Migrating a Monolithic Website to Microservices on Google Kubernetes Engine

Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

1. Click **Activate Cloud Shell** at the top of the Google Cloud console.

2. Click through the following windows:

   - Continue through the Cloud Shell information window.
   - Authorize Cloud Shell to use your credentials to make Google Cloud API calls.

When you are connected, you are already authenticated, and the project is set to your **Project_ID**, `qwiklabs-gcp-04-cc4fddbe3b40`. The output contains a line that declares the **Project_ID** for this session:

```
Your Cloud Platform project in this session is set to qwiklabs-gcp-04-
cc4fddbe3b40
```

`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

3. (Optional) You can list the active account name with this command:

```
gcloud auth list
```

Copied!

content_copy

4. Click **Authorize**.

Output:

```
ACTIVE: *
ACCOUNT: student-01-3114ebfd23b7@qwiklabs.net

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

5. (Optional) You can list the project ID with this command:

```
gcloud config list project
```

Copied!

content_copy

Output:

```
[core]
project = qwiklabs-gcp-04-cc4fddbe3b40
```

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
```

Copied!

content_copy

It may take a few minutes for this script to run.

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
```

Copied!

content_copy

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
```

Copied!

content_copy

Warning: If you get an error about region/zone not being specified, please see the environment set up section to make sure you set the default compute zone.

It may take several minutes for the cluster to be created.

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

```
gcloud compute instances list
```

Copied!

content_copy

Output:

```
NAME                                          ZONE        MACHINE_TYPE
PREEMPTIBLE  INTERNAL_IP  EXTERNAL_IP    STATUS
gke-fancy-cluster-default-pool-ad92506d-1ng3  us-west1-a  e2-standard-4
10.150.0.7   XX.XX.XX.XX    RUNNING
gke-fancy-cluster-default-pool-ad92506d-4fvq  us-west1-a  e2-standard-4
10.150.0.5   XX.XX.XX.XX    RUNNING
gke-fancy-cluster-default-pool-ad92506d-4zs3  us-west1-a  e2-standard-4
10.150.0.6   XX.XX.XX.XX    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***.

Congratulations! You have just created your first Kubernetes 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
```

Copied!

content_copy

Accessing the monolith

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

```
kubectl get service monolith
```

Copied!

content_copy

You should see output similar to the following:

```
NAME          CLUSTER-IP      EXTERNAL-IP     PORT(S)         AGE
monolith      10.3.251.122    203.0.113.0     80:30877/TCP    3d
```

2. If your output lists the external IP as `<pending>` give it a minute and run the command again.

3. 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.

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 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 Artifact Registry with a single command!

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 Artifact Registry.

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

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

Copied!

content_copy

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

```
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
ID                                    CREATE_TIME
DURATION  SOURCE
IMAGES                               STATUS
1ae295d9-63cb-482c-959b-bc52e9644d53  2019-08-29T01:56:35+00:00  33S
gs://<project_id>_cloudbuild/source/1567043793.94-
abfd382011724422bf49af1558b894aa.tgz  gcr.io/<project_id>/orders:1.0.0
SUCCESS
</project_id></project_id>
```

2. To view your build history, or watch the process in real time, in the console, search for **Cloud Build** then click on the **Cloud Build** result.

3. On the **History** page you can see a list of all your 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.

4. From the build details page, to view the container image that was created, 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 Artifact 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
  ```

  Copied!

  content_copy

  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 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
```
Copied!

content_copy

It may take a few moments for the pod status to be Running.

Output:

```
NAME                              READY    STATUS     RESTARTS    AGE
pod/monolith-779c8d95f5-dxnzl     1/1      Running    0           15h
pod/orders-5bc6969d76-kdxkk       1/1      Running    0           21s
NAME                    TYPE           CLUSTER-IP       EXTERNAL-IP
PORT(S)         AGE
service/kubernetes      ClusterIP      10.39.240.1      <none>
443/TCP         19d
service/monolith        LoadBalancer   10.39.241.130    34.74.209.57
80:30412/TCP    15h
NAME                          READY    UP-TO-DATE    AVAILABLE    AGE
deployment.apps/monolith      1/1      1             1            15h
deployment.apps/orders        1/1      1             1            21s
NAME                                     DESIRED    CURRENT    READY    AGE
replicaset.apps/monolith-779c8d95f5      1          1          1        15h
replicaset.apps/orders-5bc6969d76        1          1          1        21s
</none>
```

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

You have deployed our application on GKE, but 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.
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.

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
  ```
  Copied!

  content_copy

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

- To find out the external IP that GKE provisioned for your application, inspect the Service with the `kubectl get service` command:

```
kubectl get service orders
```

Copied!

content_copy

Output:

```
NAME          CLUSTER-IP      EXTERNAL-IP      PORT(S)           AGE
orders        10.3.251.122    203.0.113.0      80:30877/TCP      3s
```

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 to 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
```

Copied!

content_copy

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
```

Copied!

content_copy

3. Press `CTRL+O`, press `ENTER`, then `CTRL+X` to save the file in the nano editor.

4. 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.

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

```
npm run build:monolith
```
Copied!

content_copy

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 .
```
Copied!

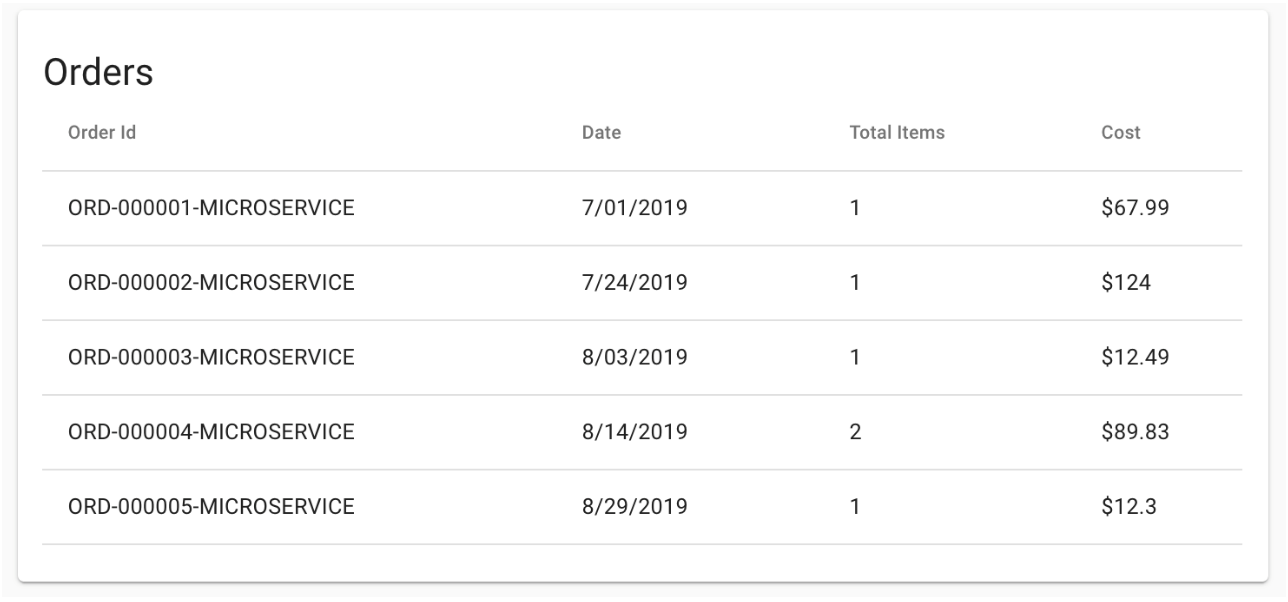
content_copy

7. Deploy container to GKE:

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

content_copy

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:



Orders

Order Id	Date	Total Items	Cost
ORD-000001-MICROSERVICE	7/01/2019	1	$67.99
ORD-000002-MICROSERVICE	7/24/2019	1	$124
ORD-000003-MICROSERVICE	8/03/2019	1	$12.49
ORD-000004-MICROSERVICE	8/14/2019	2	$89.83
ORD-000005-MICROSERVICE	8/29/2019	1	$12.3

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 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 .
```

Copied!

content_copy

2. Deploy container to GKE:

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

Copied!

content_copy

3. Expose the GKE container:

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

Copied!

content_copy

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

```
kubectl get service products
```

Copied!

content_copy

Output:

```
NAME          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
products      10.3.251.122    203.0.113.0      80:30877/TCP     3d
```

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
```
Copied!

content_copy

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
```
Copied!

content_copy

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
```
Copied!

content_copy

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 .
```
Copied!

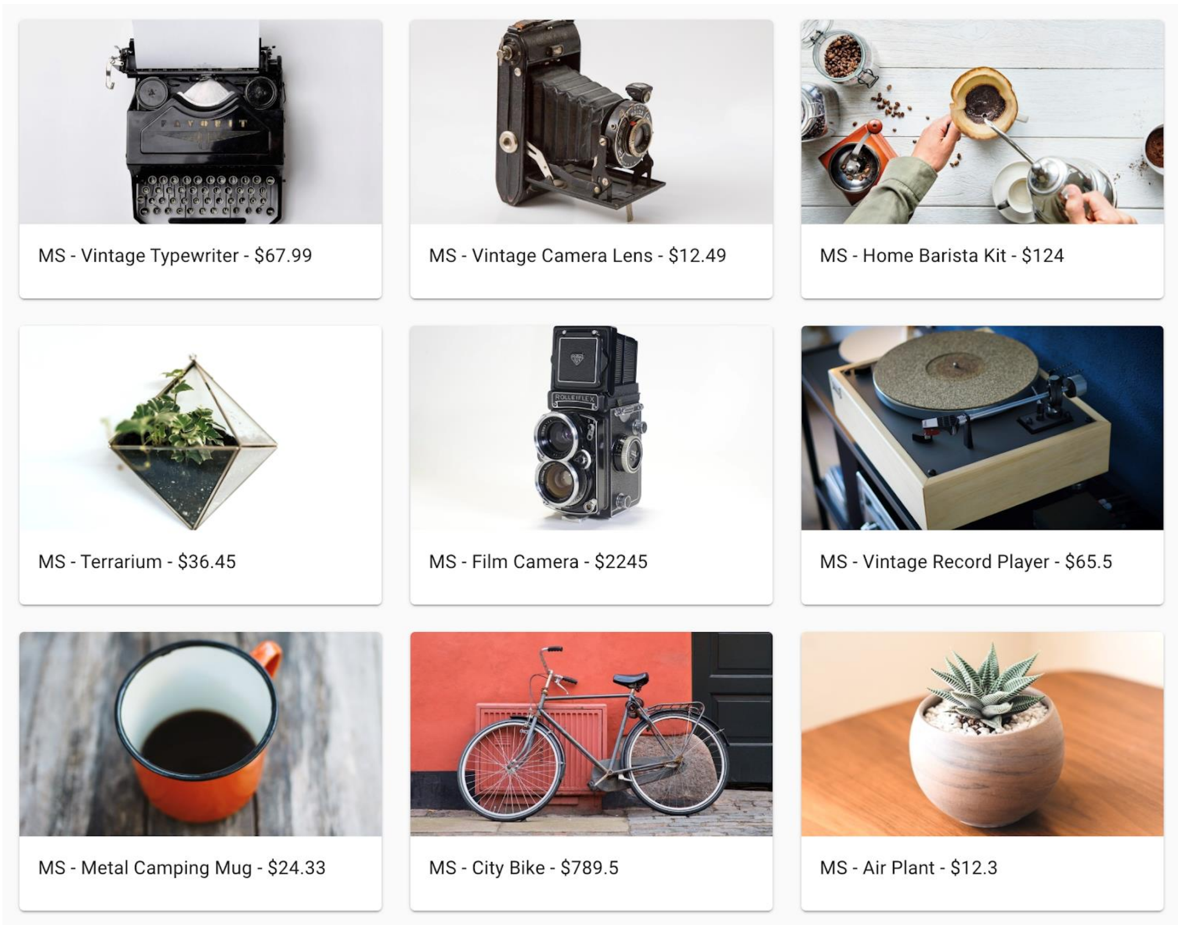
content_copy

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:


MS - Vintage Typewriter - $67.99
MS - Vintage Camera Lens - $12.49
MS - Home Barista Kit - $124
MS - Terrarium - $36.45
MS - Film Camera - $2245
MS - Vintage Record Player - $65.5
MS - Metal Camping Mug - $24.33
MS - City Bike - $789.5
MS - Air Plant - $12.3

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, you will have successfully migrated the 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
```

Copied!

content_copy

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 .
```

Copied!

content_copy

4. Deploy container to GKE:

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

Copied!

content_copy

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, delete the monolith application! In an actual migration, this would also entail DNS changes, etc., to get the existing domain names to point to the new frontend microservices for the application.

- Run the following commands to delete the monolith:

```
kubectl delete deployment monolith
kubectl delete service monolith
```

Copied!

content_copy

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
  ```

  Copied!

  content_copy

  Your output should look similar to the following:

  ```
  NAME            TYPE            CLUSTER-IP        EXTERNAL-IP        PORT(S)
  AGE
  frontend        LoadBalancer    10.39.246.135     35.227.21.154
  80:32663/TCP    12m
  kubernetes      ClusterIP       10.39.240.1       <none>             443/TCP
  18d
  orders          LoadBalancer    10.39.243.42      35.243.173.255
  80:32714/TCP    31m
  products        LoadBalancer    10.39.250.16      35.243.180.23
  80:32335/TCP    21m
  </none>
  ```

  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!