Deploying a Python Flask Web Application to App Engine Flexible

[favorite\_border](https://www.qwiklabs.com/favorite.json?item_id=3339&item_type=CatalogItem)

Add to favorites

help

—/100

Checkpoints

arrow\_forward

Authenticate API Requests

Check my progress

/ 25

Create an App Engine instance and Storage Bucket

Check my progress

/ 25

Run the Application

Check my progress

/ 25

Deploy the App

Check my progress

/ 25

Start Lab

01:00:00

[GSP023](https://www.qwiklabs.com/focuses/3339?parent=catalog#step1)[Overview](https://www.qwiklabs.com/focuses/3339?parent=catalog#step2)[Setup and Requirements](https://www.qwiklabs.com/focuses/3339?parent=catalog#step3)[Get the sample code](https://www.qwiklabs.com/focuses/3339?parent=catalog#step4)[Authenticate API Requests](https://www.qwiklabs.com/focuses/3339?parent=catalog#step5)[Testing the Application Locally](https://www.qwiklabs.com/focuses/3339?parent=catalog#step6)[Exploring the Code](https://www.qwiklabs.com/focuses/3339?parent=catalog#step7)[Deploying the App to App Engine Flexible](https://www.qwiklabs.com/focuses/3339?parent=catalog#step8)[Congratulations!](https://www.qwiklabs.com/focuses/3339?parent=catalog#step9)

Deploying a Python Flask Web Application to App Engine Flexible

1 hour5 Credits

Rate Lab

**GSP023**



**Overview**

**Summary**

In this lab, you will learn how to deploy a Python Flask web application to the App Engine Flexible environment. The example application allows a user to upload a photo of a person's face and learn how likely it is that the person is happy. The application uses Google Cloud APIs for Vision, Storage, and Datastore.

About App Engine

Google App Engine applications are easy to create, easy to maintain, and easy to scale as your traffic and data storage needs change. With App Engine, there are no servers to maintain. You simply upload your application and it's ready to go.

App Engine applications automatically scale based on incoming traffic. Load balancing, microservices, authorization, SQL and NoSQL databases, traffic splitting, logging, search, versioning, roll out and roll backs, and security scanning are all supported natively and are highly customizable.

App Engine's [Flexible Environment](https://cloud.google.com/appengine/docs/flexible) supports a host of programming languages, including Java, Python, PHP, NodeJS, Ruby, and Go. App Engine's [Standard Environment](https://cloud.google.com/appengine/docs/about-the-standard-environment) is an additional option for certain languages including Python. The two environments give users maximum flexibility in how their application behaves since each environment has certain strengths. Read [Choosing an App Engine Environment](https://cloud.google.com/appengine/docs/the-appengine-environments) for more information.

**What you'll learn**

* How to deploy a simple web application to the App Engine Flexible Environment
* How to access the Google Cloud client libraries for Vision, Storage, and Datastore
* How to use the Cloud Shell

Prerequisites

* Familiarity with Python
* Familiarity with standard Linux text editors such as vim, emacs, or nano
* Access to an image with a face

**Setup and Requirements**

**Qwiklabs 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.

**Cloud Console**

**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.



1. 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**. 

1. 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).

1. 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. 

**Cloud Shell**

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:



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](https://cloud.google.com/sdk/gcloud" \t "_blank).

**Get the sample code**

In Cloud Shell on the command-line, run the following command to clone the Github repository:

git clone https://github.com/GoogleCloudPlatform/python-docs-samples.git

Change directory into python-docs-samples/codelabs/flex\_and\_vision:

cd python-docs-samples/codelabs/flex\_and\_vision

**Authenticate API Requests**

The Datastore, Storage, and Vision APIs are automatically enabled for you in this lab. In order to make requests to the APIs, you will need service account credentials. You can generate credentials from your project using *gcloud* in Cloud Shell. Your Project ID can be found on the Qwiklabs tab, where you started the lab.

Set an environment variable for [YOUR\_PROJECT\_ID], replacing [YOUR\_PROJECT\_ID] with your own project ID:

export PROJECT\_ID=[YOUR\_PROJECT\_ID]

Create a Service Account to access the Google Cloud APIs when testing locally:

gcloud iam service-accounts create qwiklab \

--display-name "My Qwiklab Service Account"

Give your newly created Service Account appropriate permissions:

gcloud projects add-iam-policy-binding ${PROJECT\_ID} \

--member serviceAccount:qwiklab@${PROJECT\_ID}.iam.gserviceaccount.com \

--role roles/owner

After creating your Service Account, create a Service Account key:

gcloud iam service-accounts keys create ~/key.json \

--iam-account qwiklab@${PROJECT\_ID}.iam.gserviceaccount.com

This command generates a service account key stored in a JSON file named key.json in your home directory.

Using the absolute path of the generated key, set an environment variable for your service account key:

export GOOGLE\_APPLICATION\_CREDENTIALS="/home/${USER}/key.json"

You can read more about [authenticating the Vision API](https://cloud.google.com/vision/docs/common/auth).

Click **Check my progress** below to check your lab progress.

Authenticate API Requests

Check my progress

**Testing the Application Locally**

**Starting your virtual environment and installing dependencies**

Create an isolated Python 3 environment named env with [virtualenv](https://virtualenv.pypa.io/en/stable" \t "_blank):

virtualenv -p python3 env

Enter your newly created *virtualenv* named env:

source env/bin/activate

Use pip to install dependencies for your project from the requirements.txt file:

pip install -r requirements.txt

The requirements.txt file is a list of package dependencies you need for your project. The above command downloaded all of these listed package dependencies to the *virtualenv*.

Creating an App Engine App

Next, create an App Engine instance by using:

gcloud app create

A prompt will display a list of regions. Select a Region that supports App Engine Flexible for Python then press **Enter**. You can read more about Regions and Zones [here](https://cloud.google.com/docs/geography-and-regions).

**Creating a Storage Bucket**

First, set the environment variable *CLOUD\_STORAGE\_BUCKET* equal to the name of your *PROJECT\_ID*. (It is generally recommended to name your bucket the same as your *PROJECT\_ID* for convenience purposes).

export CLOUD\_STORAGE\_BUCKET=${PROJECT\_ID}

Now run the following command to create a bucket with the same name as your *PROJECT\_ID*.

gsutil mb gs://${PROJECT\_ID}

Click **Check my progress** below to check your lab progress.

Create an App Engine App and Storage Bucket

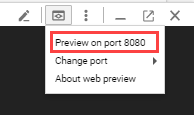
Check my progress

**Running the Application**

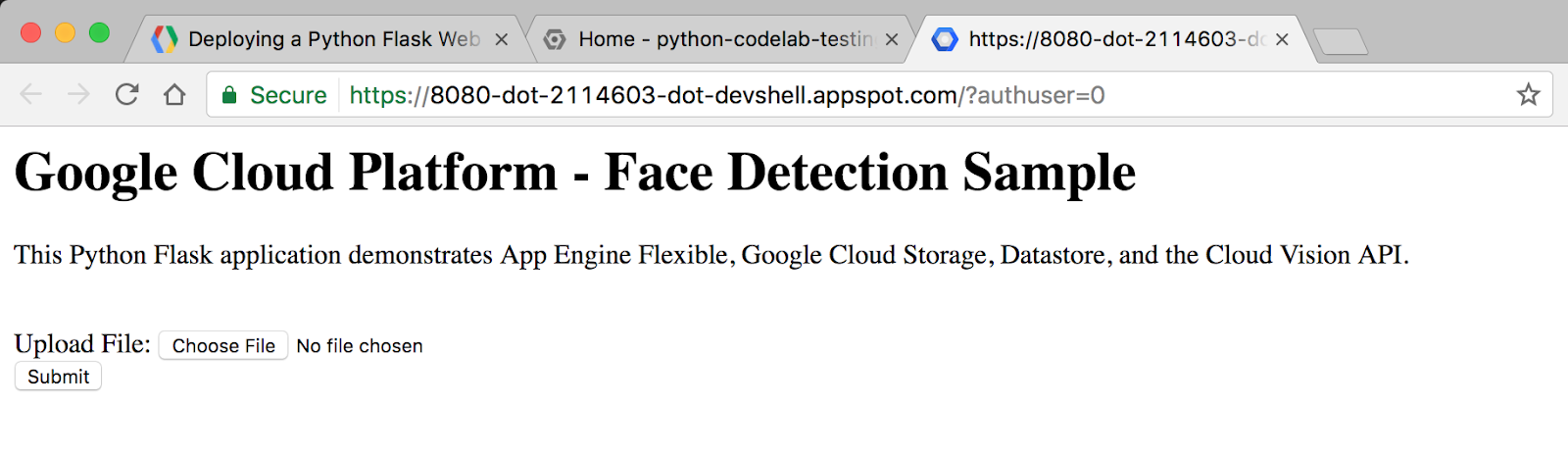
Execute the following command to start your application:

python main.py

Once the application starts, click on the Web Preview icon in the Cloud Shell toolbar and choose "Preview on port 8080."

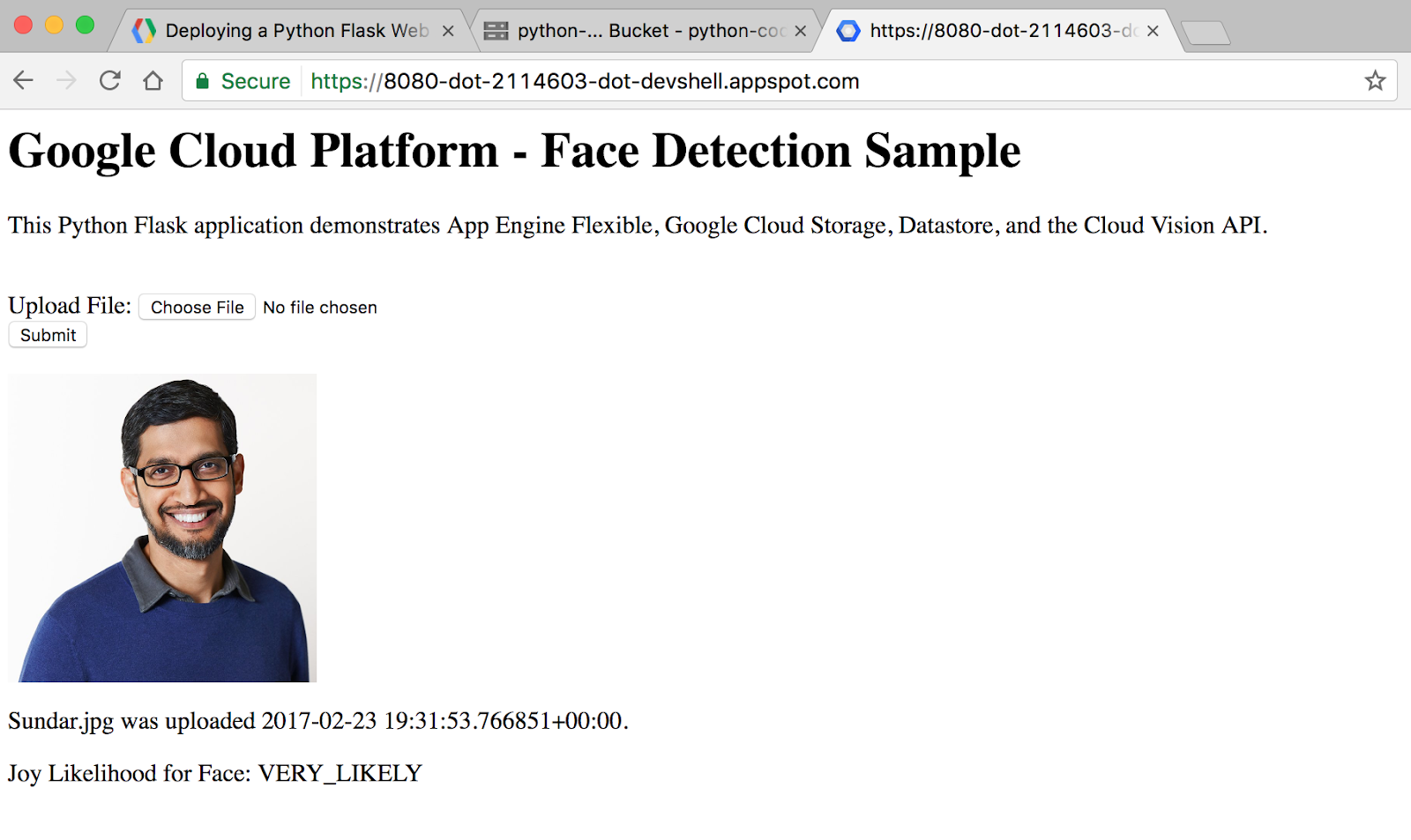


A tab in your browser opens and connects to the server you just started. You should see something like this:



Now things will get interesting! Click the **Choose File** button, find an image from your computer that has a human face, and then click **Submit**.

After uploading a photo, you should see something like this:



**Note:** When you are done testing your application locally, press **Ctrl+C** on the Cloud Shell command line to shut down the local web server.

Click **Check my progress** below to check your lab progress.

Run the Application

Check my progress

**Exploring the Code**

**Sample Code Layout**

The sample has the following layout:

templates/

homepage.html /\* HTML template that uses Jinja2 \*/

app.yaml /\* App Engine application configuration file \*/

main.py /\* Python Flask web application \*/

requirements.txt /\* List of dependencies for the project \*/

**main.py**

This Python file is a Flask web application. The application allows users to submit photos (preferably of faces), which are stored in Cloud Storage and analyzed using the face detection feature of the Cloud Vision API. Key information about each photo is stored in Datastore, Google Cloud's NoSQL database, where it is accessed each time a user visits the website.

This application uses the Google Cloud client libraries for Storage, Datastore, and Vision. These client libraries make it easy to access Cloud APIs from your favorite programming languages.

Let's take a look at some key snippets of the code.

The imports section at the top is where we import the various packages we need for our code. This is how we import our Google Cloud client libraries for Datastore, Storage, and Vision:

from google.cloud import datastore

from google.cloud import storage

from google.cloud import vision

Here is the code for what happens when a user visits the root URL of the website. A Datastore client object is created, which is used to access the Datastore client library. A query on Datastore is run for entities of kind *Faces*. Finally, the HTML template is rendered, passing in the *image\_entities* we extract from Datastore as a variable.

@app.route('/')

def homepage():

# Create a Cloud Datastore client.

datastore\_client = datastore.Client()

# Use the Cloud Datastore client to fetch information from Datastore about

# each photo.

query = datastore\_client.query(kind='Faces')

image\_entities = list(query.fetch())

# Return a Jinja2 HTML template and pass in image\_entities as a parameter.

return render\_template('homepage.html', image\_entities=image\_entities)

Let's take a look at how [entities](https://cloud.google.com/datastore/docs/concepts/entities) are saved to Datastore. Datastore is Google Cloud's NoSQL database solution. Data is stored in objects called *entities*. Each entity is assigned a unique identifying *key*, which can be created using a *kind* and a *key name* string. A *kind* is an organizational bucket for what type of *entity* it is. For example, we might want to set up *kinds* for Photos, People, and Animals.

Each *entity* can have multiple developer-defined *properties*, which can have values of a number of types, including integers, floats, strings, dates, or binary data.

# Create a Cloud Datastore client.

datastore\_client = datastore.Client()

# Fetch the current date / time.

current\_datetime = datetime.now()

# The kind for the new entity.

kind = 'Faces'

# The name/ID for the new entity.

name = blob.name

# Create the Cloud Datastore key for the new entity.

key = datastore\_client.key(kind, name)

# Construct the new entity using the key. Set dictionary values for entity

# keys blob\_name, storage\_public\_url, timestamp, and joy.

entity = datastore.Entity(key)

entity['blob\_name'] = blob.name

entity['image\_public\_url'] = blob.public\_url

entity['timestamp'] = current\_datetime

entity['joy'] = face\_joy

# Save the new entity to Datastore.

datastore\_client.put(entity)

The Storage and Vision client libraries can be accessed programmatically in a similar manner to Datastore. You can open the *main.py* file yourself using *vim*, *emacs*, or *nano* to explore all of the sample code.

**homepage.html**

The Flask web framework leverages Jinja2 as a template engine. This allows us to pass in variables and expressions from *main.py* into *homepage.html* that get replaced with values once the page is rendered.

Learn more about Jinja2 at <http://jinja.pocoo.org/docs/2.9/templates>.

This Jinja2 HTML template displays a form for users to submit photos to the database. It also displays each previously submitted image along with its file name, upload date/time, and the likelihood that the face detected by the Vision API is happy.

**homepage.html**

<h1>Google Cloud Platform - Face Detection Sample</h1>

<p>This Python Flask application demonstrates App Engine Flexible, Google Cloud

Storage, Datastore, and the Cloud Vision API.</p>

<br>

<html>

<body>

<form action="upload\_photo" method="POST" enctype="multipart/form-data">

Upload File: <input type="file" name="file"><br>

<input type="submit" name="submit" value="Submit">

</form>

{% for image\_entity in image\_entities %}

<img src="{{image\_entity['image\_public\_url']}}" width=200 height=200>

<p>{{image\_entity['blob\_name']}} was uploaded {{image\_entity['timestamp']}}.</p>

<p>Joy Likelihood for Face: {{image\_entity['joy']}}</p>

{% endfor %}

</body>

</html>

**Deploying the App to App Engine Flexible**

App Engine Flexible uses a file called app.yaml to describe an application's deployment configuration. If this file is not present, App Engine will try to guess the deployment configuration. However, it is a good idea to provide this file.

Next, you will modify app.yaml using an editor of your choice *vim*, *nano*, or *emacs*. We will use the nano editor:

nano app.yaml

Once you have app.yaml open, replace <your-cloud-storage-bucket> with the name of your Cloud Storage bucket. (If you forgot the name of your Cloud Storage bucket, copy the *Project ID* from the Qwiklabs tab). The env\_variables section sets up environment variables that will be used in main.py once the application is deployed.

runtime: python

env: flex

entrypoint: gunicorn -b :$PORT main:app

runtime\_config:

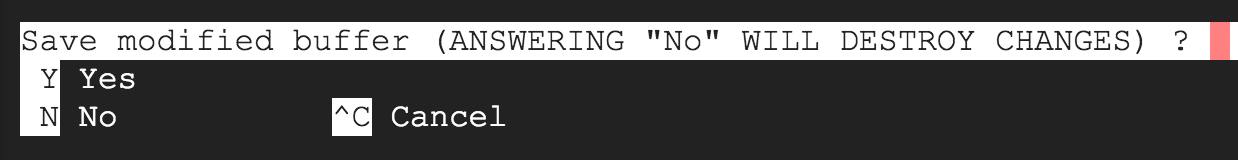
python\_version: 3

env\_variables:

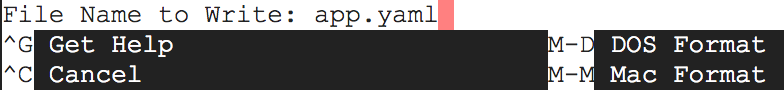
CLOUD\_STORAGE\_BUCKET: <your-cloud-storage-bucket>

This is the basic configuration needed to deploy a Python 3 App Engine Flex application. You can learn more about configuring App Engine [here](https://cloud.google.com/appengine/docs/flexible/python/configuring-your-app-with-app-yaml).

You can now save and close the file in nano by using (**Ctrl + x**), which will prompt:



Type a letter **Y** and then press the ***ENTER*** key one more time to confirm the filename for the following prompt:



Update your Cloud Build timeout:

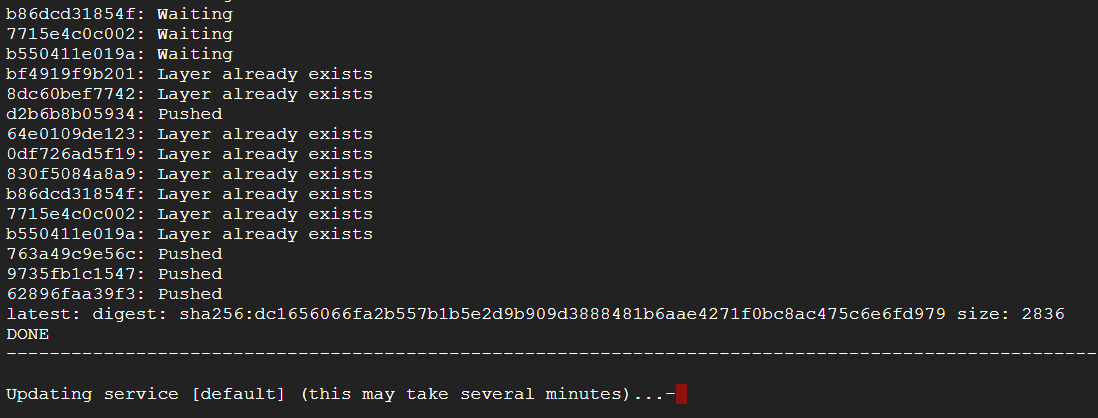
gcloud config set app/cloud\_build\_timeout 1000

Deploy your app on App Engine by using gcloud:

gcloud app deploy

If asked, Do you want to continue (Y/n), press **Y** and then **Enter**.

Watch in Cloud Shell as the application gets built. This will up to **10** minutes. The App Engine Flexible environment is automatically provisioning a Compute Engine virtual machine for you behind the scenes, and then installing the application, then starting it.



After the application is deployed, open the app in your web browser with the following URL:

https://<PROJECT\_ID>.appspot.com

If you forgot your *PROJECT\_ID*, run gcloud config list project from the Cloud Shell command line.

Click **Check my progress** below to check your lab progress.

Deploy the App

Check my progress

**Summary**

You have set up a Python web application and deployed it to the App Engine Flexible environment.

**Congratulations!**

You learned how to write and deploy a Python web application to the App Engine Flexible environment!