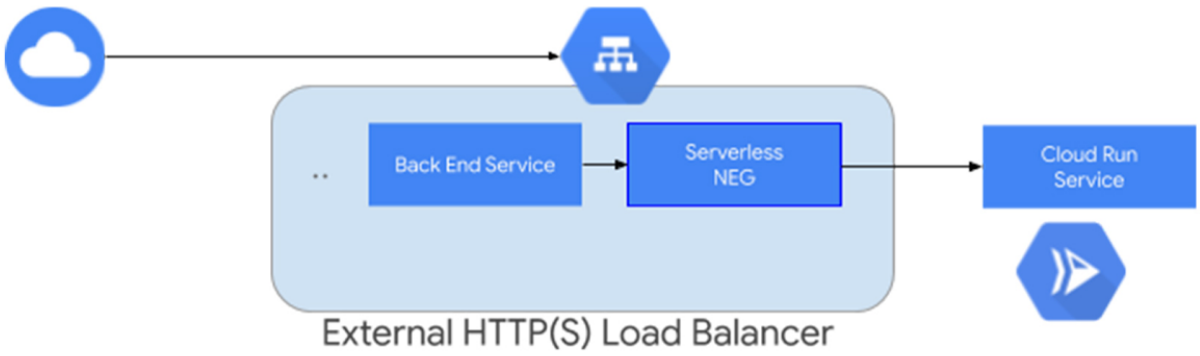
Using a Global Load Balancer with Cloud Run

Overview



Serverless Network Endpoint Groups (NEGs) allow you to use Google Cloud serverless apps with external HTTP(S) Load Balancing. After you have configured a load balancer

with the serverless NEG backend, requests to the load balancer are routed to the serverless app backend.

In this lab you will learn how to set up and use an HTTP global load balancer with Cloud Run.

Objectives

In this lab, you learn to:

- Enable the Cloud Run API.
- Create and deploy a sample application in Cloud Run.
- Configure a Serverless Network Endpoint Group (NEG).
- Create a global load balancer with a serverless NEG backend.
- Route requests for the sample application through the global load balancer.

Prerequisites

These labs are based on intermediate knowledge of Google Cloud. While the steps required are covered in the content, it would be helpful to have familiarity with any of the following products:

- HTTP load balancer
- Cloud Run

Basic Linux Commands

Below you will find a reference list of a few very basic Linux commands which may be included in the instructions or code blocks for this lab.

Command -->	**Action**	**.**	**Command -->**	**Action**
mkdir (*make directory*)	create a new folder	.	**cd** (*change directory*)	change location to another folder
ls (*list*)	list files and folders in the directory	.	**cat** (*concatenate*)	read contents of a file without using an editor
apt-get update	update package manager library	.	**ping**	signal to test reachability of a host
mv (*move*)	moves a file	.	**cp** (*copy*)	makes a file copy
pwd (*present working directory*)	returns your current location	.	**sudo** (*super user do*)	gives higher administration privileges

Task 1. Configure the environment

Cloud Run requires some environmental configuration before we begin. In this section, you enable the Cloud Run API and set the compute region.

1. Enable Cloud Run API:

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

If you are asked to authorize the use of your credentials, do so.

You should then see a successful message similar to this one:

```
Operation "operations/acf.cc11852d-40af-47ad-9d59-477a12847c9e" finished successfully.
```

Note: You can also enable the API using the **APIs & Services** section of the console.

2. Set the compute and run regions. Cloud Run requires knowledge of the region in which it will be deployed:

```
gcloud config set compute/region us-central1
gcloud config set run/region us-central1
```

3. Create a LOCATION environment variable:

```
LOCATION="us-central1"
```

This lab also requires the `gcloud` command line tool for Google Cloud Platform. However, the `gcloud` command line tool comes pre-installed on Cloud Shell so we will not need to configure it in this lab.

Task 2. Write a simple application

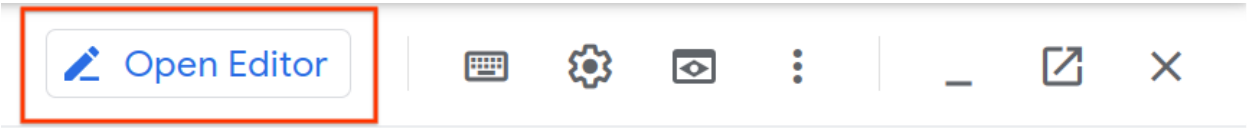
For this lab, we will create a simple Cloud Run Python "Hello, World" app and deploy it as a Cloud Run service in the us-central1 region. This simple app will become the target for an external HTTP load balancer which will use a serverless NEG backend to route requests to that service.

1. In Cloud Shell create a new directory named `helloworld`, then move your view into that directory:

```
mkdir helloworld && cd helloworld
```

Next you'll be creating and editing files.

2. To edit files, use `vi`, `emac`, `nano` or the Cloud Shell Editor by clicking on the pencil icon in Cloud Shell ("Open Editor").



When you open the Cloud Shell Editor, it will show your user's home directory, NOT the directory you are seeing in the terminal.

3. Be sure to click on the "helloworld" folder to be certain you are in the correct directory before you create the following two files.
4. Create a `main.py` file, then add the following content to it:

```
import os
from flask import Flask
app = Flask(__name__)
```

```
@app.route("/")
def hello_world():
    name = os.environ.get("NAME", "World")
    return "Hello {}!".format(name)
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=int(os.environ.get("PORT",
8080")))
```

This code responds to requests on port 8080 with a "Hello World" greeting.

Your app is now finished and ready to be containerized and uploaded to Container Registry.

Note: You can use languages other than Python to get started with Cloud Run. You can find instructions for Go, Python, Java, PHP, Ruby, Shell scripts, and others in the Cloud Run documentation.

Task 3. Containerize your app and upload it to Container Registry (Artifact Registry)

1. To containerize the sample app you just made, create a new file named `Dockerfile` in the same directory as the `main.py` file, and add the following content:

```
# Use the official lightweight Python image.
# https://hub.docker.com/_/python
FROM python:3.9-slim
```

```
# Allow statements and log messages to immediately appear in the Knative
logs
    ENV PYTHONUNBUFFERED True
# Copy local code to the container image.
ENV APP_HOME /app
WORKDIR $APP_HOME
COPY . ./
# Install production dependencies.
RUN pip install Flask gunicorn
# Run the web service on container startup. Here we use the gunicorn
# webserver, with one worker process and 8 threads.
# For environments with multiple CPU cores, increase the number of workers
# to be equal to the cores available.
# Timeout is set to 0 to disable the timeouts of the workers to allow
Cloud Run to handle instance scaling.
CMD exec gunicorn --bind :$PORT --workers 1 --threads 8 --timeout 0
main:app
```

This Python Dockerfile starts a Gunicorn web server which listens on the port defined by the `PORT` environmental variable set in the main.py file (port 8080).

2. Now, build your container image using Cloud Build, by running the following command **from the directory containing the Dockerfile**:

```
gcloud builds submit --tag gcr.io/$GOOGLE_CLOUD_PROJECT/helloworld
```

Upon success, you will see a SUCCESS message containing the image name (`gcr.io/PROJECT-ID/helloworld`). The image is now stored in Container Registry and can be deployed in the next task.

Note: You can ignore any warning while building the container image using Cloud Build.

Task 4. Deploy your container to Cloud Run

Now that we have our containerized image, we will need to use the `gcloud` command below to deploy it to Cloud Run.

1. Replace PROJECT-ID with your GCP project ID. You can view your project ID by running the command `gcloud config get-value project.`

```
gcloud run deploy --image gcr.io/$GOOGLE_CLOUD_PROJECT/helloworld
```

2. The API should already be enabled, but if you are prompted to enable the API, Reply `y`.
3. You will then be prompted for the service name: press **Enter** to accept the default name, `helloworld`.
4. If prompted for region: in this case, select us-central1.
5. You will be prompted to allow unauthenticated invocations: respond `y`.

Wait a few moments until the deployment is complete. Upon success, the command line displays the service URL.

6. Visit your deployed container by opening the service URL in a web browser.

Task 5. Reserve an external IP address

Now that your services are up and running, you need to set up a global static external IP address that your customers use to reach your load balancer.

A static external IP address provides a single address to point your serverless app to. Reserving an IP address would also be essential if you were using a custom domain for your serverless app.

1. Use the following command to reserve your static IP address:

```
gcloud compute addresses create example-ip \
    --ip-version=IPV4 \
    --global
```

2. To display the IP address that was just reserved, use:

```
gcloud compute addresses describe example-ip \
    --format="get(address)" \
    --global
```

3. Note this IP address, because you will need it later.

Task 6. Create the external HTTP load balancer

Load balancers use a serverless Network Endpoint Group (NEG) backend to direct requests to a serverless Cloud Run service.
So, let's first create our serverless NEG for our serverless Python app created earlier in this lab.

1. To create a serverless NEG with a Cloud Run service, enter the following in Cloud Shell:

```
gcloud compute network-endpoint-groups create myneg \
   --region=$LOCATION \
   --network-endpoint-type=serverless  \
   --cloud-run-service=helloworld
```

2. Now, we need to create a backend service:

```
gcloud compute backend-services create mybackendservice \
    --global
```

3. And then add the serverless NEG as a backend to this backend service:

```
gcloud compute backend-services add-backend mybackendservice \
    --global \
    --network-endpoint-group=myneg \
    --network-endpoint-group-region=$LOCATION
```

4. Finally, create a URL map to route incoming requests to the backend service:

```
gcloud compute url-maps create myurlmap \
    --default-service mybackendservice
```

If you had more than one backend service, you could use host rules to direct requests to different services based on the host name, or you could set up path matchers to direct requests to different services based on the request path. None of this is necessary here because we have only created one backend service for this lab.

5. Now, create a target HTTP(S) proxy to route requests to your URL map:

```
gcloud compute target-http-proxies create mytargetproxy \
    --url-map=myurlmap
```

6. Create a global forwarding rule to route incoming requests to the proxy:

```
gcloud compute forwarding-rules create myforwardingrule \
    --address=example-ip \
    --target-http-proxy=mytargetproxy \
    --global \
    --ports=80
```

Task 7. Test your external HTTP load balancer

- You can test your HTTP load balancer by going to `http://IP_ADDRESS`, where `IP_ADDRESS` is the load balancer's IP address you reserved earlier in this lab. When you open this URL, you should see the helloworld service homepage.

Congratulations!

Over this course of this lab, you have learned how to use an HTTP global load balancer with a Cloud Run application.

- Deploy a simple Cloud Run service
- Expose this service to the internet using an external IP address
- Create a global HTTP load balancer for the service