Deploying highly available PostgreSQL with GKE

This tutorial describes how to deploy the database engine PostgreSQL in a Google Kubernetes Engine (GKE) cluster. The tutorial also discusses the GKE deployment in comparison to a conventional deployment of PostgreSQL on a virtual machine or in Cloud SQL. This tutorial assumes that you're familiar with using Kubernetes, the Google Cloud Console, and the `gcloud` command-line tool.

Objectives

- Learn to install a PostgreSQL instance in GKE using a standard Docker image.
- Enable access from your laptop and public access to the database instance.
- Understand the architectural considerations of installing PostgreSQL in GKE compared to a virtual machine installation.

Costs

This tutorial uses the following billable components of Google Cloud:

- Compute Engine
- Google Kubernetes Engine

To generate a cost estimate based on your projected usage, use the pricing calculator (/products/calculator).

Before you begin

In this tutorial, you use `gcloud` tool commands. This tutorial assumes that you have either already installed the Cloud SDK (/sdk/docs/install) or that you are familiar with how to use Cloud Shell (/shell/docs/running-gcloud-commands).

1. In the Google Cloud Console, on the project selector page, select or create a Google Cloud project.

★ **Note**: If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

Go to the project selector page (https://console.cloud.google.com/projectselector2/home/dashboa

2. Make sure that billing is enabled for your Cloud project. Learn how to confirm that billing is enabled for your project (/billing/docs/how-to/modify-project).

When you finish this tutorial, you can avoid continued billing by deleting the resources you created. For more information, see Cleaning up (#clean-up) later on this page.

Understanding stateful Kubernetes database applications

You use Docker containers to implement microservice-based applications on Kubernetes. You then encode the business logic of the application in Docker images that you deploy into Kubernetes. When you start the application, it's executed as containers in Kubernetes pods.

To provide their business logic, stateful applications rely on persistent state. The application stores state into a persistence layer that ensures that data is available even if the application is restarted (for example, when the application is upgraded or after an outage).

Databases rely heavily on local disks for persistence. This document shows how you can deploy a PostgreSQL instance into GKE as a container based on standard PostgreSQL images. This document also shows how you can deploy highly available PostgreSQL on GKE using regional persistent disks.

Various Kubernetes operators are available to install a PostgreSQL instance, such as those available from Zalando (https://www.postgresql.org/about/news/postgres-operator-v150-2036/) and CrunchyData (https://github.com/CrunchyData/postgres-operator). However, this document focuses on the basic installation relying solely on standard PostgreSQL Docker images.

Deploying a database instance as a container isn't the only option and might not be the best approach for you. Instead, you can run the database instance within a Compute Engine

instance or in Cloud SQL for PostgreSQL. There are positives and negatives to these approaches, which are discussed in the Understanding options to deploy a database instance in GKE (#understanding_options_to_deploy_a_database_instance_in_gke) section later in this document.

Database deployment on Kubernetes

On a high level, a database instance can run within a Kubernetes container. A database instance stores data in files, and the files are stored in persistent volume claims. A `PersistentVolumeClaim` must be created and made available to a PostgreSQL instance. In this tutorial, you use a regional persistent disk as the underlying storage class in order to implement PostgreSQL with high availability.

To create the database instance as a container, you use a deployment configuration. In order to provide an access interface that is independent of the particular container, you create a service that provides access to the database. The service remains unchanged even if a container (or pod) is moved to a different node.

A database that runs as a service in a Kubernetes cluster and that stores its database files in `PersistentVolumeClaims` is bound to the lifecycle of the cluster. If the cluster is deleted, the database is also deleted.

Deploying PostgreSQL

The following steps show you how to install a high-availability PostgreSQL database instance running in GKE as a service. The configuration values that you use are example values. You can adjust the configuration values to fit your workload.

In this tutorial, you run `gcloud` tool commands from your local command line or in Cloud Shell.

Create a GKE regional cluster

In this step, you install a regional GKE cluster (/kubernetes-engine/docs/concepts/regional-clusters). Unlike a zonal cluster, a regional cluster is replicated into several zones, so an outage in a single zone doesn't make the service unavailable.

1. Verify that `kubectl` is installed:

```
kubectl version
```

If **kubectl** is installed, the command returns a client version and you can skip the next step. This tutorial works with any version of **kubectl**.

2. Install **kubectl** into your local environment if necessary:

```
gcloud components install kubectl
```

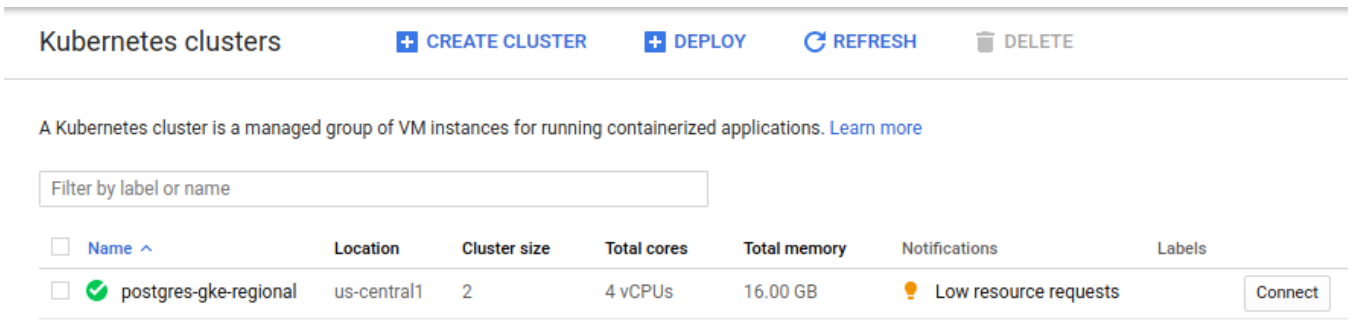
3. Create a regional GKE cluster in the **us-central1** region with one node each in two different zones:

```
gcloud container clusters create "postgres-gke-regional" \
  --region "us-central1" \
  --machine-type "e2-standard-2" --image-type "COS" --disk-type "pd-standar
  --num-nodes "1" --node-locations "us-central1-b","us-central1-c"
```

4. Get the GKE cluster credentials:

```
gcloud container clusters get-credentials postgres-gke-regional --region us
```

You now have a regional cluster installed and available for installing PostgreSQL into the cluster. The following screenshot of the Cloud Console Kubernetes clusters (https://console.cloud.google.com/kubernetes/list) page shows the regional cluster:



Deploy PostgreSQL to the regional GKE cluster

In this step, you create the volume and persistent volume claim. The volume will be a blank regional persistent disk across two zones (`us-central1-b` and `us-central1-c`).

1. Create a `postgres-pv.yaml` file, and then apply it to the GKE cluster. Doing this creates the required `PersistentVolumeClaim` based on a regional persistent disk (/compute/docs/disks#repds).

```
cat > postgres-pv.yaml << EOF
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: regionalpd-storageclass
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: regional-pd
allowedTopologies:
  - matchLabelExpressions:
      - key: failure-domain.beta.kubernetes.io/zone
        values:
          - us-central1-b
          - us-central1-c
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: postgresql-pv
spec:
  storageClassName: regionalpd-storageclass
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 300Gi
EOF

kubectl apply -f postgres-pv.yaml
```

2. Create and apply a PostgreSQL deployment:

```
cat > postgres-deployment.yaml << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
 name: postgres
spec:
 strategy:
   rollingUpdate:
     maxSurge: 1
     maxUnavailable: 1
   type: RollingUpdate
 replicas: 1
 selector:
   matchLabels:
     app: postgres
 template:
   metadata:
     labels:
       app: postgres
   spec:
     containers:
       - name: postgres
         image: postgres:10
         resources:
           limits:
             cpu: "1"
             memory: "4Gi"
           requests:
             cpu: "1"
             memory: "4Gi"
         ports:
           - containerPort: 5432
         env:
           - name: POSTGRES_PASSWORD
             value: password
           - name: PGDATA
             value: /var/lib/postgresql/data/pgdata
         volumeMounts:
           - mountPath: /var/lib/postgresql/data
             name: postgredb
     volumes:
       - name: postgredb
         persistentVolumeClaim:
           claimName: postgresql-pv
```

```
EOF

kubectl apply -f postgres-deployment.yaml
```

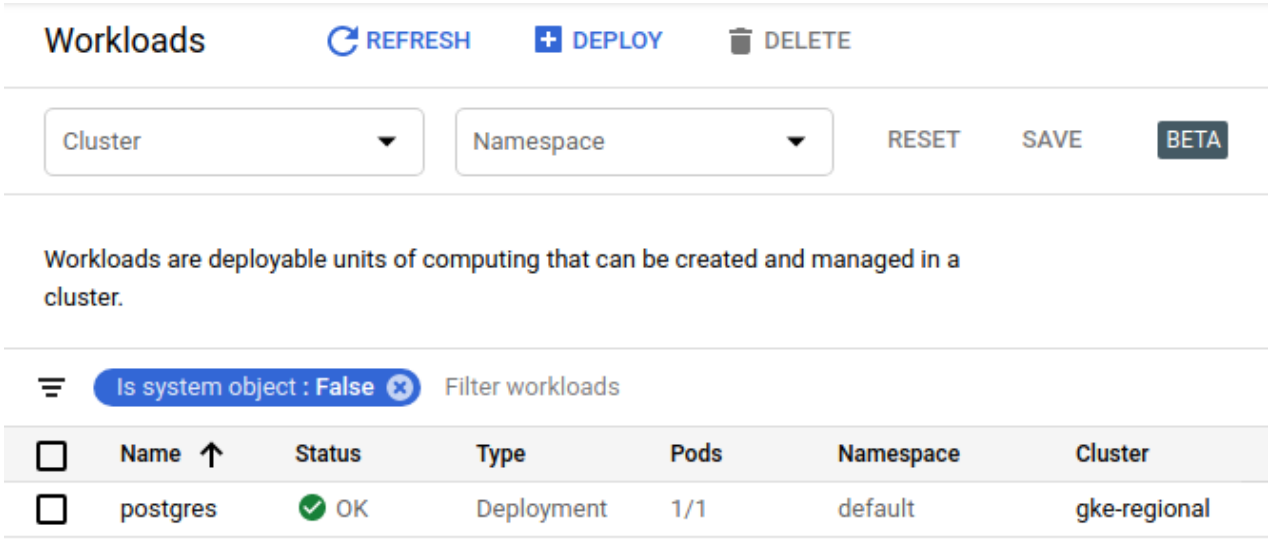
In a production environment, don't include a cleartext password in a configuration file that resides in a code repository. Instead, use Secrets to store sensitive data (/config-connector/docs/how-to/secrets).

3. Create and apply the PostgreSQL service:

```
cat > postgres-service.yaml << EOF
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  ports:
    - port: 5432
  selector:
    app: postgres
  clusterIP: None
EOF

kubectl apply -f postgres-service.yaml
```

The PostgreSQL database instance is now running in GKE as a service. The following screenshot of the Cloud Console Workloads (https://console.cloud.google.com/kubernetes/workload) page shows the service status:



Next, you create a sample dataset and execute a simulated failover to test the high-availability setup of the PostgreSQL service.

Creating a test dataset

In this section, you create a database and a table with sample values. The database serves as a test dataset for the failover process that you test later in this document.

1. Connect to your PostgreSQL instance:

```
POD=`kubectl get pods -l app=postgres -o wide | grep -v NAME | awk '{print $

kubectl exec -it $POD -- psql -U postgres
```

2. Create a database and a table, and then insert some test rows:

```
create database gke_test_regional;

\c gke_test_regional;

CREATE TABLE test(
   data VARCHAR (255) NULL
);
```

```
insert into test values
  ('Learning GKE is fun'),
  ('Databases on GKE are easy');
```

3. To verify that the test rows were inserted, select all rows:

```
select * from test;
```

4. Exit the PostgreSQL shell:

```
\q
```

The database instance now has a test dataset. After a failover based on a regional persistent disk, the same dataset must be available after the failover.

Simulating database instance failover

To simulate failover, you remove the node that is hosting the PostgreSQL pod, and then delete the existing pod. When you delete the pod, the GKE cluster is forced to move PostgreSQL into a different zone.

1. Identify the node that is currently hosting PostgreSQL:

```
CORDONED_NODE=`kubectl get pods -l app=postgres -o wide | grep -v NAME | awk

echo ${CORDONED_NODE}

gcloud compute instances list --filter="name=${CORDONED_NODE}"
```

   Notice the zone where this node was created.

   The node now has two disks attached, as shown in the following screenshot of the Cloud Console VM instances (https://console.cloud.google.com/compute/instances) page:

Size (GB)	Device name	Type	Encryption	Mode	When deleting instance
100	persistent-disk-0	Standard persistent disk	Google managed	Boot, read/write	Delete disk

Size (GB)	Device name	Type	Encryption	Mode	When deleting instance
300	gke-postgres-gke-regio-pvc-7e30e0f6-44e4-46bb-b389-de6212d33664	Regional standard persistent disk	Google managed	Read/write	Keep disk

2. Disable scheduling of any new pods on this node:

```
kubectl cordon ${CORDONED_NODE}

kubectl get nodes
```

   The node is cordoned, so scheduling is disabled on the node that the database instance resides on.

3. Delete the existing PostgreSQL pod:

```
POD=`kubectl get pods -l app=postgres -o wide | grep -v NAME | awk '{print 

kubectl delete pod ${POD}
```

4. Verify that a new pod is created on the other node.

```
kubectl get pods -l app=postgres -o wide
```

   It might take a while for the new pod to be ready (usually around 30 seconds).

5. Verify the node's zone:

```
NODE=`kubectl get pods -l app=postgres -o wide | grep -v NAME | awk '{print

echo ${NODE}

gcloud compute instances list --filter="name=${NODE}"
```

Notice that the pod is deployed in a different zone from where the node was created at the beginning of this procedure.

The following screenshot shows that the new VM has the same disks available (including the regional persistent disk) that the preceding VM had:

Size (GB)	Device name	Type	Encryption	Mode	When deleting instance
100	persistent-disk-0	Standard persistent disk	Google managed	Boot, read/write	Delete disk

Size (GB)	Device name	Type	Encryption	Mode	When deleting instance
300	gke-postgres-gke-regio-pvc-7e30e0f6-44e4-46bb-b389-de6212d33664	Regional standard persistent disk	Google managed	Read/write	Keep disk

6. Connect to the database instance:

```
POD=`kubectl get pods -l app=postgres -o wide | grep -v NAME | awk '{print $

kubectl exec -it $POD -- psql -U postgres
```

7. Verify that the test dataset exists:

```
\c gke_test_regional;

select * from test;

\q
```

The output matches the test values that you inserted when you created the test database earlier in this tutorial.

You have now completed a simulated failover that proves that based on the regional disk, PostgreSQL can fail over without loss of data.

To make the cordoned node schedulable again, execute the following commands:

1. Re-enable scheduling for the node for which scheduling was disabled:

```
kubectl uncordon $CORDONED_NODE
```

2. Check that the node is ready again:

```
kubectl get nodes
```

   The output is a list of nodes that includes the previously cordoned node.

The regional cluster is now fully functional again.

Configuring database access

This section provides options to configure database access. First, you create public access to connect to the database. Next, you set up port forwarding to connect to the database from your local machine.

If you want to provide access to the database from outside your organization, you can enable public access. You can use firewall rules to restrict the IP addresses that are allowed to access the database. A public IP address doesn't mean that anybody can access the database.

If you need to access the database from your machine in context of software development, or to analyze the database contents, you can set up port forwarding. Port forwarding doesn't require you to create a public IP address.

Create public access

At this time, the database isn't accessible over a public IP address. To see that no public IP address is available, run the following command:

```
kubectl get services postgres
```

To make the service publicly accessible, complete the following steps:

1. Remove the current non-public service:

```
kubectl delete -f postgres-service.yaml
```

2. Create and deploy a new service configuration file to create a public IP address:

```
cat > postgres-service.yaml << EOF
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  ports:
    - port: 5432
  selector:
    app: postgres
  type: LoadBalancer
EOF

kubectl apply -f postgres-service.yaml
```

3. Verify that a public IP (`EXTERNAL-IP`) address will be assigned. This step might take a few moments.

```
kubectl get services postgres
```

After the public IP address is available, you can connect to the database.

Set up port forwarding

Instead of opening up the database service and providing a public IP address, you can use port forwarding to access the database from your machine:

1. To remove the public IP address, reset the service:

```
kubectl delete -f postgres-service.yaml

cat > postgres-service.yaml << EOF
```

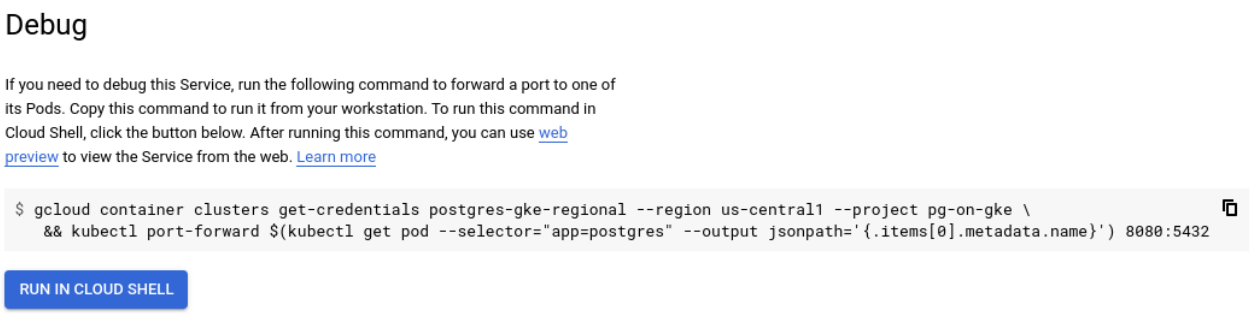
```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  ports:
    - port: 5432
  selector:
    app: postgres
  clusterIP: None
EOF

kubectl apply -f postgres-service.yaml
```

2. Verify that `EXTERNAL-IP` has a value of `<none>`:

```
kubectl get services postgres
```

3. In the Cloud Console, go to **Kubernetes Engine > Services & Ingress** (https://console.cloud.google.com/kubernetes/discovery).
4. On the **Services** tab, click the service name **`postgres`**.
5. Click **Port Forwarding**. The **Debug** dialog appears, similar to the following screenshot:



6. In the **Debug** dialog, copy the **`gcloud`** tool command, and then change the port **`8080`** to **`5432`**. The resulting command looks like the following:

```
gcloud container clusters get-credentials postgres-gke-regional \
  --region us-central1 --project pg-on-gke \
    && kubectl port-forward $(kubectl get pod --selector="app=postgres" \
  --output jsonpath='{.items[0].metadata.name}') 5432:5432
```

The change from `8080` to `5432` means that you can access the database using port `5432`.

7. On your local machine, open a terminal window, and then run the port forwarding command that you modified in the preceding step:

```
gcloud container clusters get-credentials postgres-gke-regional \
  --region us-central1 --project pg-on-gke \
    && kubectl port-forward $(kubectl get pod --selector="app=postgres" \
  --output jsonpath='{.items[0].metadata.name}') 5432:5432
```

8. If you aren't already logged in, enter your login credentials when you're prompted:

```
gcloud auth login
```

After the authentication succeeds, run the modified port forwarding command again.

9. Start the PostgreSQL client (or use your preferred IDE), and then verify that you can access the database:

```
psql --host 127.0.0.1 -U postgres -p 5432

\c gke_test_regional;

select * from test;

\q
```

When you use port forwarding, you can access the database without creating a public IP address.

Understanding options to deploy a database instance in GKE

Assuming a microservices-based application running in GKE, the following are deployment options for database instances in GKE:

- **Kubernetes pod**: As shown in this tutorial, you can deploy and run a PostgreSQL instance in a container.
- **Compute Engine instance**: You can install and manage a PostgreSQL instance in a Compute Engine (/compute) instance.
- **Cloud SQL**: For a more managed option, you can launch PostgreSQL in Cloud SQL for PostgreSQL (/sql/docs/postgres).

Each of the preceding options has different considerations, some of which are described in the following list. Some of the considerations aren't strict technical facts, and they might involve trade-offs during the decision process. There is no universal deployment option: use the list to help you decide which option is best for your workload and requirements.

- **Move to a cloud provider-managed database**: A database like Cloud SQL or Cloud Spanner is a managed database. A managed database provides reduced operational overhead and is optimized for the Google Cloud infrastructure. You might find it easier to manage and to operate than a database deployed in Kubernetes.
- **Keep your current database deployment**: If your current database deployment is stable and reliable, and if there is no real requirement or reason to change it, it might be best to keep it.

  If your team develops new applications on Kubernetes, it's important to consider your database deployment. In that case, you might consider a change to your database deployment as a second phase after the application is in production.
- **Database independence**: As mentioned earlier, the lifecycle of a `PersistentVolumeClaim` is tied to the corresponding GKE cluster. If you need to maintain the database and have its lifecycle independent of GKE clusters, you might prefer to keep it separate from GKE in a VM instance or as a managed database.
- **Scaling with GKE**
  - **Vertical scaling**: You can configure automatic vertical scaling for GKE. However, we don't recommend this because vertical scaling causes the pod to be redeployed, which causes a brief outage.
  - **Horizontal scaling**: You can't do horizontal scaling for singleton services such as PostgreSQL.
- **GKE overhead**: GKE reserves resources (/kubernetes-engine/docs/concepts/cluster-architecture#memory_cpu) for its own operations.

Databases aren't scaled automatically, so overhead might be high for small pods.

- **Number of database instances**: In the context of Kubernetes, each database instance runs in its own pod and has its own `PersistentVolumeClaim`. If you have a high number of instances, you have to operate and manage a large set of pods, nodes, and volume claims. You might want to use a managed database instead, because Google Cloud handles a lot of management functionality.
- **Database backup in GKE**: Typically, you make database backups on a separate volume (or sometimes on a shared location). However, to create database backups using GKE, you can deploy a separate pod running the `psql` (http://postgresguide.com/utilities/psql.html) utility `pg_dump` on respective volumes.
- **Kubernetes-specific recovery behavior**: Kubernetes has a built-in assumption and design that any failed pod is recreated, not restarted. From a database instance perspective, this means that when a pod is recreated, any configuration that isn't persistent within a database or on stable storage outside pods is also recreated.
- **Kubernetes configuration**: When you create configuration controls, you have to ensure that you create only one primary pod. You must verify configuration files to ensure that no incorrect configuration accidentally takes place.
- **`PersistentVolumeClaim` scope**: A `PersistentVolumeClaim` is scoped to a GKE cluster. This scoping means that when a GKE cluster is deleted, the volume claim is deleted. Any database files in the cluster are also deleted. In order to guard against accidental loss of the database files, we recommend replication or frequent backup.
- **Database migration**: Unless you continue to use an existing database system, you need to plan for database migration so that the databases in your current system are migrated to the databases running in GKE. For more information, see Database migration: Concepts and principles (Part 1) (/solutions/database-migration-concepts-principles-part-1) and Database migration: Concepts and principles (Part 2) (/solutions/database-migration-concepts-principles-part-2).
- **Re-training**: If you move from a self-managed or provider-managed deployment to a Kubernetes database deployment, you need to retrain database administrators (DBAs) to operate in the new environment as reliably as they operate in the current environment. Application developers might also have to learn about differences to a lesser extent.

The preceding list provides a discussion of some of the considerations for database deployment. However, the list doesn't include all possible considerations. You also need to

consider disaster recovery, connection pooling, and monitoring.

Configuring alternative cluster deployments

In this document, you deployed a regional GKE cluster with one node each in two zones, and you created a regional persistent disk in the same zones. This configuration enables failover on a database instance level: if one of the zones fails, the other zone provides the consistent state on the regional disk, and a PostgreSQL container is started in the zone that remains available.

From a database instance perspective, PostgreSQL runs in one zone only at any time. However, if an application runs (as containers) in two zones at the same time, it assumes the capacity of nodes in two zones. Therefore, a zone failure halves the capacity available for the application

From an application perspective, you might want to create a cluster across three zones. In this case, we recommend planning the outage of one zone from a capacity perspective so that a zone failure doesn't impact the application performance.

Cleaning up

To avoid incurring charges to your Google Cloud account for the resources used in this tutorial, either delete the project that contains the resources, or keep the project and delete the individual resources.

To avoid incurring charges to your Google Cloud account for the resources used in this tutorial, delete the project you created for the tutorial.

Caution: Deleting a project has the following effects:

- **Everything in the project is deleted.** If you used an existing project for this tutorial, when you delete it, you also delete any other work you've done in the project.
- **Custom project IDs are lost.** When you created this project, you might have created a custom project ID that you want to use in the future. To preserve the URLs that use the project ID, such as an **`appspot.com`** URL, delete selected resources inside the project instead of deleting the whole project.

If you plan to explore multiple tutorials and quickstarts, reusing projects can help you avoid exceeding project quota limits.

1. In the Cloud Console, go to the **Manage resources** page.

   Go to Manage resources (https://console.cloud.google.com/iam-admin/projects)

2. In the project list, select the project that you want to delete, and then click **Delete**.
3. In the dialog, type the project ID, and then click **Shut down** to delete the project.

What's next

- For information about using MySQL to create a primary database with replicas, see Run a Replicated Stateful Application (https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/).
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).

Last updated 2021-03-01 UTC.