is accessing that service we can scale back to zero pods.

3 Practical

Now with a greater understanding of what Knative is and how it works, we can get our hands dirty with the framework, furthering our exposure to Knative. We will go through many steps and examples from choosing the right Kubernetes hosting to deploying a Knative app. All which leads to our final product which will be a web application with a serverless backend all deployed to Knative.

3.1 Picking the Right Solution

We are able to run Kubernetes on various platforms, from laptops to bare metal servers and everything inbetween. It was decided to explore Knative on 3 platforms – Minikube, Amazons Elastic Container Service for Kubernetes and Google Kubernetes Engine. Minikube is method for creating a local, single node Kuberneters cluster, designed for local development and testing. Amazons EKS is a managed Kubernetes service, as is Googles Kubernetes Engine.
The steps I followed to instantiate Knative where taking from the Knative documentation found at Appendix B. My first port of call was running Knative within Minikube. I run Minikube as a virtual machine locally using virtual box. When trying to provision Istio, the documentation warn it takes some time, I was finding some of my pods where failing. When I inspected the logs of the failing pods I noticed that I was running into memory issues. This left me puzzeled as on launching MiniKube you pass parameters to increase the amount of memory. After several attempts I kept running into the same error. It turned out virtual box was limiting the resources it was passing to the VM, so from this I bumped my configuration within virtual box. On running MiniKube again I was able to provision all Istio pods, when I applied the Knative manifest to provision the Knative pods my laptop ran out of memory and essentially was rendered useless as I lost control to my laptop. This left me no choice but to look at a hosted managed service.
Here I decided it best to stick with what I was familiar with and try Amazon EKS. Now this left me with no official Knative documentation only the generic how to install Knative on a Kubernetes cluster. So throwing caution to the wind I dived into the Amazon documentation and began provisioning a Kubernetes cluster. The first, I guess road block which I ran into was the documentation, I found them not as descriptive as they needed to be and also to be written, like so many other docs out there, with an assumed understanding of the technology. Despite this I was able to provision a cluster, unfortunately the provisioning time was quite long, over 20 minutes. Now with my cluster provisioned I had to configure my kubectl to be able to communicate with my cluster. Again documentation here was a hurdle I had to overcome, and after some heavy googling I finally have kubectl configured. Now being able to talk to my cluster I began installing Knative, and once again I ran into memory issues. My own fault, so I had to tear everything down and re-provision a cluster this time allocating greater resources to it. This time around the time it took to provision was around

30 minutes. Finally I had successfully installed Knative on my cluster. Deployed a hello world app and everything seemed to work.
While retrospectively looking back at the process of how I got to being able to deploy an app on my cluster I felt the wait time to provision a cluster was too long, as these clusters can be expensive to run I need a solution I can spin up quick and easy as well as tear it down when I don't need it. So from this I decided to explore Google Cloud and their Kubernetes Engine. As GKE is around longer then EKS I felt there might be some improvement here, along with having access to Knative official documentation for running Knative on GKE. Provisioning time was under 10 minutes, and with the Google Cloud SDK being able to sync with kubectl there was no need for configuration here.
Taking into account the provisioning times, the documentation and the support available I decided to opt for using GKE for the rest of this term paper along with other work which I am carrying out for my final year project. In the following Section, I will document the steps needed to install Knative on GKE.

3.2 Installing Knative

The following section will result in a Knative deployment on GKE
Knative requires a Kubernetes cluster v1.11 or newer. 'kubectl' v1.10 is also required. This section will walks us through creating a cluster with the correct specifications for Knative on Google Cloud Platform (GCP).

3.2.1 Installing the Google Cloud SDK and 'kubectl'

We need to have 'gcloud' installed with 'kubectl' version 1.10 or newer To check the version of 'kubectl' we have installed, enter:

```
$ kubectl version
```

We need to download and install the 'gcloud' command line tool which can be done from the following link:

```
https://cloud.google.com/sdk/install
```

With gcloud installed we will need to install kubectl, this can be done as follows:

```
gcloud components install kubectl
```

Now that kubectl is installed we need to authorize our gcloud:

```
$ gcloud auth login
```

3.2.2 Setting environment variables

To simplify the commands, we can define a few environment variables. We need to set 'CLUSTER_NAME' and 'CLUSTER_ZONE' variables to our preference for our deployment. It should be noted that the 'CLUSTER_NAME' needs to be lowercase and unique among any other Kubernetes clusters in our GCP project. The zone can be any compute zone available on GCP. These variables are used later to create a Kubernetes cluster.

```
$ export CLUSTER_NAME=knative
$ export CLUSTER_ZONE=europe-west2-a
```

3.2.3 Setting up a Google Cloud Platform project

We need to have a Google Cloud Platform (GCP) project to create a Google Kubernetes Engine cluster. First we will set a 'PROJECT' environment variable

```
$ export PROJECT=r3x-knative-project
```

Now we can create a project:

```
$ gcloud projects create $PROJECT --set-as-default
```

It should be noted that billing has to be enabled on new projects. GCP locks new projects till you give permission to be billed. With our new project created we need to enable the necessagy APIs which we will use.

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

3.2.4 Creating a Kubernetes cluster

To make sure our cluster is large enough to host all the Knative and Istio components, the recommended configuration for a cluster is:

- Kubernetes version 1.11 or later.
- 4 vCPU nodes ('n1-standard-4').
- Node autoscaling, up to 10 nodes.

- API scopes for ‘cloud-platform‘, ‘logging-write‘, ‘monitoring-write‘, and ‘pubsub‘.

To create a Kubernetes cluster on GKE with the required specifications we use the following command:

```
$ gcloud container clusters create $CLUSTER_NAME \
   --zone=$CLUSTER_ZONE \
   --cluster-version=latest \
   --machine-type=n1-standard-4 \
   --enable-autoscaling --min-nodes=1 --max-nodes=10 \
   --enable-autorepair \
   --scopes=service-control,service-management,compute-rw,storage-ro,cloud-
       platform,logging-write,monitoring-write,pubsub,datastore \
   --num-nodes=3
```

We need to grant cluster-admin permissions to our current user:

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

3.2.5 Installing Istio

As we discussed earlier Knative depends on Istio. To install Istio on our cluster we use the following command:

```
$ kubectl apply --filename https://github.com/knative/serving/releases/download/
    v0.3.0/istio-crds.yaml && \
 kubectl apply --filename https://github.com/knative/serving/releases/download/
    v0.3.0/istio.yaml
```

It goes without saying, we dont just install a random file from the internet onto our cluster, so we need to have a look at those files before running the command. It should also be noted that the resources (CRDs) defined in the ‘istio-crds.yaml‘ file are also included in the ‘istio.yaml‘ file, but they are pulled out so that the CRD definitions are created first. Some times this throws an error when creating resources about an unknown type, in that case we need to run the second ‘kubectl apply‘ command again.
We now need to label the default namespace with ‘istio-injection=enabled‘:

```
$ kubectl label namespace default istio-injection=enabled
```

This can take a few minutes to provision all the pods, we can use the following command to monitor the pods progress:

```
$ kubectl get pods --namespace istio-system --watch
```

3.2.6 Installing Knative

The following commands install all available Knative components as well as the standard set of observability plugins. To install Knative and its dependencies we must run:

```
$ kubectl apply --filename https://github.com/knative/serving/releases/download/
    v0.3.0/serving.yaml \
  --filename https://github.com/knative/build/releases/download/v0.3.0/release.
     yaml \
  --filename https://github.com/knative/eventing/releases/download/v0.3.0/
     release.yaml \
  --filename https://github.com/knative/eventing-sources/releases/download/v0
     .3.0/release.yaml \
  --filename https://github.com/knative/serving/releases/download/v0.3.0/
     monitoring.yaml
```

Like before this can take a few minutes to provision so to monitior the progress we can run the following:

```
$ kubectl get pods --namespace knative-serving
$ kubectl get pods --namespace knative-build
$ kubectl get pods --namespace knative-eventing
$ kubectl get pods --namespace knative-sources
$ kubectl get pods --namespace knative-monitoring
```

And that is it, we have now provisioned Knative in a Kubernetes cluster running within GKE.

3.2.7 Cleaning up

As we know running instances in the cloud costs money, so to tear down our cluster we must run:

```
$ gcloud container clusters delete $CLUSTER_NAME --zone $CLUSTER_ZONE
```

Deleting the cluster will also remove Knative, Istio, and any apps we have deployed.

3.3 Deploying an App

For this section I will be following on from Section 3.2 with a cluster running in GKE. I will also be using my FYP to create example functions, for brevity I not go into to much detail on my FYP or how to install it, but a link to the documentation can be found in Appendix G.

Serverless development requires a different mindset, this is something Lambda does extremely well, you write your logic, click a button and you have a Serverless function. Coming from a Kubernetes mindset, mixed with working with Knaitve which feels native to Kubernetes it can be tricky to get into the Serverless mindset. When we break down what a Serverless Pod in Kubernetes looks like, we have our business logic, wrapped in a http server, which is wrapped in an image, which then requires a Knative Manifest to be deployed. With lots of moving parts and plenty of noise, one of the main benefits of Serverless is lost. So with that my FYP (RubiX) does all the heavy lifting allowing us to just write our business logic. This section will go through these steps.

First we need to initalise a function, RubiX will bootstrap our function, providing all the config needed, but what we need to focus on is the r3x func file. Here we will have a preconfigured file in the chosen language, for this term paper it will be JavaScript. So in our r3x-func.js we write our business logic, which is then consumed by the RubiX SDK. To initialise a project run the following:

```
$ r3x init <<our function name>> --type js
```

If we dive into the r3x-func.js file we can see our hello world program where we are looking for an environment variable and basically returning some JSON.

```
 r3x.execute(function(){
      const target = process.env.TARGET || 'RubiX';
      let response = {'message' : ‘Hello ${target}!‘}
      return response
})
```

With our function wrote we need it in a state where Knative can use it, this requires it to be turned into an image, pushed somewhere it can be accessed from and a manifest file which will be consumed by Knative. The RubiX build function will handle all this for us. Simply run the following

```
$ r3x build --push --name <<docker hub user name>>
```

If we check back into our project we will see a service.yaml file has be generated:

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: r3x-<<our function name>>
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/<<docker hub user name>>/r3x-<<our function name>>
```

Now we need to add an environment variable to our manifest, this is done in the same way we would with any Kubernetes service. It is also worth mentioning here, Knative treats our pods as services, there is no need to create deployments, a simple service file is all that is needed. With the environement variable added, our manifest should look like this:

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: r3x-<<our function name>>
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/<<docker hub user name>>/r3x-<<our function name>>
            env:
            - name: TARGET
              value: "RubiX␣Sample␣v1"
```

And that is it, to deploy it to our cluster simply run the following:

```
  $ kubectl apply --filename service.yaml
```

What I like to do next is, Knative will instantly spin up 3 pods for our function. This is so Knative can assign a revision number to our latest deployment of the function along with assign it a custom domain name within our cluster and configure the routing. So while this is happening I run the following:

```
 $ kubectl get pods --watch
```

Here we can watch till our pod its status is running. Once done we need to find out our functions domain and the IP address of our cluster gateway. First we will get our cluster IP, to make subsequent calls to our gateway easier we will declare the following variable:

```
$ export INGRESSGATEWAY=istio-ingressgateway
```

Next we can run the following:

```
$ kubectl get svc $INGRESSGATEWAY --namespace istio-system
```

Which will return something similar to:

```
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
istio-ingressgateway LoadBalancer 10.7.241.67 35.246.108.94 80:31380/TCP
    ,443:31390/TCP,31400:31400/TCP,15011:30271/TCP,8060:31049/TCP,853:32614/TCP
    ,15030:30885/TCP,15031:32329/TCP 55m
```

What we are interested in is the External-IP which in this case is 35.246.108.94

```
$ kubectl get ksvc r3x-<<our function name>> --output=custom-columns=NAME:.
    metadata.name,DOMAIN:.status.domain
```

Which will return something similar to:

```
NAME DOMAIN
r3x-hello-knative r3x-hello-knative.default.example.com
```

We now have all the pieces to our puzzle, and we can curl our function. It is worth opening another terminal and running the pod watch command, here we will see that Knative has terminated our pod since we deployed it, after 3 minutes of no access to the pod it gets scaled back to 0.

```
$ curl -H "Host:␣r3x-hello-knative.default.example.com" http://35.246.108.94
Hello RubiX Sample v1!
```

Watching the pods we will see our pods spin up and our response returned. Now the latency is notable, in some cases it can be around 5 seconds, but once the pod is running subsequent calls to

the pod will return in milliseconds.
Thats it we have now spun up a Knative Kubernetes cluster on GKE and deployed our first function, in the next Section we will look at revisions, updating and autoscaling our function.

3.4 Updating and AutoScaling an App

For this section, I recommend opening 4 terminal windows, we will use one to interact with our cluster, in the second terminal we will want to watch our deployments, we can do this with the following command:

```
$ kubectl get deployments --watch
```

We also want to watch our pods, this can be done with the following:

```
$ kubectl get pods --watch
```

Finally we want to poll our function with requests by entering this simple loop, just change the IP address to suit:

```
$ while true; do curl --header "Host:␣r3x-hello-knative.default.example.com" http
    ://35.246.108.94;done

{"message":"Hello␣RubiX␣Sample␣v1!"}
```

We should see from the output v1, now we are going to make a simple change to show a basic rollout to an update, we will make a small change to our service yaml, by incrementing our environment variable.

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: r3x-hello-knative
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/muldoon/r3x-hello-knative
            env:
```

```
          - name: TARGET
            value: "RubiX␣Sample␣v2"
```

Now all we need to do is run:

```
$ kubectl apply --filename service.yaml
```

We should see a new deployment created, along with new pods being created, once they get up and running our existing pods will be terminated. We should also see v2 in the output from our polling script. So lets step up this example and bring in some autoscaling policies to our updates.
So now our updated service.yaml should resemble something like this:

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: r3x-hello-knative
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        metadata:
          annotations:
            # Target 10 in-flight-requests per pod.
            autoscaling.knative.dev/target: "10"
        spec:
          container:
            image: docker.io/muldoon/r3x-hello-knative
            env:
            - name: TARGET
              value: "RubiX␣Sample␣v2"
```

As can be seen, we have added metadata to our revisionTemplate, and we are setting our autoscaling policy target to 10, this will be triggered by 10 requested per pod. In order to test out autoscaling policy we are going to use a tool called Hey, which can be found at Appendix J. To quickly install hey, assuming you have GO installed locally you can just run :

```
$ go get -u github.com/rakyll/hey
```

With hey installed we are going to apply our latest service manifest and poll it. Be sure to change the host to suit. What hey will do, is send 30 seconds of traffic maintaining 50 in-flight requests.

```
$ kubectl apply -f service.yaml
$ hey -z 30s -c 50 \
  -host "r3x-hello-knative.default.example.com" \
  "http://${IP_ADDRESS?}?sleep=100&prime=10000&bloat=5" \
  && kubectl get pods
```

After 30 seconds we will get some output, but on the subsequent get pods we will notice that we did not scale up beyond 3 pods. I initially put this down to Kubernetes handling the traffic with ease as it is essentially a hello world program. After a quick visit to Stackover flow I got a function for calculating prime numbers, now my new functions looks like this

```
"use␣strict";
const r3x = require('@rubixfunctions/r3x-js-sdk')

var primes = [];

r3x.execute(function(){
       return findPrimes(10000000)
})
...
```

We will calculate and return the number of prime numbers in 10 million. Anything more then 10 million I was over flowing a javascript heap. We now run an r3x build, and reapply our service manifest. We will see our latest version roll out. Before running the hey command again I decided to tweak our load balancer to make testing a little easier.

```
$ kubectl patch configmap config-autoscaler \
       --namespace=knative-serving \
       --patch '{"data":{"container-concurrency-target-default":"10"}}'
$ kubectl patch configmap config-autoscaler \
       --namespace=knative-serving \
       --patch '{"data":{"scale-to-zero-threshold":"1m"}}'
$ kubectl patch configmap config-autoscaler \
       --namespace=knative-serving \
       --patch '{"data":{"scale-to-zero-grace-period":"30s"}}'
```

So what we have done here is set the desired container concurrency number to 10 from 100 which is standard, we reduced the scale back to zero from 3 minutes to 1 minute, and set a grace period of 30 seconds on the threshold. Now when we run the hey command we get the following output

```
Summary:
  Total: 33.7802 secs
  Slowest: 16.2597 secs
  Fastest: 0.0946 secs
  Average: 1.5252 secs
  Requests/sec: 30.6392


Response time histogram:
  0.095 [1] |
  1.711 [761] |
  3.328 [42] |
  4.944 [163] |
  6.561 [17] |
  8.177 [24] |
  9.794 [8] |
  11.410 [2] |
  13.027 [2] |
  14.643 [0] |
  16.260 [15] |


Latency distribution:
  10% in 0.1146 secs
  25% in 0.1709 secs
  50% in 0.2784 secs
  75% in 2.8774 secs
  90% in 4.1180 secs
  95% in 6.5950 secs
  99% in 15.3213 secs

Details (average, fastest, slowest):
  DNS+dialup: 0.0011 secs, 0.0946 secs, 16.2597 secs
  DNS-lookup: 0.0000 secs, 0.0000 secs, 0.0000 secs
  req write: 0.0001 secs, 0.0000 secs, 0.0043 secs
  resp wait: 1.5229 secs, 0.0943 secs, 16.2193 secs
  resp read: 0.0006 secs, 0.0000 secs, 0.0968 secs
```

```
Status code distribution:
  [200] 1035 responses




NAME READY STATUS RESTARTS AGE
r3x-hello-knative-wdtqr-deployment-54f896d4fc-89tn8 3/3 Running 0 <invalid>
r3x-hello-knative-wdtqr-deployment-54f896d4fc-fpg5l 3/3 Running 0 <invalid>
r3x-hello-knative-wdtqr-deployment-54f896d4fc-mcdjr 3/3 Running 0 <invalid>
r3x-hello-knative-wdtqr-deployment-54f896d4fc-vp2hr 3/3 Running 0 <invalid>
r3x-hello-knative-wdtqr-deployment-54f896d4fc-w27h7 3/3 Running 0 <invalid>
```

With some basic math we can break down what happened here. We sent 50 concurrent requests which meant the autoscaler created 5 pods, 50 requests / target 10 = 5 pods. Its actually really cool under the hood, the autoscaler is calculating the average concurrency over a 60 second window, so every 60 seconds the autoscaler stabilizes, but if it comes under heavy load it will trigger a 6 second panic window. If the target concurrency is doubled in 6 seconds the autoscaler goes into panic mode and will handle the load, once the panic conditions are no longer met for 60 seconds the autoscaler then returns to the initial 60 second stable window.

A really cool feature Knative does is allow for AB deployments through revisions. Following the documentation to demonstrate this, I ran into problems. I reached out to the Knative community and I was told that currently what I am trying to do is not supported as it conflicts with other operations.

Lets break down what I am trying to do, if we remember how the Knative Serve is made up of a config and a route, the config holds revisions of each deploy we do of a specific service. The route then forwards traffic to these revisions. So we can actually declare a percentageRollout to which traffic is routed across revisions. The problem I am running into is that when you apply the manifest, Knative relys on Kubernetes generateName to produce new Revision names. This gives each revision a random string appended to it. Meaning at time of deployment I do not know the revision name so I can not declare a rollout percentage to direct traffic between my new revision at deployment of a new revision.

Knative engineers on slack told me they are currently working on a proposal to solve this problem. So while this is not supported I am still able to show how easy we can direct traffic through different revisions.

For this example I will be going back to our original function where we just returned the target value. Now we will update the service.yaml and enter a new number in our env value parameter. Once done apply this change to our cluster. We must rinse and repeat, changing the env value parameter and applying the service.yaml file. If we now run the following:

```
$ kubectl get deployments
```

We will be returned with something that looks like this:

```
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
r3x-hello-knative-sr958-deployment 0 0 0 0 1m
r3x-hello-knative-wnqcj-deployment 0 0 0 0 3m
```

We now have our two deployments in Kubernetes, Knative id's these as revisions in that it sees *r3x-hello-knative-sr958-deployment* as revision *r3x-hello-knative-sr958*. Now we have everything we need, so we can now update our service.yaml, by replacing *runLatest* under our spec with:

```
...
spec:
  release:
    revisions:
      - r3x-hello-knative-sr958
      - r3x-hello-knative-wnqcj
    rolloutPercent: 50
    configuration:
...
```

What we are doing here is declaring our revisions for this deployment, and setting the rolloutPercent to 50, meaning traffic will be split 50 percent across both revisions.

```
$ while true; do curl --header "Host:␣r3x-hello-knative.default.example.com" http
    ://35.197.208.7;done

{"message":"Hello␣RubiX␣Sample␣v1!"}{"message":"Hello␣RubiX␣Sample␣v2!"}{"message"
    :"Hello␣RubiX␣Sample␣v1!"}{"message":"Hello␣RubiX␣Sample␣v1!"}...
```

And if we are watching our pods, we will see the two previous deployments spin up and our output from the curl will show the traffic split. This split does not appear to be in a round robin fashion, but does look to be around the 50 percent mark.
The current Knative project is in alpha at the moment, they are working hard on proposals for their Beta version, and as mentioned above they are looking towards stream lining this into a cleaner process as the infrastructure is there to easily allow us to configure our routes.

3.5 Building and Deploying a Real World App

So far we have mostly looked at helloworld examples of Knative. While they do showcase the technology and its capabilities often going from a helloworld to real app can be an up hill struggle but the knowledge gained from pushing passed helloworld truly is valuable. So with that I set out to develop a Web App on Knative. As I was searching for a novel idea as to what application I will build, as the actual use case of the app is unimportant what is important is the underlying technologies. When presented with an end 2 end lab in class for AWS and Lambda I decided to replicate this app on Knative and GCP, as it does give me some context to work off and pretty much a bench mark to compare off.
The architecture of the App consists of the following and can be seen visually in Figure 8:

- GKE cluster (Knative)
- Google Storage
- Google Datastore
- Google Speech API
- 1 NginX Server (Front End)
- 4 RubiX Functions (Create, Delete, Dictate, List)

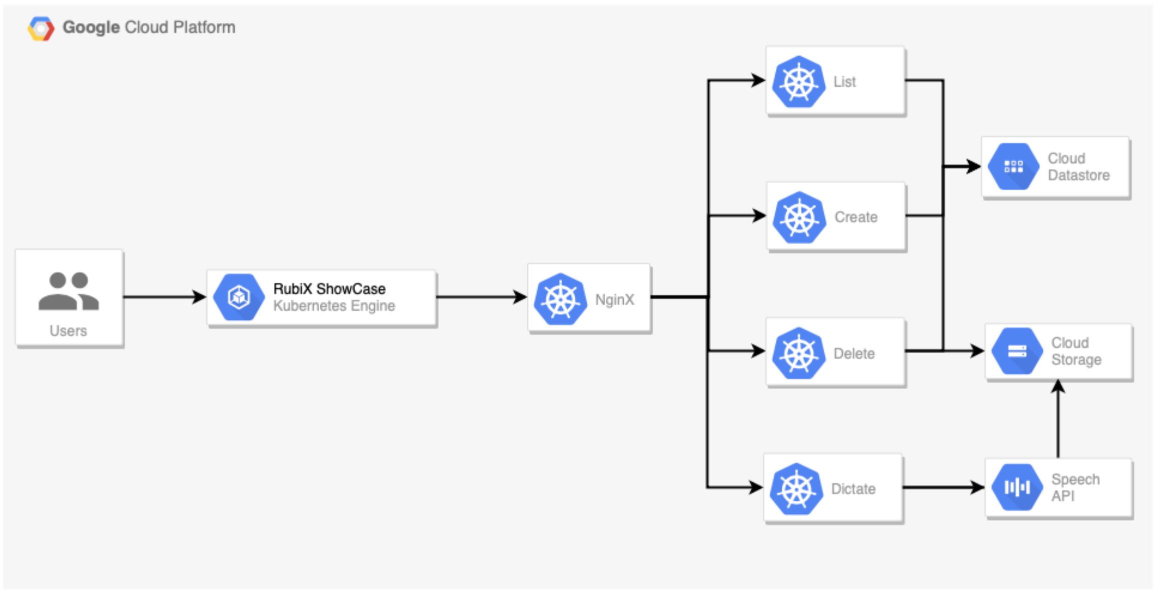

Figure 8: *Showcase App*

Before beginning, It is worth noting the biggest difference when working on GCP over AWS is their

use of Projects. You create specific projects and in that project you build the architecture you need for your application. Which personally I feel is a nice touch as it keeps everything cleaner and more manageable when working on different projects, there is no naming conflicts and when the project is over you can delete the project, and all associated services in one click. When working on a budget this is a nice piece of mind. Another item that is worth the mention, which can be quite irritating at times is how they handle billing. So everything by default is locked to you. When you provision a specific service for the first time in a new project, lets say google database then you have to enable billing for google database for that project. Quite an easy step to do, but can become tedious.
Now with that out of the way my first port of call was to create my project, enable my services and billing. Create a service user and download that key, which is a JSON file. The service user is given permissions to interact with the necessary services in our project. Enabling these services is quite mundane and well documented so for brevity I will be leaving them out of this paper.
As Kubernetes clusters can be expensive I decided to build my application locally first before deploying it. My google services where provisioned so I could easily interact with them after I exported my service account key:

```
$ export GOOGLE_APPLICATION_CREDENTIALS="/Users/ciaranroche/.gcd/rubix.json"
```

Once you export the path as specified above the Google SDK's will authenticate you and allow your localhost access. So a few days of development later I had my application running locally on 5 different ports all as node servers. Now this is where the fun starts.
So we spin up the cluster as described in Section 3.2, next we need to mount our service user key as a secret. The full deployment file can be found at Appendix K, but what we are interested here is:

```
    ...
        image: docker.io/muldoon/r3x-rubix-list
        env:
          - name: GOOGLE_APPLICATION_CREDENTIALS
            value: /var/secret/rubix.json
        volumeMounts:
          - name: rubix-secret
            mountPath: /var/secret
      volumes:
        - name: rubix-secret
          secret:
            secretName: google-rubix-secret
```

Each service we add a volume to our pod which points to the secret in our cluster. From that volume we mount it to our container. Next we set an environment variable which we give it that value of where our service user json file is. Before we can deploy that file we must create the secret in our cluster. This can be done:

```
$ kubectl create secret generic google-rubix-secret --from-file=./rubix.json
```

This is where I run into the first problem. So my app runs fine local, now when I deploy it when I ping my pods I get no response. I check the logs and all I get is an error 14, with little other information. When I go looking online I find it hard to find anything about it only it is a TCP error. With nothing else to go on, I end up sitting down with 2 engineers at RedHat for an hour and they are unable to figure it out, so I reach out to the Knative community and create a thread on their Google Group. It turned out by default Istio and Knative does not allow traffic out from the cluster and we need to configure this outbound traffic ourselves. So in order to do this, we first need to determine the IP ranges of our cluster, the command here will cover this:

```
$ gcloud container clusters describe ${CLUSTER_NAME} \
--zone=${CLUSTER_ZONE} | grep -e clusterIpv4Cidr -e servicesIpv4Cidr
```

Now we have our range we need to set this scope to the istio sidecar to include these ranges. To do this we enter the following command and we edit the config file to something similar as below:

```
$ kubectl edit configmap config-network --namespace knative-serving


...
apiVersion: v1
data:
  istio.sidecar.includeOutboundIPRanges: '10.16.0.0/14,10.19.240.0/20'
kind: ConfigMap
metadata:
...
```

Now I redeployed my functions and was able to curl these functions from my terminal. Everything was looking good, it was just a matter of testing my frontend with my function end points. I updated my frontend and spun it up locally. This is where I ran into a world of problems with CORS. I dug into my actual HTTP server for my functions and ensured CORS was enabled, but it did not make a difference, I still ran into errors, finally I disabled CORS in the browser as I noticed the preflight OPTIONS request was not sending my host header to my cluster. When CORS was disabled my error changed to a 404. Again I was left scratching my head and could not find anything in the Knative docs, so I reached out to the community again and a workaround was given. Knative as of now does not allow us to configure CORS at the cluster domain gateway. The workaround was to leverage xip.io to create a dns wildcard and give us a url to directly access our pods. To do this we run the following:

```
# Get Cluster IP address
$ export IP_ADDRESS=$(kubectl get svc istio-ingressgateway -n istio-system -
    ojsonpath='{.status.loadBalancer.ingress[0].ip}')

# Configure Knative to use xip.io as domain suffix
$ kubectl patch cm -n knative-serving config-domain \
  --type merge \
  -p "{\"data\":{\"$IP_ADDRESS.xip.io\":\"\"}}"
```

With that done we can get our domain URL for our services:

```
$ kubectl get ksvc r3x-rubix-list --output=custom-columns=NAME:.metadata.name,
    DOMAIN:.status.domain
```

An example output will be similar to this:

```
NAME DOMAIN
r3x-rubix-list http://r3x-rubix-list.default.35.246.108.94.xip.io
```

Now we can directly get to our service using the Domain address and allow us access it from the browser. I updated the frontend app with the latest domains, an example of one of my functions in the frontend app is:

```
  list() {
    var options = {
        method: 'POST',
        url: 'http://r3x-rubix-list.default.35.246.108.94.xip.io',
        headers:
        {
            'Content-Type': 'application/json'
        },
    };
    return new Promise(function (resolve, reject) {
        request(options, function (err, resp, body) {
            if (err) {
              reject(err);
            } else {
              resolve(body)
            }
        }).catch(function (err) {
            console.log(err)
        })
    })
}
```

We just need to deploy the frontend app now that we have our end points. Couple of things I want to point out here:

```
# Stage 1 - Build App
FROM node:8.15 as build-deps
WORKDIR /usr/src/app
COPY package.json package-lock.json ./
RUN npm i
COPY . ./
RUN npm run build
# Stage 2 - Prod Build
FROM nginx:1.15.9-alpine
COPY --from=build-deps /usr/src/app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon␣off;"]
```

First is the Dockerfile seen above, As my frontend is developed with React and Webpack, when I run npm build, webpack compiles my frontend to vanilla javascript and html. Because of this I can use a multistage Docker build. Now this is really cool in my opinon. In the first stage I am pulling in a full operating system with all node dependencies to allow me to build my front end. So once I use this stage to build my app, I can now kick off stage two where I just pull in an alpine nginx image, I can copy over my built files from stage one, expose my port and set my start command. A little bit of extra work at the Dockerfile gives me a smaller more secure Image for my frontend app. Now unfortubatly the next thing I want to point out is down to time which I dont have. Knative requires pods to listen on 8080, NginX runs at port 80 by default, to go in an configure this to work on port 8080 required time I didnt have. Meaning I could not deploy this to Knative. Which would of been really cool as it would of been a full serverless web app. So instead I have opted to deploy it as a normal deployment within my cluster. To do this I need to run the following to build and push our frontend:

```
$ r3x build -p -n <<your docker user name>>
```

Now we will use kubectl to deploy the image to our cluster with this command:

```
$ kubectl run rubix-web --image=docker.io/<<your docker user name>>/r3x-rubix-
    frontend --port 80
```

Our app is now running in our cluster, we can run get pods to verify this. Once the pod is running we need to expose this pod to the internet. Again with kubectl we enter:

```
$ kubectl expose deployment rubix-web --type=LoadBalancer --port 80 --target-port
    80
```

We just need the external IP of our new deployment of our frontend app. To get this we run:

```
$ kubectl get service
```

There you have it, once we get the IP we can access our app in a browser and interact with our serverless functions via our web app. A demo video of this showcase app can be seen in Appendix I. If you would like to run the application yourself, go to the GitHub repo at Appendix H and follow the documentation there.

4 Summary

For my summary, I really do not know where to begin. It has been an awesome journey this semester working with and on Knative. From a high level non technical view of the project, I can not say enough good things about it and the community behind it. Between the work for my FYP and this term paper, I have created several Google Group Threads, Github Issues, and an abundance of slack conversations with the devs. Along with one contribution to the project too. Without the community, I do not think I would of had a showcase app running to demo. I can really see the appeal for people and Open Source when they experience what I have experienced these last few weeks.
Now to dig a bit deeper, Knative is super new, still in Alpha and only announced around a year ago. They put emphasis around the pronunciation of the name, being Knative (Kay nay tive) this is because before it was announced people wanted to make the 'K' silent and just pronounce it as 'native'. Why am I telling you this? because they made a big deal about the 'native' part, the main goal of the project is to make something that feels native to Kubernetes and native to cloud centric development. All I can say, in these early days is they have achieved this. As their components are custom resources we are getting new primatives to interact with Kubernetes, just like calling get pods, we can call get builds. It all feels right when working with it.
It does a great deal in abstracting away unnecessary bits that are normally needed for development in Kubernetes. For example, to deploy one of my functions, say the list function to Kubernetes it required a manifest file 40 + lines long. A Knative manifest for the same deploy is just 22 lines. Around half the size. Why do I know this? I did find debugging tricky in a sense, as I would jump into a pod to see what was going on and if it would be possible to get more logs then what I was getting from the Google Console, then it would be scaled back to 0, so to give myself some freedom I deployed my list function as a normal pod to give me some time to debug and troubleshoot.
Which brings me nicely to the down side of Knative, which I found was the debugging, much like the same lab with AWS there is so many moving parts, where do you begin? where is the issue? According to the docs you debug Knative the same as you debug any Kubernetes app, which now I am comfortable to do, but at the time when I was running into errors I was out of my depth. And the whole experience was a nasty one, mixed with the pressure of the this term paper and my FYP riding on over coming the problems.
On more downside is the initial spin up time of a pod from 0. They have it benchmarked at around 4 seconds which is way too long, they admit that. But as this project is backed by so many companies, IBM is spearheading work on getting that to sub 1 second times, and according to the devs they are close, soon we will have sub 1 second spin up from 0, which is just mind blowing. On the back of that, and something I mentioned before, is all the exciting tools which are being built, providing more abstraction and cleaner work flows to developing on Kubernetes. Only time will tell how far and how big this will go.

But we got to remember that this is just a tool, there is many serverless platforms out there, as it is a hot topic. So it really boils down to what tool makes the most sense for you. But one thing I do feel, all these platforms can take something from each other and learn from each other going forward.

Appendices

A Google Cloud Knative

https://cloud.google.com/knative/

B Knative Docs

https://github.com/knative/docs/

C Knative Build

https://github.com/knative/build

D Knative Event

https://github.com/knative/eventing

E Knative Serve

https://github.com/knative/eventing

F Kubernetes Docs

https://kubernetes.io/docs/home/

G RubiX Functions Docs

https://github.com/rubixFunctions/r3x-docs

H RubiX Showcase

https://github.com/rubixFunctions/r3x-showcase-apps

I RubiX Showcase Demo Video

https://youtu.be/LA1nVQK8ETQ

J Hey

https://github.com/rakyll/hey

K Showcase Deployment File

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: r3x-rubix-create
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/muldoon/r3x-rubix-create
            env:
              - name: GOOGLE_APPLICATION_CREDENTIALS
                value: /var/secret/rubix.json
            volumeMounts:
              - name: rubix-secret
                mountPath: /var/secret
          volumes:
            - name: rubix-secret
              secret:
                secretName: google-rubix-secret
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: r3x-rubix-delete
  namespace: default
```

```
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/muldoon/r3x-rubix-delete
            env:
              - name: GOOGLE_APPLICATION_CREDENTIALS
                value: /var/secret/rubix.json
            volumeMounts:
              - name: rubix-secret
                mountPath: /var/secret
          volumes:
            - name: rubix-secret
              secret:
                secretName: google-rubix-secret
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: r3x-rubix-dictate
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/muldoon/r3x-rubix-dictate
            env:
              - name: GOOGLE_APPLICATION_CREDENTIALS
                value: /var/secret/rubix.json
            volumeMounts:
              - name: rubix-secret
                mountPath: /var/secret
          volumes:
            - name: rubix-secret
          secret:
            secretName: google-rubix-secret
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: r3x-rubix-list
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/muldoon/r3x-rubix-list
            env:
              - name: GOOGLE_APPLICATION_CREDENTIALS
                value: /var/secret/rubix.json
            volumeMounts:
              - name: rubix-secret
                mountPath: /var/secret
          volumes:
            - name: rubix-secret
              secret:
                secretName: google-rubix-secret
```