

# Migrating a Monolithic Website to Microservices on Google Kubernetes Engine

**GSP699**



Google Cloud Self-Paced Labs

# Introduction

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.

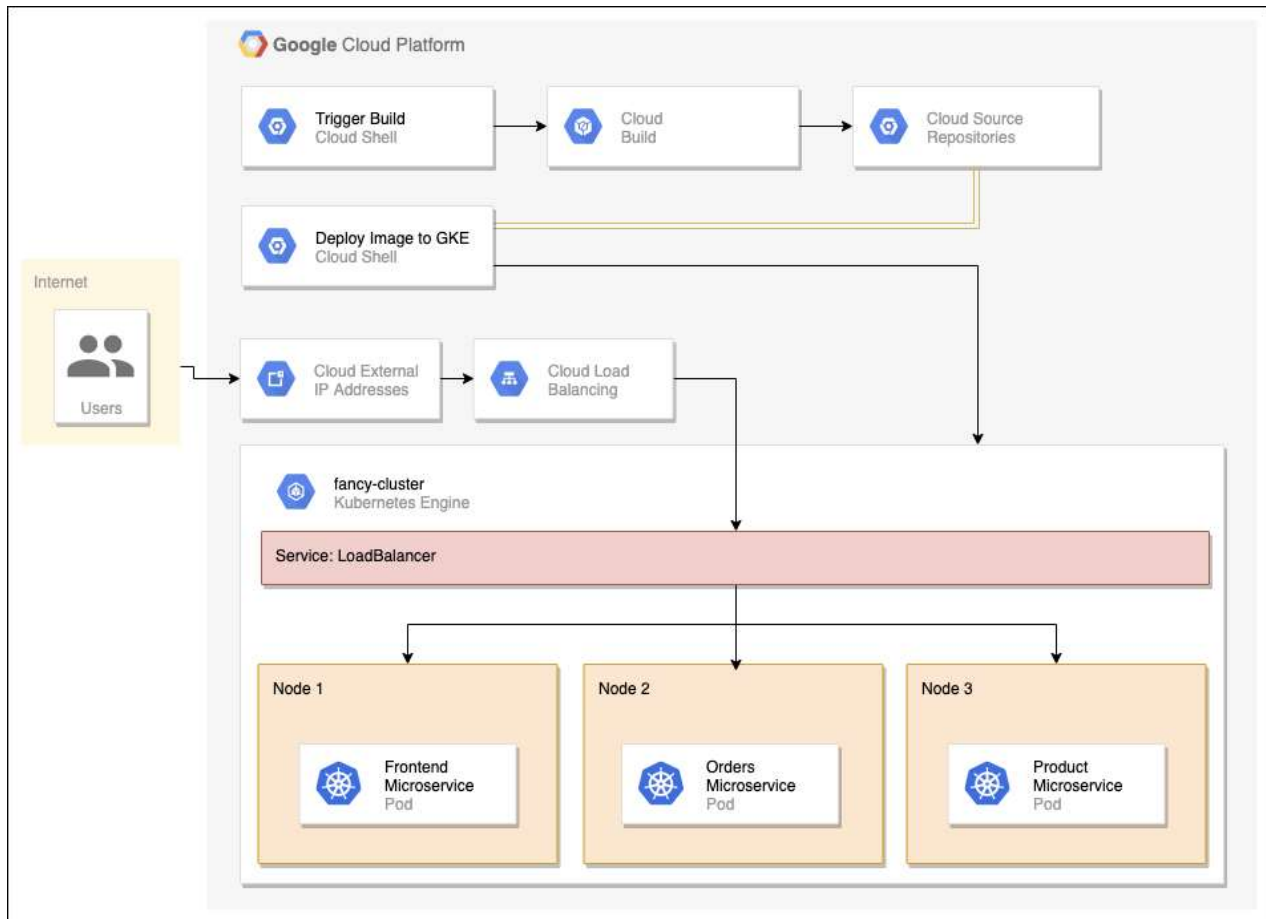
Some of the disadvantages when compared to monoliths are:

- 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

## Prerequisites

- A Google Cloud Platform account with administrative access to create projects or a project with Project Owner role
- A basic understanding of Docker and Kubernetes

# Environment Setup

## Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

## What you need

To complete this lab, you need:

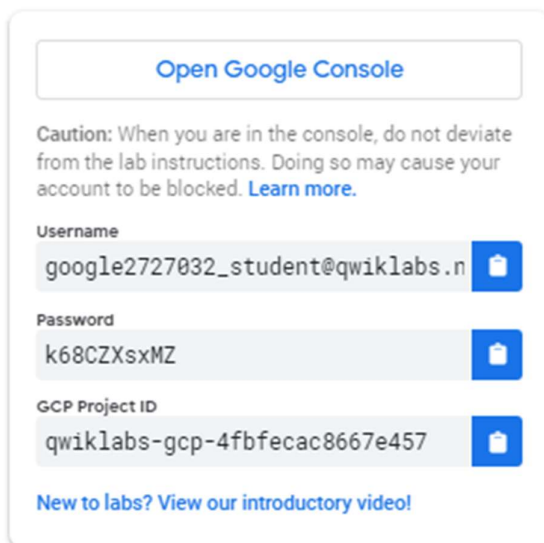
- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.

**Note:** If you already have your own personal Google Cloud account or project, do not use it for this lab.

**Note:** If you are using a Pixelbook, open an Incognito window to run this lab.

## How to start your lab and sign in to the Google Cloud Console

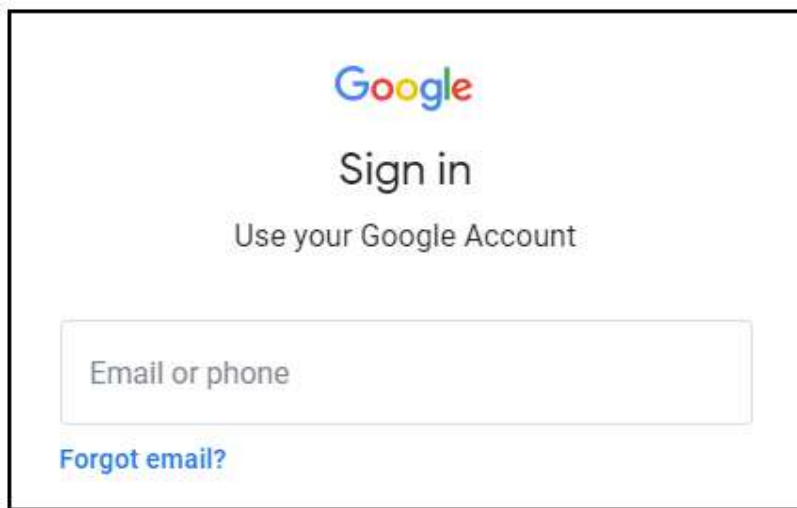
1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.



The screenshot shows a sign-in panel with the following elements:

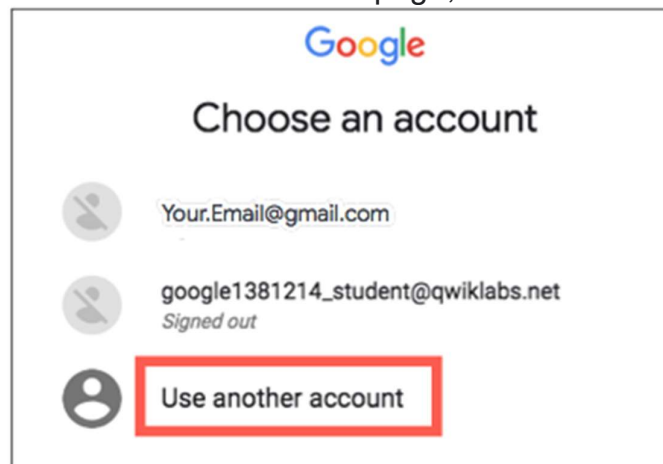
- A button at the top labeled "Open Google Console".
- A caution message: "Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. [Learn more.](#)"
- Three input fields, each with a copy icon to its right:
  - Username:** google2727032\_student@qwiklabs.n
  - Password:** k68CZXsxMZ
  - GCP Project ID:** qwiklabs-gcp-4fbfecac8667e457
- A link at the bottom: "New to labs? [View our introductory video!](#)"

2. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.



**Tip:** Open the tabs in separate windows, side-by-side.

If you see the **Choose an account** page, click **Use Another**



**Account.**

3. In the **Sign in** page, paste the username that you copied from the Connection Details panel. Then copy and paste the password.

**Important:** You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

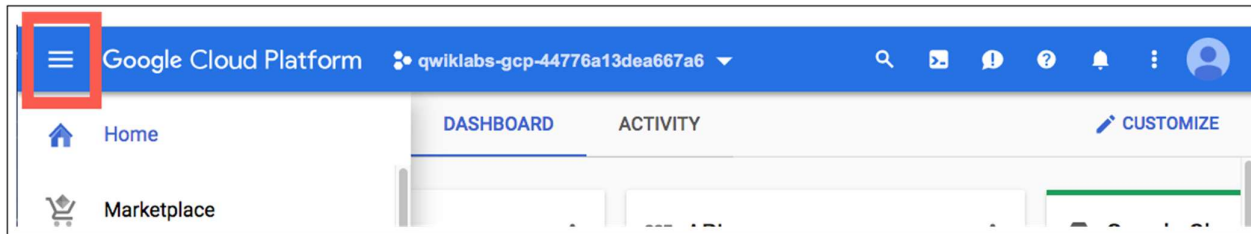
4. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

After a few moments, the Cloud Console opens in this tab.

**Note:** You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-

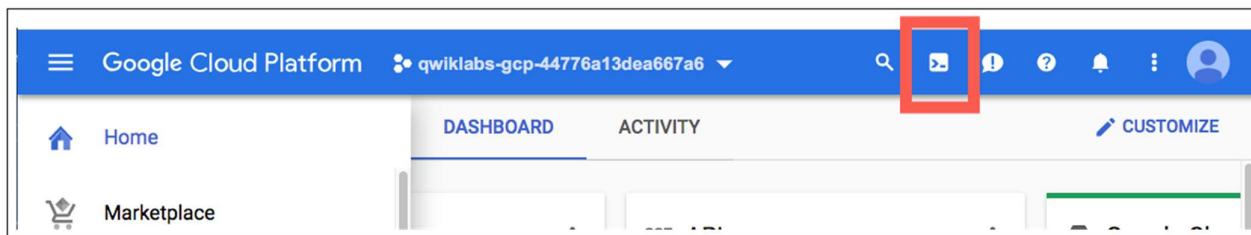
left.



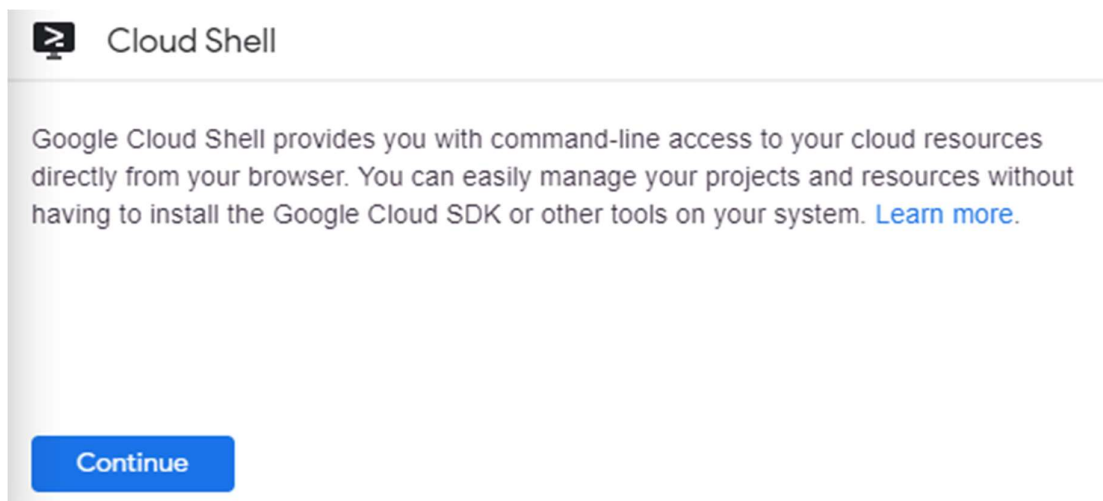
## Activate Cloud Shell

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.

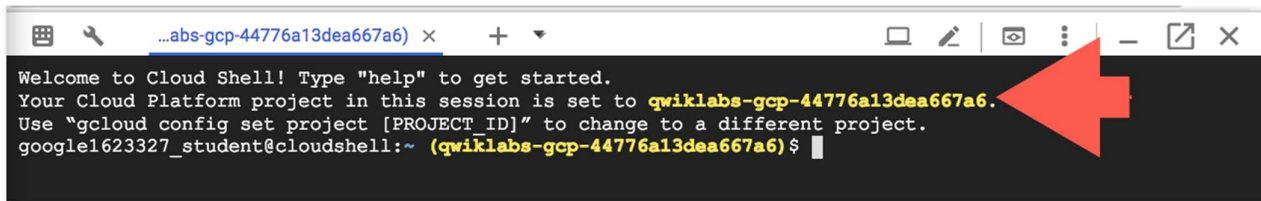
In the Cloud Console, in the top right toolbar, click the **Activate Cloud Shell** button.



Click **Continue**.



It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your *PROJECT\_ID*. For example:



```
...abs-gcp-44776a13dea667a6) x + ▾
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to qwiklabs-gcp-44776a13dea667a6.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
google1623327_student@cloudshell:~ (qwiklabs-gcp-44776a13dea667a6) $
```

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

You can list the active account name with this command:

```
gcloud auth list
```

(Output)

```
Credentialed accounts:
- <myaccount>@<mydomain>.com (active)
```

(Example output)

```
Credentialed accounts:
- google1623327_student@qwiklabs.net
```

You can list the project ID with this command:

```
gcloud config list project
```

(Output)

```
[core]
project = <project ID>
```

(Example output)

```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

For full documentation of `gcloud` see the [gcloud command-line tool overview](#).

Set the default zone and project configuration:

```
gcloud config set compute/zone us-central1-f
```

# Clone 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
```

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



# 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.

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

```
gcloud services enable container.googleapis.com
```

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

```
gcloud container clusters create fancy-cluster --num-nodes 3
```

## 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.

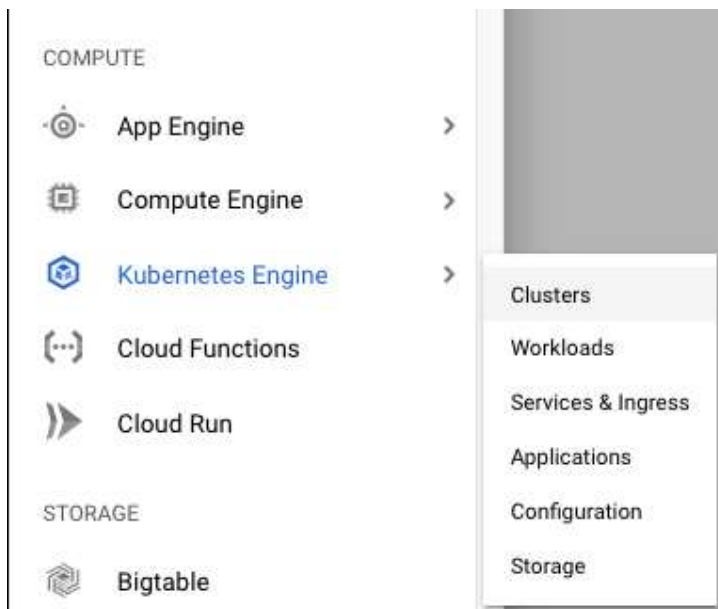
Once the command has completed, run the following to see the cluster's three worker VM instances:

```
gcloud compute instances list
```

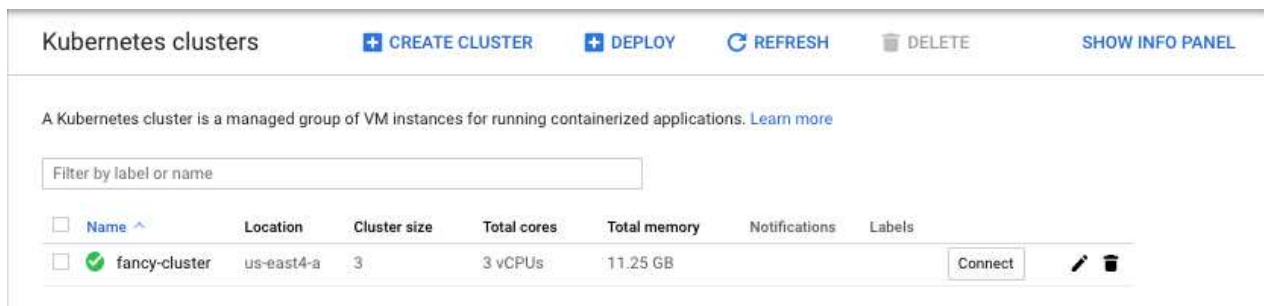
Output:

NAME	INTERNAL_IP	EXTERNAL_IP	STATUS	ZONE	MACHINE_TYPE	PREEMPTIBLE
gke-fancy-cluster-default-pool-ad92506d-1ng3	10.150.0.7	XX.XX.XX.XX	RUNNING	us-central1-f	n1-standard-1	
gke-fancy-cluster-default-pool-ad92506d-4fvq	10.150.0.5	XX.XX.XX.XX	RUNNING	us-central1-f	n1-standard-1	
gke-fancy-cluster-default-pool-ad92506d-4zs3	10.150.0.6	XX.XX.XX.XX	RUNNING	us-central1-f	n1-standard-1	

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!

Click *Check my progress* to verify the objective.

# Deploy 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
```

## Accessing the monolith

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

```
kubectl get service monolith
```

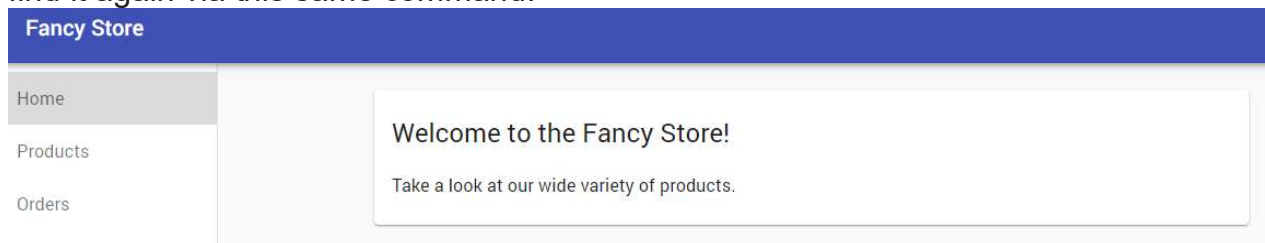
You should see output similar to the following:

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

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

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.

Remember this IP address as you will continue to use it going forward. You can always find it again via this same command.



You should see the welcome page for the monolithic website just like the picture above. 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!

Click *Check my progress* to verify the objective.

# 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 view the manual process of creating a docker file and pushing it you can go [here](#).

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.

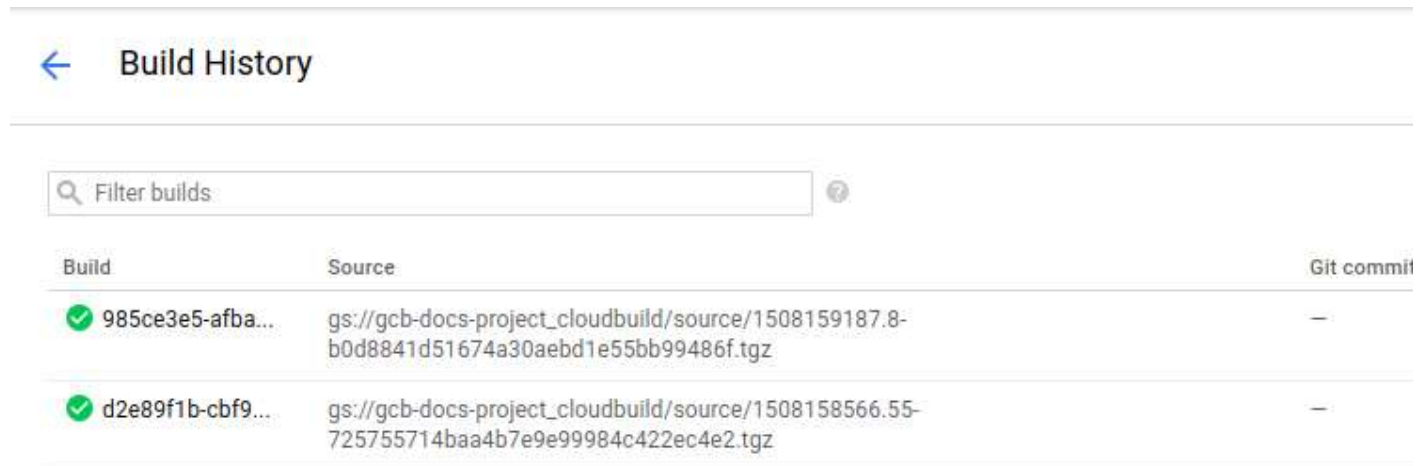
Run the following commands to build your Docker container and push it to the Google Container Registry:

```
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:

```
-----
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
```

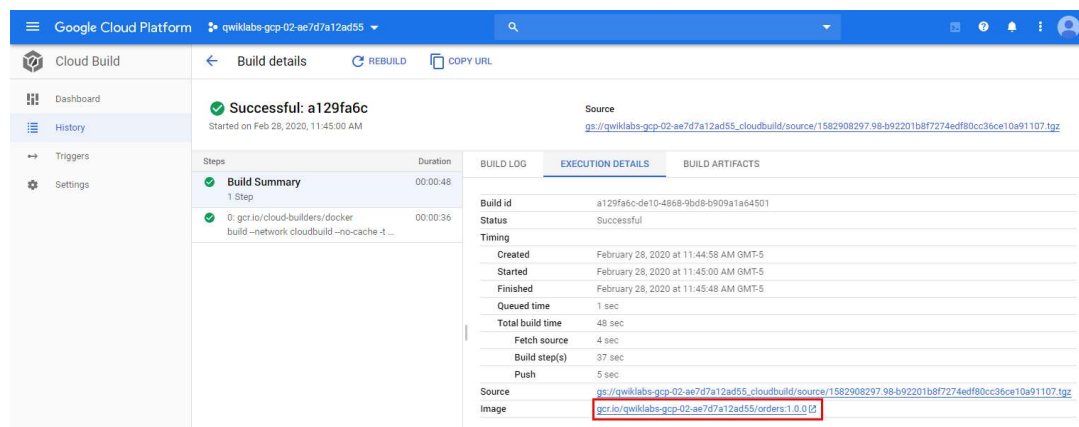
To view your build history or watch the process in real time, in the Console click the **Navigation Menu** button on the top left and scroll down to Tools and click **Cloud Build > History**. Here you can see a list of all your previous builds, there should only be 1 that you just created.



Build	Source	Git commit
985ce3e5-afba...	gs://gcb-docs-project_cloudbuild/source/1508159187.8-b0d8841d51674a30aebd1e55bb99486f.tgz	—
d2e89f1b-cbf9...	gs://gcb-docs-project_cloudbuild/source/1508158566.55-725755714baa4b7e9e99984c422ec4e2.tgz	—

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.



Steps	Duration
<b>Build Summary</b> 1 Step	00:00:48
0: gcr.io/cloud-builders/docker build --network cloudbuild --no-cache -t ...	00:00:36

	BUILD LOG	EXECUTION DETAILS	BUILD ARTIFACTS
Build id	a129fa6c-de10-4868-9bd8-b909a1a64501		
Status	Successful		
Timing			
Created	February 28, 2020 at 11:44:58 AM GMT-5		
Started	February 28, 2020 at 11:45:00 AM GMT-5		
Finished	February 28, 2020 at 11:45:48 AM GMT-5		
Queued time	1 sec		
Total build time	48 sec		
Fetch source	4 sec		
Build step(s)	37 sec		
Push	5 sec		
Source	gs://wikilabs-gcp-02-ae7d7a12ad55_cloudbuild/source/1582908297_98-b92201b8f7274edf80cc36ce10a91107.tgz		
Image	gcr.io/wikilabs-gcp-02-ae7d7a12ad55/orders:1.0.0		

## 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
```

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. See these resources for more information: [Kubernetes Deployments](#).

## Verify Deployment

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

```
kubectl get all
```

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

Output:

NAME	READY	STATUS	RESTARTS	AGE
pod/monolith-779c8d95f5-dxnz1	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

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](#) ) 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. Read more [here](#) 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
```

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 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. 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
```

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
```

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

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.

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
```

Create Docker Container with Cloud Build:

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

Deploy Container to GKE:

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

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

Click *Check my progress* to verify the objective.



# 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.

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 .
```

Deploy Container to GKE:

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

Expose the GKE container:

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

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

```
kubectl get service products
```

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 Monolith

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
```

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
```

Press `CTRL+O`, press `ENTER`, then `CTRL+X` to save the file.

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.

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:

Rebuild Monolith Config Files:

```
npm run build:monolith
```

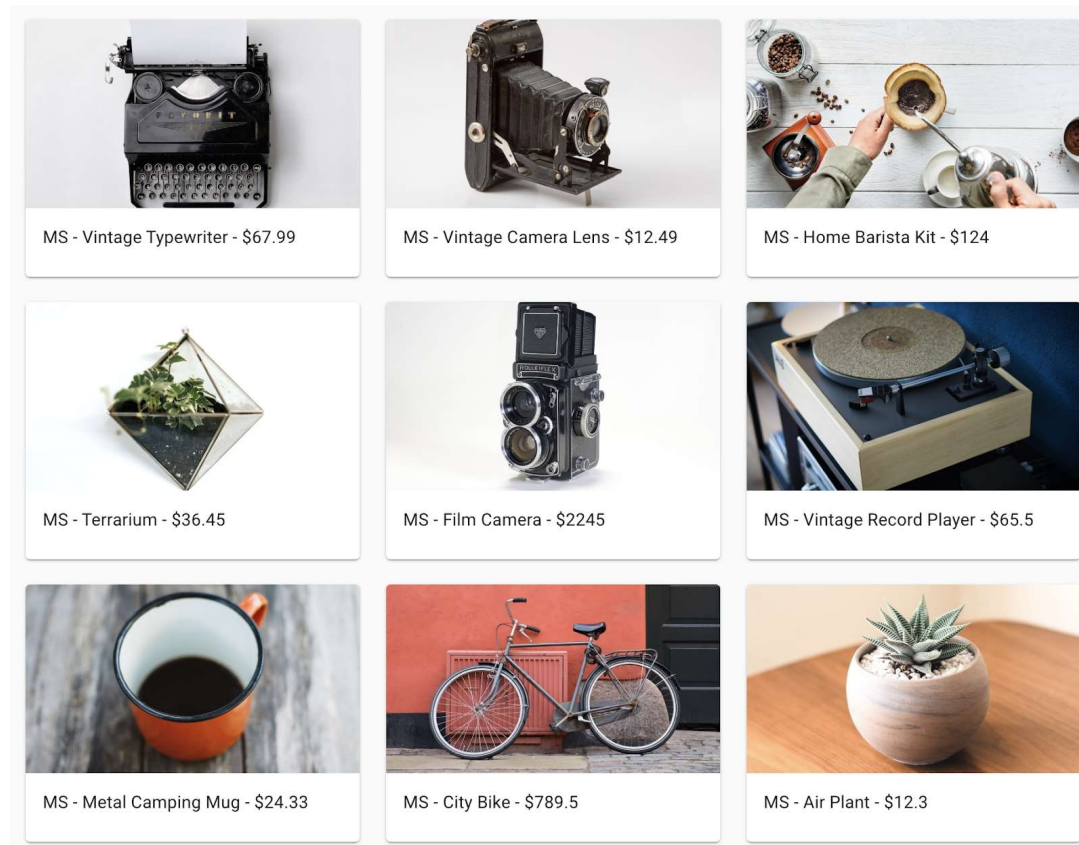
Create Docker Container with Cloud Build:

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

Deploy Container to GKE:

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

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:



Click *Check my progress* to verify the objective.

# 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 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.

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
```

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 via a Kubernetes service.

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 .
```

Deploy Container to GKE:

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

Expose GKE Container

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

Click *Check my progress* to verify the objective.

## 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
```

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

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!

# Congratulations!

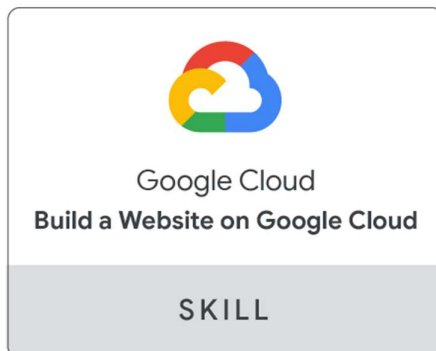
You successfully broke down your monolithic application into microservices and deployed them on Google Kubernetes Engine!



## Finish Your Quest

This self-paced lab is part of the Qwiklabs [Website on Google Cloud](#) Quest. A Quest is a series of related labs that form a learning path. Enroll in this Quest and get immediate completion credit if you've taken this lab. [See other available Qwiklabs Quests.](#)

Looking for a hands-on challenge lab to demonstrate your skills and validate your knowledge? On completing this quest, finish this additional [challenge lab](#) to receive an exclusive Google Cloud digital badge.



## Take your next lab

Continue your learning by watching this video case study on [Hosting Scalable Web Applications on Google Cloud](#) or check out these suggestions:

- [Hosting a Web App on Google Cloud Using Compute Engine](#)
- Watch this video on [Hosting a Static Web App on Google Cloud](#)

## Next Steps / Additional Resources

- Docker - <https://docs.docker.com/>
- Kubernetes - <https://kubernetes.io/docs/home/>
- Google Kubernetes Engine (GKE) - <https://cloud.google.com/kubernetes-engine/docs/>
- Google Cloud Build - <https://cloud.google.com/cloud-build/docs/>
- Google Container Registry - <https://cloud.google.com/container-registry/docs/>
- Migrating Monoliths to Microservices - <https://cloud.google.com/solutions/migrating-a-monolithic-app-to-microservices-gke>
- [Microservices Best Practices](#)