Why migrate from a monolithic application to a microservices architecture? Breaking down an application into microservices has the following advantages, most of these stem from the fact that microservices are loosely coupled:

- The microservices can be independently tested and deployed. The smaller the unit of deployment, the easier the deployment.
- They can be implemented in different languages and frameworks. For each microservice, you're free to choose the best technology for its particular use case.
- They can be managed by different teams. The boundary between microservices makes it easier to dedicate a team to one or several microservices.
- By moving to microservices, you loosen the dependencies between the teams. Each team has to care only about the APIs of the microservices they are dependent on. The team doesn't need to think about how those microservices are implemented, about their release cycles, and so on.
- You can more easily design for failure. By having clear boundaries between services, it's easier to determine what to do if a service is down.

Some of the disadvantages when compared to monoliths are:

- Because a microservice-based app is a network of different services that often interact in ways that are not obvious, the overall complexity of the system tends to grow.
- Unlike the internals of a monolith, microservices communicate over a network. In some circumstances, this can be seen as a security concern. Istio solves this problem by automatically encrypting the traffic between microservices.
- It can be hard to achieve the same level of performance as with a monolithic approach because of latencies between services.
- The behavior of your system isn't caused by a single service, but by many of them and by their interactions. Because of this, understanding how your system behaves in production (its observability) is harder. Istio is a solution to this problem as well.

In this lab you will deploy an existing monolithic application to a Google Kubernetes Engine cluster, then break it down into microservices. Kubernetes is a platform to manage, host, scale, and deploy containers. Containers are a portable way of packaging and running code. They are well suited to the microservices pattern, where each microservice can run in its own container.

Architecture diagram of our microservices

Start by breaking the monolith into three microservices, one at a time. The microservices include, Orders, Products, and Frontend. Build a Docker image for each microservice using Cloud Build, then deploy and expose the microservices on Google Kubernetes Engine (GKE) with a Kubernetes service type LoadBalancer. You will do this for each service while simultaneously refactoring them out of the monolith. During the process you will have both the monolith and the microservices running until the very end when you are able to delete the monolith.

What you'll learn

- How to break down a Monolith to Microservices
- How to create a Google Kubernetes Engine cluster
- How to create a Docker image
- How to deploy Docker images to Kubernetes

You can list the active account name with this command:

gcloud auth list

```
Welcome to Cloud Shell! Type "help" to get started.
To set your Cloud Platform project in this session use "gcloud config set project [PROJECT_ID]"
student_00_70ca9b59ae67@cloudshell:~$ gcloud auth list
Credentialed Accounts

ACTIVE: *
ACCOUNT: student-00-70ca9b59ae67@qwiklabs.net

To set the active account, run:
    $ gcloud config set account `ACCOUNT`

student_00_70ca9b59ae67@cloudshell:~$
```

You can list the project ID with this command:

gcloud config list project

```
student_00_70ca9b59ae67@cloudshell:~$ gcloud config list project
[core]
project (unset)

Your active configuration is: [cloudshell-23313]
student_00_70ca9b59ae67@cloudshell:~$
```

Set the default zone and project configuration:

gcloud config set compute/zone (zone)

```
student_00_70ca9b59ae67@cloudshell:~ (qwiklabs-gcp-03-c894137e5528)$ gcloud config list project
[core]
project = qwiklabs-gcp-03-c894137e5528

Your active configuration is: [cloudshell-23313]
student_00_70ca9b59ae67@cloudshell:~ (qwiklabs-gcp-03-c894137e5528)$ gcloud config set compute/zone us-west1-a
Updated property [compute/zone].
student_00_70ca9b59ae67@cloudshell:~ (qwiklabs-gcp-03-c894137e5528)$
```

Task 1. Clone the source repository

You will use an existing monolithic application of an imaginary ecommerce website, with a simple welcome page, a products page and an order history page. We will just need to clone the source from our git repo, so we can focus on breaking it down into microservices and deploying to Google Kubernetes Engine (GKE).

- Run the following commands to clone the git repo to your Cloud Shell instance and change to the appropriate directory. You will also install the NodeJS dependencies so you can test your monolith before deploying:

```
cd ~
git clone https://github.com/googlecodelabs/monolith-to-microservices.git
cd ~/monolith-to-microservices
```

./setup.sh

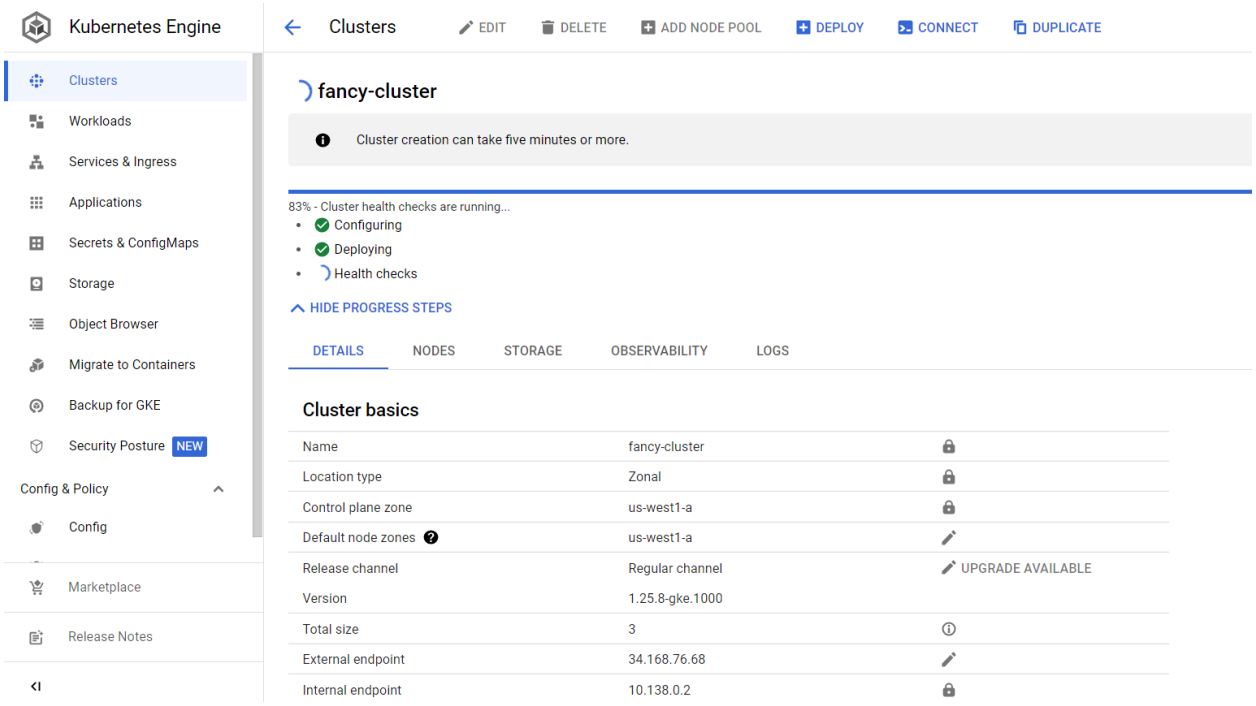
Task 2. Create a GKE cluster
Now that you have your working developer environment, you need a Kubernetes cluster to deploy your monolith, and eventually the microservices, to! Before you can create a cluster, make sure the proper API's are enabled.

1. Run the following command to enable the Containers API so you can use Google Kubernetes Engine:

gcloud services enable container.googleapis.com

2. Run the command below to create a GKE cluster named **fancy-cluster** with **3** nodes:

gcloud container clusters create fancy-cluster --num-nodes 3 --machine-type=e2-standard-4



3. Once the command has completed, run the following to see the cluster's three worker VM instances:

gcloud compute instances list

```
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices (qwiklabs-gcp-03-c894137e5528)$ gcloud compute instances list
NAME: gke-fancy-cluster-default-pool-49f5c9a8-49ts
ZONE: us-west1-a
MACHINE_TYPE: e2-standard-4
PREEMPTIBLE:
INTERNAL_IP: 10.138.0.5
EXTERNAL_IP: 34.145.54.76
STATUS: RUNNING

NAME: gke-fancy-cluster-default-pool-49f5c9a8-5f1q
ZONE: us-west1-a
MACHINE_TYPE: e2-standard-4
PREEMPTIBLE:
INTERNAL_IP: 10.138.0.3
EXTERNAL_IP: 34.145.86.234
STATUS: RUNNING

NAME: gke-fancy-cluster-default-pool-49f5c9a8-g9mn
ZONE: us-west1-a
MACHINE_TYPE: e2-standard-4
PREEMPTIBLE:
INTERNAL_IP: 10.138.0.4
EXTERNAL_IP: 34.145.55.33
STATUS: RUNNING
```

You can also view your Kubernetes cluster and related information in the Cloud Console. From the **Navigation menu**, scroll down to **Kubernetes Engine** and click **Clusters**.

You should see your cluster named ***fancy-cluster***.

Task 3. Deploy the existing monolith

Since the focus of this lab is to break down a monolith into microservices, you need to get a monolith application up and running.

- Run the following script to deploy a monolith application to your GKE cluster:

cd ~/monolith-to-microservices
./deploy-monolith.sh

```
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices (qwiklabs-gcp-03-c894137e5528)$ ./deploy-monolith.sh
Enabling Cloud Build APIs...
Completed.

Building Monolith Container...
Creating temporary tarball archive of 27 file(s) totalling 2.5 MiB before compression.
Uploading tarball of [.] to [gs://qwiklabs-gcp-03-c894137e5528_cloudbuild/source/1688144879.336566-b26ade4819524340a1c37a076c91022b.tgz]
Created [https://cloudbuild.googleapis.com/v1/projects/qwiklabs-gcp-03-c894137e5528/locations/global/builds/5b1abdcb-7e89-4b2e-abd0-3bbbb57a4713].
Logs are available at [ https://console.cloud.google.com/cloud-build/builds/5b1abdcb-7e89-4b2e-abd0-3bbbb57a4713?project=3091839841 ].
------------------------------------------------------------------------ REMOTE BUILD OUTPUT ------------------------------------------------------------------------
starting build "5b1abdcb-7e89-4b2e-abd0-3bbbb57a4713"

FETCHSOURCE
Fetching storage object: gs://qwiklabs-gcp-03-c894137e5528_cloudbuild/source/1688144879.336566-b26ade4819524340a1c37a076c91022b.tgz#1688144883968924
Copying gs://qwiklabs-gcp-03-c894137e5528_cloudbuild/source/1688144879.336566-b26ade4819524340a1c37a076c91022b.tgz#1688144883968924...
/ [1 files][  1.5 MiB/  1.5 MiB]
Operation completed over 1 objects/1.5 MiB.
```

Accessing the monolith

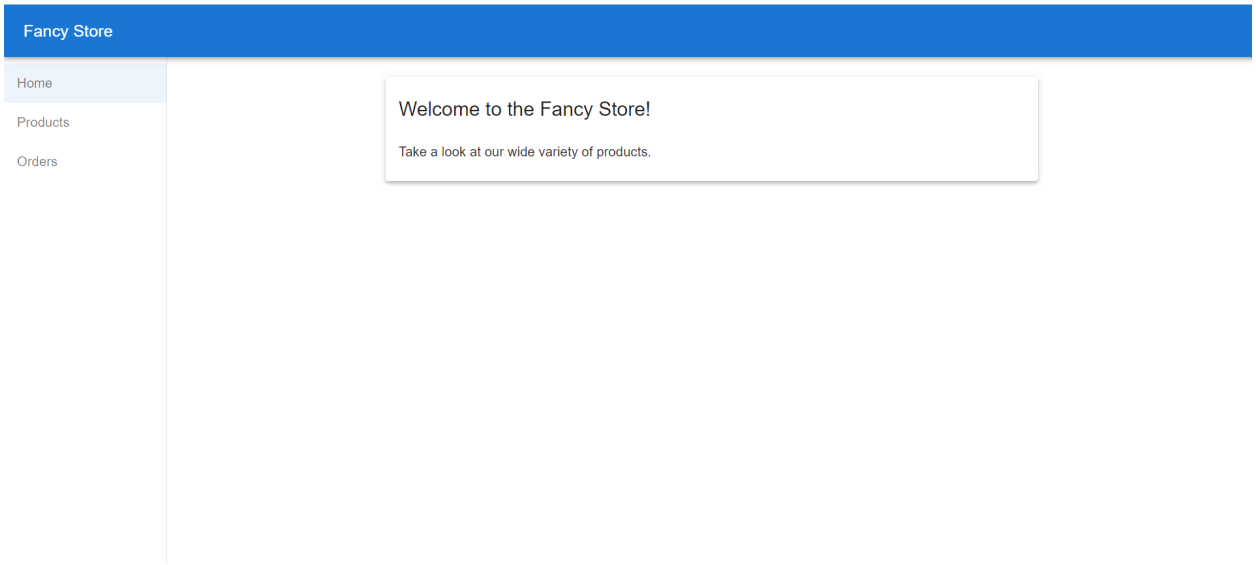
1. To find the external IP address for the monolith application, run the following command:

kubectl get service monolith

```
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices (qwiklabs-gcp-03-c894137e5528)$ kubectl get service monolith
NAME       TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE
monolith   LoadBalancer   10.96.11.77   35.230.62.224   80:31636/TCP   119s
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices (qwiklabs-gcp-03-c894137e5528)$
```

2. Once you've determined the external IP address for your monolith, copy the IP address. Point your browser to this URL (such as http://203.0.113.0) to check if your monolith is accessible.

You should see the welcome page for the monolithic website. The welcome page is a static page that will be served up by the Frontend microservice later on. You now have your monolith fully running on Kubernetes!



Task 4. Migrate orders to a microservice

Now that you have a monolith website running on GKE, start breaking each service into a microservice. Typically, a planning effort should take place to determine which services to break into smaller chunks, usually around specific parts of the application like business domain.

For this lab you will create an example and break out each service around the business domain: Orders, Products, and Frontend. The code has already been migrated for you so you can focus on building and deploying the services on Google Kubernetes Engine (GKE).

Create new orders microservice

The first service to break out is the Orders service. Make use of the separate codebase provided and create a separate Docker container for this service.

Create a Docker container with Cloud Build

Since the codebase is already available, your first step will be to create a Docker container of your Order service using Cloud Build.

Normally this is done in a two step process that entails building a Docker container and pushing it to a registry to store the image for GKE to pull from. Cloud Build can be used to build the Docker container ***and*** put the image in the Container Registry with a single command! This allows you to issue a single command to build and move your image to the container registry. To learn about the manual process of creating a docker file and pushing it, you can go to the Google Cloud Container Registry documentation and view Quickstart for Container Registry.

Google Cloud Build will compress the files from the directory and move them to a Cloud Storage bucket. The build process will then take all the files from the bucket and use the Dockerfile to run the Docker build process. The `--tag` flag is specified with the host as gcr.io for the Docker image, the resulting Docker image will be pushed to the Google Cloud Container Registry.

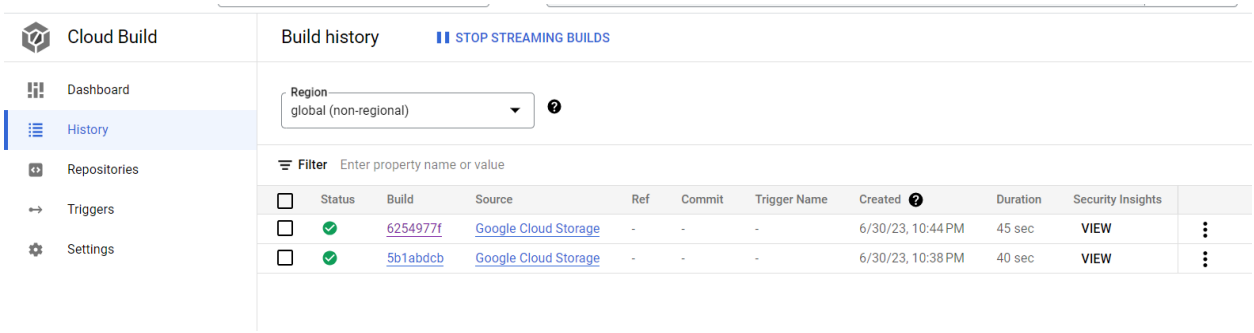
1. Run the following commands to build your Docker container and push it to the Google Container Registry:

```
cd ~/monolith-to-microservices/microservices/src/orders
gcloud builds submit --tag gcr.io/${GOOGLE_CLOUD_PROJECT}/orders:1.0.0 .
```

This process will take a minute, but after it is completed, there will be output in the terminal similar to the following:

2. To view your build history or watch the process in real time, in the Console click the **Navigation Menu** button on the top left and scroll down to **Tools** and click **Cloud Build** > **History**. Here you can see a list of all your previous builds, there should only be 1 that you just created.

If you click on the build ID, you can see all the details for that build including the log output.



From the build details page you can view the container image that was created by clicking on the Execution Details tab.

To view the container image that was created, from the Build details page, in the right section click the Execution Details tab and see Image.

Deploy container to GKE

Now that you have containerized the website and pushed the container to the Google Container Registry, it is time to deploy to Kubernetes!

Kubernetes represents applications as Pods, which are units that represent a container (or group of tightly-coupled containers). The Pod is the smallest deployable unit in Kubernetes. In this tutorial, each Pod contains only your microservices container.

To deploy and manage applications on a GKE cluster, you must communicate with the Kubernetes cluster management system. You typically do this by using the **kubectl** command-line tool from within Cloud Shell.

First, create a Deployment resource. The Deployment manages multiple copies of your application, called replicas, and schedules them to run on the individual nodes in your cluster. In this case, the Deployment will be running only one pod of your application. Deployments ensure this by creating a ReplicaSet. The ReplicaSet is responsible for making sure the number of replicas specified are always running.

The `kubectl create deployment` command below causes Kubernetes to create a Deployment named **Orders** on your cluster with **1** replica.

- Run the following command to deploy your application:

kubectl create deployment orders --image=gcr.io/${GOOGLE_CLOUD_PROJECT}/orders:1.0.0

```
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/microservices/src/orders (qwiklabs-gcp-03-c894137e5528)$ kubectl create deployment orders --image=gcr.io/${GOOGLE_CLOUD_PROJECT}/
orders:1.0.0
deployment.apps/orders created
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/microservices/src/orders (qwiklabs-gcp-03-c894137e5528)$
```

Note: As a best practice, using a YAML file is recommended to declare your change to the Kubernetes cluster (e.g. creating or modifying a deployment or service) and a source control system such as GitHub or Cloud Source Repositories to store those changes. You can learn more about this from the Kubernetes Deployments Documentation.

Verify the deployment

- To verify the Deployment was created successfully, run the following command:

`kubectl get all`

```
deployment.apps/orders created
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/microservices/src/orders (qwiklabs-gcp-03-c894137e5528)$ kubectl get all
NAME                            READY   STATUS    RESTARTS   AGE
pod/monolith-69558fbc6-5vxgq    1/1     Running   0          13m
pod/orders-64bd7d97-x6ggh       1/1     Running   0          107s

NAME                 TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
service/kubernetes   ClusterIP      10.96.0.1      <none>          443/TCP        20m
service/monolith     LoadBalancer   10.96.11.77    35.230.62.224   80:31636/TCP   13m

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/monolith    1/1     1            1           13m
deployment.apps/orders      1/1     1            1           108s

NAME                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/monolith-69558fbc6     1         1         1       13m
replicaset.apps/orders-64bd7d97        1         1         1       108s
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/microservices/src/orders (qwiklabs-gcp-03-c894137e5528)$
```

You can see your Deployment which is current, the `replicaset` with the desired pod count of 1, and the pod which is running. Looks like everything was created successfully!

You can also view your Kubernetes deployments in the Cloud Console from the **Navigation menu**, go to **Kubernetes Engine** > **Workloads**.

Expose GKE container

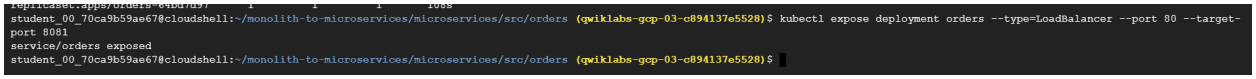
We have deployed our application on GKE, but we don't have a way of accessing it outside of the cluster. By default, the containers you run on GKE are not accessible from the Internet, because they do not have external IP addresses. You must explicitly expose your application to traffic from the Internet via a Service resource. A Service provides networking and IP support to your application's Pods. GKE creates an external IP and a Load Balancer (subject to billing - from the Google Cloud website, review Compute Engine pricing) for your application.

For purposes of this lab, the exposure of the service has been simplified. Typically, you would use an API gateway to secure your public endpoints. In the Google Cloud Architecture Center, read more about microservices best practices.

When you deployed the Orders service, you exposed it on port 8081 internally via a Kubernetes deployment. In order to expose this service externally, you need to create a Kubernetes service of type `LoadBalancer` to route traffic from port 80 externally to internal port 8081.

- Run the following command to expose your website to the Internet:

```
kubectl expose deployment orders --type=LoadBalancer --port 80 --target-port 8081
```



Accessing the service
GKE assigns the external IP address to the Service resource, not the Deployment.

- If you want to find out the external IP that GKE provisioned for your application, you can inspect the Service with the `kubectl get service` command:

```
kubectl get service orders
```

Once you've determined the external IP address for your application, copy the IP address. Save it for the next step when you change your monolith to point to the new Orders service!

Reconfigure the monolith

Since you removed the Orders service from the monolith, you will have to modify the monolith to point to the new external Orders microservice.

When breaking down a monolith, you are removing pieces of code from a single codebase to multiple microservices and deploying them separately. Since the microservices are running on a different server, you can no longer reference your service URLs as absolute paths - you need route to the Order microservice server address. This will require some downtime to the monolith service to update the URL for each service that has been broken out. This should be accounted for when planning on moving your microservices and monolith to production during the microservices migration process.

You need to update your config file in the monolith to point to the new Orders microservices IP address.

1. Use the nano editor to replace the local URL with the IP address of the Orders microservice:

```
cd ~/monolith-to-microservices/react-app
nano .env.monolith
```

When the editor opens, your file should look like this:

```
REACT_APP_ORDERS_URL=/service/orders
REACT_APP_PRODUCTS_URL=/service/products
```

2. Replace the REACT_APP_ORDERS_URL to the new format while replacing with your Orders microservice IP address so it matches below:

```
REACT_APP_ORDERS_URL=http://<ORDERS_IP_ADDRESS>/api/orders
REACT_APP_PRODUCTS_URL=/service/products
```

```
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/react-app (qwiklabs-gcp-03-c894137e5528)$ cat .env.monolith
REACT_APP_ORDERS_URL=http://34.82.150.87/api/orders
REACT_APP_PRODUCTS_URL=/service/products
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/react-app (qwiklabs-gcp-03-c894137e5528)$
```

3. Test the new microservice by navigating the URL you just set in the file. The webpage should return a JSON response from your Orders microservice.

Not secure | 34.82.150.87/api/orders


```
// 20230630230321
// http://34.82.150.87/api/orders

[
  {
    "id": "ORD-000001-MICROSERVICE",
    "date": "7/01/2019",
    "cost": 67.99,
    "items": [
      "OLJCESPC7Z"
    ]
  },
  {
    "id": "ORD-000002-MICROSERVICE",
    "date": "7/24/2019",
    "cost": 124,
    "items": [
      "1YMWWN1N4O"
    ]
  },
  {
    "id": "ORD-000003-MICROSERVICE",
    "date": "8/03/2019",
    "cost": 12.49,
    "items": [
      "66VCHSJNUP"
    ]
  },
  {
```

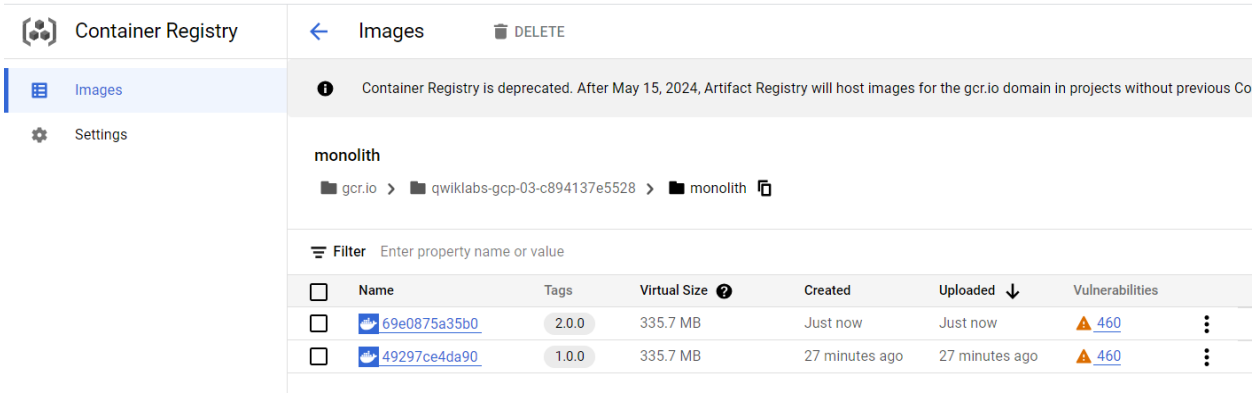
4. Next, rebuild the monolith frontend and repeat the build process to build the container for the monolith and redeploy to the GKE cluster:

- Rebuild Monolith Config Files:

```
npm run build:monolith
```

6. Create Docker Container with Cloud Build:

```
cd ~/monolith-to-microservices/monolith
gcloud builds submit --tag gcr.io/${GOOGLE_CLOUD_PROJECT}/monolith:2.0.0 .
```



7. Deploy Container to GKE:

```
kubectl set image deployment/monolith
monolith=gcr.io/${GOOGLE_CLOUD_PROJECT}/monolith:2.0.0
```

8. Verify the application is now hitting the Orders microservice by going to the monolith application in your browser and navigating to the Orders page. All the order ID's should end in a suffix -MICROSERVICE as shown below:

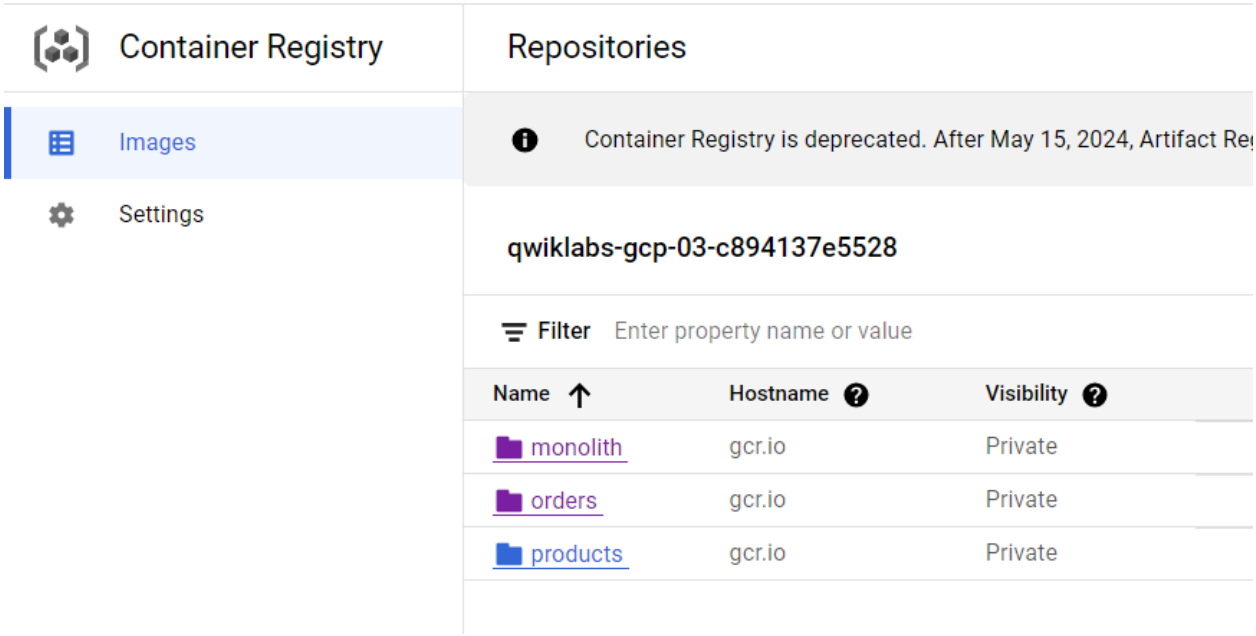
Task 5. Migrate Products to microservice

Create new Products microservice

Continue breaking out the services by migrating the Products service next. Follow the same process as before. Run the following commands to build a Docker container, deploy your container and expose it to via a Kubernetes service.

1. Create Docker Container with Cloud Build:

cd ~/monolith-to-microservices/microservices/src/products
gcloud builds submit --tag gcr.io/${GOOGLE_CLOUD_PROJECT}/products:1.0.0 .



2. Deploy Container to GKE:

```
kubectl create deployment products
--image=gcr.io/${GOOGLE_CLOUD_PROJECT}/products:1.0.0
```

3. Expose the GKE container:

```
kubectl expose deployment products --type=LoadBalancer --port 80 --target-port
8082
```

4. Find the public IP of the Products services the same way you did for the Orders service:

```
kubectl get service products
```

```
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/microservices/src/products (qwiklabs-gcp-03-c894137e5528)$ kubectl get all
NAME                                 READY   STATUS    RESTARTS   AGE
pod/monolith-6b9df994ff-rm9ml        1/1     Running   0          3m30s
pod/orders-64bd7d97-x6ggh            1/1     Running   0          21m
pod/products-6866b7cd68-dw5ww        1/1     Running   0          12s

NAME                 TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE
service/kubernetes   ClusterIP      10.96.0.1     <none>          443/TCP        40m
service/monolith     LoadBalancer   10.96.11.77   35.230.62.224   80:31636/TCP   32m
service/orders       LoadBalancer   10.96.1.223   34.82.150.87    80:32559/TCP   16m
service/products     LoadBalancer   10.96.2.48    <pending>       80:32066/TCP   5s

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/monolith     1/1     1            1           32m
deployment.apps/orders       1/1     1            1           21m
deployment.apps/products     1/1     1            1           13s

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/monolith-69558fbc6      0         0         0       32m
replicaset.apps/monolith-6b9df994ff     1         1         1       3m31s
replicaset.apps/orders-64bd7d97         1         1         1       21m
replicaset.apps/products-6866b7cd68     1         1         1       13s
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/microservices/src/products (qwiklabs-gcp-03-c894137e5528)$
```

You will use the IP address in the next step when you reconfigure the monolith to point to your new Products microservice.

Reconfigure the monolith

1. Use the nano editor to replace the local URL with the IP address of the new Products microservices:

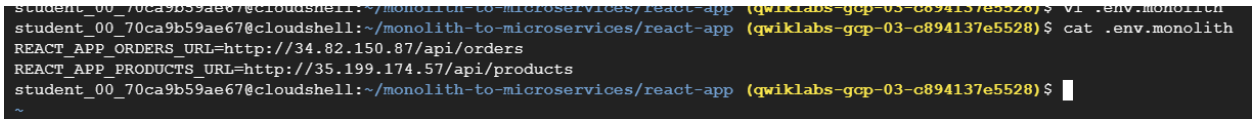
```
cd ~/monolith-to-microservices/react-app
nano .env.monolith
```

When the editor opens, your file should look like this:

```
REACT_APP_ORDERS_URL=http://<ORDERS_IP_ADDRESS>/api/orders
REACT_APP_PRODUCTS_URL=/service/products
```

2. Replace the REACT_APP_PRODUCTS_URL to the new format while replacing with your Product microservice IP address so it matches below:

```
REACT_APP_ORDERS_URL=http://<ORDERS_IP_ADDRESS>/api/orders
REACT_APP_PRODUCTS_URL=http://<PRODUCTS_IP_ADDRESS>/api/products
```

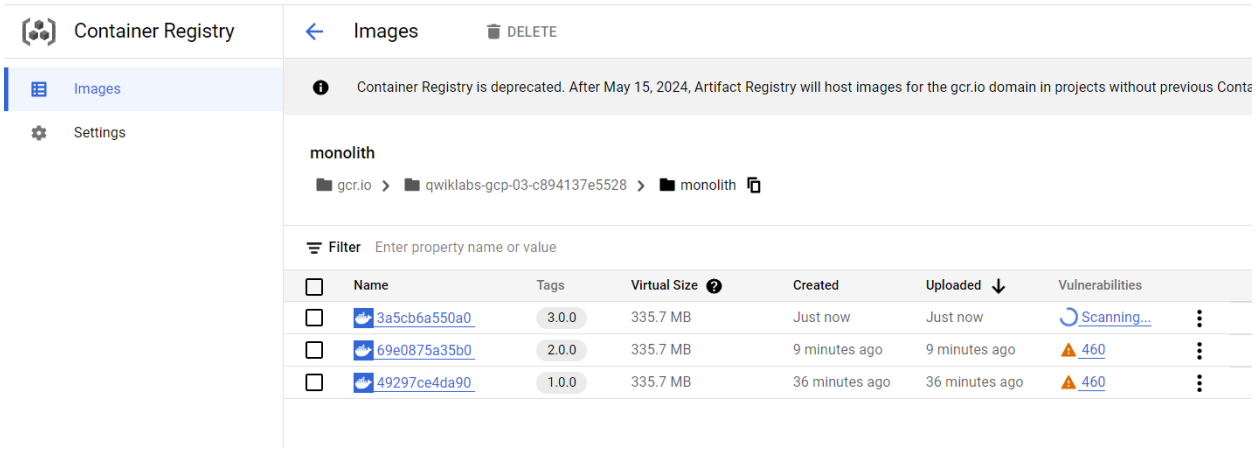


3. Press CTRL+O, press ENTER, then CTRL+X to save the file.
4. Test the new microservice by navigating the URL you just set in the file. The webpage should return a JSON response from the Products microservice.
5. Next, rebuild the monolith frontend and repeat the build process to build the container for the monolith and redeploy to the GKE cluster. Run the following commands complete these steps:
6. Rebuild Monolith Config Files:

```
npm run build:monolith
```

7. Create Docker Container with Cloud Build:

```
cd ~/monolith-to-microservices/monolith
gcloud builds submit --tag gcr.io/${GOOGLE_CLOUD_PROJECT}/monolith:3.0.0 .
```



8. Deploy Container to GKE:

```
kubectl set image deployment/monolith
monolith=gcr.io/${GOOGLE_CLOUD_PROJECT}/monolith:3.0.0
```

Copied!

content_copy

9. Verify your application is now hitting the new Products microservice by going to the monolith application in your browser and navigating to the Products page. All the product names should be prefixed by MS- as shown below:

Task 6. Migrate frontend to microservice

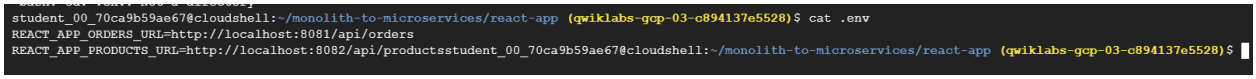
The last step in the migration process is to move the Frontend code to a microservice and shut down the monolith! After this step is completed, we will have successfully migrated our monolith to a microservices architecture!

Create a new frontend microservice

Follow the same procedure as the last two steps to create a new frontend microservice.

Previously when you rebuilt the monolith you updated the config to point to the monolith. Now you need to use the same config for the frontend microservice.

1. Run the following commands to copy the microservices URL config files to the frontend microservice codebase:




```
cd ~/monolith-to-microservices/react-app
cp .env.monolith .env
npm run build
```

2. Once that is completed, follow the same process as the previous steps. Run the following commands to build a Docker container, deploy your container and expose it to via a Kubernetes service.
3. Create Docker Container with Google Cloud Build:

```
cd ~/monolith-to-microservices/microservices/src/frontend
gcloud builds submit --tag gcr.io/${GOOGLE_CLOUD_PROJECT}/frontend:1.0.0 .
```

4. Deploy Container to GKE:

```
kubectl create deployment frontend
--image=gcr.io/${GOOGLE_CLOUD_PROJECT}/frontend:1.0.0
```

5. Expose GKE Container

```
kubectl expose deployment frontend --type=LoadBalancer --port 80 --target-port
8080
```

Delete the monolith

Now that all of the services are running as microservices, you can delete the monolith application! Note, in an actual migration, this would also entail DNS changes, etc to get our existing domain names to point to the new frontend microservices for our application.

- Run the following commands to delete the monolith:

```
kubectl delete deployment monolith
```

```
kubectl delete service monolith
```

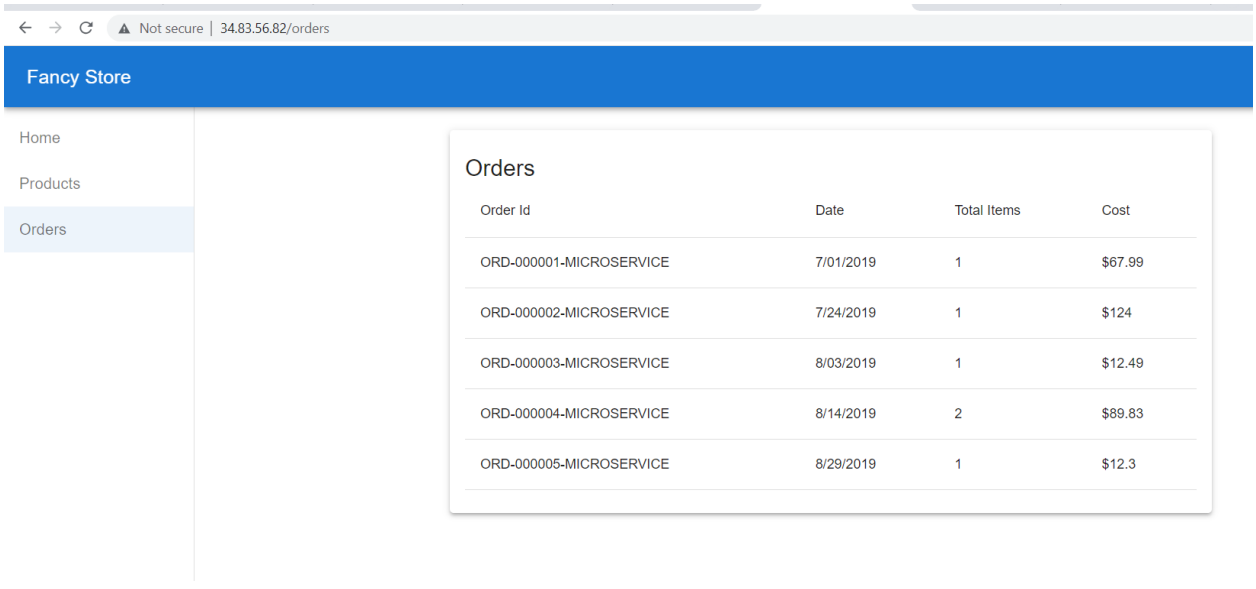
Test your work

To verify everything is working, your old IP address from your monolith service should not work now, and your new IP address from your frontend service should host the new application.

- To see a list of all the services and IP addresses, run the following command:

```
kubectl get services
```

Once you've determined the external IP address for your frontend microservice, copy the IP address. Point your browser to this URL (such as http://203.0.113.0) to check if your frontend is accessible. Your website should be the same as it was before you broke down the monolith into microservices!




```
service "monolith" deleted
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/microservices/src/frontend (qwiklabs-gcp-03-c894137e5528)$ kubectl get all
NAME                            READY   STATUS    RESTARTS   AGE
pod/frontend-7d9488fb68-644mg   1/1     Running   0          5m53s
pod/orders-64bd7d97-x6ggh       1/1     Running   0          37m
pod/products-6866b7cd68-dw5ww   1/1     Running   0          16m

NAME                 TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE
service/frontend     LoadBalancer   10.96.7.98    34.83.56.82     80:31246/TCP   5m42s
service/kubernetes   ClusterIP      10.96.0.1     <none>          443/TCP        56m
service/orders       LoadBalancer   10.96.1.223   34.82.150.87    80:32559/TCP   32m
service/products     LoadBalancer   10.96.2.48    35.199.174.57   80:32066/TCP   16m

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/frontend    1/1     1            1           5m53s
deployment.apps/orders      1/1     1            1           37m
deployment.apps/products    1/1     1            1           16m

NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/frontend-7d9488fb68   1         1         1       5m54s
replicaset.apps/orders-64bd7d97       1         1         1       37m
replicaset.apps/products-6866b7cd68   1         1         1       16m
student_00_70ca9b59ae67@cloudshell:~/monolith-to-microservices/microservices/src/frontend (qwiklabs-gcp-03-c894137e5528)$
```