Summary

This solution consists from three parts - a Python application **myapi**, **Nginx** web server and **PostgreSQL** database engine (see the system diagram in docs).

myapi is a simple implementation of REST API based on Flask.

Nginx is used as a front-end proxy and a software load balancer.

PostgreSQL is used as a data storage.

You will need an account in GCP where you will create a GKE cluster for deploying nginx with myapi application. Also you will create a database instance in Cloud SQL. And GCR will be used for storing docker images.

This guide requires execution of commands in a shell environment, for instance, in Mac OS X Terminal.

This guide assumes that you have already installed and configured **jq**, **git**, **curl**, **psql**, **docker**, **gcloud**, **kubectl** and **helm** tools on your host [1].

This solution also can be deployed to AWS (EKS + RDS) or even locally (Minikube + PostgreSQL server).

Explanation of my RESTful API implementation

I implemented only one endpoint and two methods GET and PUT for this endpoint as they were described in the task.

I also want to note that there is a small inaccuracy in the task.

There are the HTTP response codes 201 (Created) and 204 (No Content), but there is no a response code 201 (No Content) which was mentioned in the task.

I was guided by the requirements for implementing the PUT method from RESTful API documentation [2]:

> *If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request.*

I also added two extra response message in case a user doesn't exist in a database with the HTTP response code 404 (Not Found) and for a birthday date validation error the HTTP response code 400 (Bad Request) [3].

Fetch source code and scripts

Create a workdir folder

```
$ mkdir ~/workdir
$ cd ~/workdir
```

Clone the repository with myapi application into the workdir

```
$ git clone git@github.com:acpfog/python-myapi.git ~/workdir/python-myapi
```

Set variables

Set an environment variable with your external IP address A.A.A.A

```
$ export MY_EXTERNAL_IP=A.A.A.A
```

Set a special environment variable with a password for postgres user

```
$ export PGPASSWORD="[POSTGRES_PASSWORD]"
```

Set variables with a database name for storing data, a user's name for connecting to the database and a user's password

```
$ export DATABASE_NAME=myapi
$ export DATABASE_USER=myapi
$ export DATABASE_PASS=[SOME_DB_PASSWORD]
```

Set environment variables with a docker image name and a tag with version for myapi image

```
$ export IMAGE_NAME=gcr.io/yourproject/somenamespace/myapi-app
$ export IMAGE_TAG=0.1.1
```

Prepare an environment in GCP

Install Google Container Registry's Docker credential helper

```
$ gcloud components install docker-credential-gcr
```

Create a GKE cluster

```
$ gcloud container clusters create test-cluster --num-nodes=3 \
 --machine-type=g1-small --region europe-west3-b
```

Authenticate into the cluster

```
$ gcloud container clusters get-credentials test-cluster \
 --region europe-west3-b
```

Create a service account for Helm in the GKE cluster

```
$ kubectl create serviceaccount --namespace kube-system tiller
```

Bind the cluster admin role to the service account

```
$ kubectl create clusterrolebinding tiller-cluster-admin \
--clusterrole=cluster-admin --serviceaccount=kube-system:tiller
```

Init Helm with the created service account

```
$ helm init --service-account tiller
```

Create a database instance with the PostgreSQL engine

```
$ gcloud sql instances create db-test --database-version=POSTGRES_9_6 \
--tier db-g1-small --region europe-west3
```

Add to the database authorized networks external IP addresses (your and the nodes)

```
$ gcloud sql instances patch db-test --authorized-networks=$(kubectl get
nodes -o json | jq -r '.items[].status.addresses[] |
select(.type=="ExternalIP") | { address: .address }' | jq -rs "[.[][]] +
[\"$MY_EXTERNAL_IP\"] | join(\",\")")
```

Set an environment variable with the database instance external IP address

```
$ export DATABASE_HOST=$(gcloud sql instances list --filter name=db-test
--format json | jq -r '.[].ipAddresses[] | select(.type=="PRIMARY") |
.ipAddress')
```

Set the password for postgres user

```
$ gcloud sql users set-password postgres no-host --instance=db-test \
--password=$PGPASSWORD
```

Create a database for storing data

```
$ psql -h $DATABASE_HOST -U postgres -d postgres -c “CREATE DATABASE
$DATABASE_NAME”
```

Add a user for accessing the database

```
$ psql -h $DATABASE_HOST -U postgres -d postgres -c “CREATE USER
$DATABASE_USER PASSWORD \”$DATABASE_PASS\””
```

Grant all privileges to the user on the database

```
$ psql -h $DATABASE_HOST -U postgres -d postgres -c “GRANT ALL PRIVILEGES
ON DATABASE $DATABASE_NAME TO $DATABASE_USER”
```

Update special environment variable for running psql with the new user privileges and without prompting a password

```
$ export PGPASSWORD="$DATABASE_PASS"
```

Create a table for storing information about users

```
$ psql -h $DATABASE_HOST -U myapi -d myapi -c “CREATE TABLE users (user_id
serial PRIMARY KEY, user_name VARCHAR(64) UNIQUE NOT NULL, user_bd DATE NOT
NULL)”
```

Build the application image

Authenticate into GCR

```
$ gcloud auth configure-docker
```

Choose the folder with Dockerfile

```
$ cd ~/workdir/python-myapi
```

Build an image

```
$ docker build -t $IMAGE_NAME:$IMAGE_TAG .
```

Push the image to GCR

```
$ docker push $IMAGE_NAME:$IMAGE_TAG
```

Deploy the solution

Choose the folder with Helm charts

```
$ cd ~/workdir/python-myapi/deploy
```

Deploy myapi and nginx

```
$ helm upgrade myapi --install --wait myapi \
--set image=$IMAGE_NAME,imageTag=$IMAGE_TAG \
--set database.host=$DATABASE_HOST \
--set database.name=$DATABASE_NAME \
--set database.user=$DATABASE_USER,database.pass=$DATABASE_PASS
$ helm upgrade nginx --install --wait nginx
```

For further deploys set the database parameters in the myapi/values.yaml file in the database section.

Test the solution

Set an environment variable with the nginx service external IP address

```
$ export API_HOST=$(kubectl get services nginx -o json | jq -r \
'.status.loadBalancer.ingress[].ip')
```

Manually with curl

```
$ curl -i -X GET http://$API_HOST/hello/John
$ curl -i -H 'Content-Type: application/json' -X PUT \
-d '{ "dateOfBirth": "2000-01-01" }' http://$API_HOST/hello/John
$ curl -i -X GET http://$API_HOST/hello/John
$ curl -i -H 'Content-Type: application/json' -X PUT \
-d '{ "dateOfBirth": "2000-09-09" }' http://$API_HOST/hello/John
$ curl -i -X GET http://$API_HOST/hello/John
```

Run automated tests

Create a Python virtual environment

```
$ virtualenv ~/workdir/venv
```

Activate installed virtual environment

```
$ source ~/workdir/venv/bin/activate
```

Install requirements

```
(venv) $ pip install -r ~/workdir/python-myapi/requirements.txt
```

Run tests

```
(venv) $ cd ~/workdir/python-myapi
(venv) $ python test_myapi.py
```

Conclusion

One of the main advantages of Kubernetes and Helm - they give you a deployment flexibility.

With the same Helm chart it possible to deploy and upgrade an application in a Kubernetes cluster with no-downtime approach in any environment.

Helm tool can use different values files with appropriate parameters for a corresponding environment. You can deploy the solution to different cloud platforms (GCP, AWS) with many environments (development, testing, production) in different clusters.

Kubernetes continuously checks each deployed objects and keeps them running. With Kubernetes you can easily scale your deployed applications. Also Kubernetes provides a lot of data and possibilities for monitoring your applications.

You can improve the PostgreSQL engine availability and reliability turning on special functionality in Cloud SQL like high availability, making snapshots, creating backups and replications.

Building, deploy and test steps from this guide can be automated with Jenkins pipelines and scripts written in Groovy for instance. A pipeline in Jenkins can be triggered by changes in a source code repository.

How to install tools on Mac OS X with Homebrew

```
$ brew install jq
$ brew install git
$ brew install docker
$ brew install docker-machine
$ brew install postgresql
$ brew cask install google-cloud-sdk
$ brew install kubernetes-cli
$ brew install kubernetes-helm
```

Links

1. HTTP PUT method in RESTful API - https://restfulapi.net/http-methods/#put
2. HTTP status codes in RESTful API - https://restfulapi.net/http-status-codes/