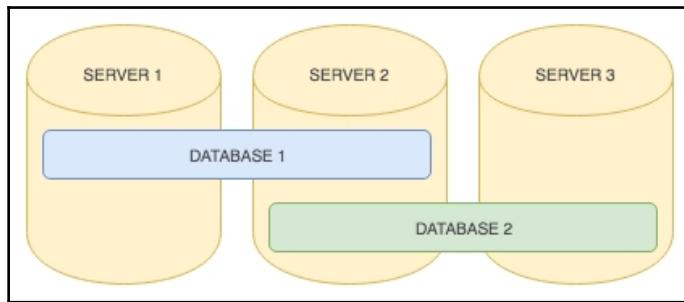


## Sharding approach

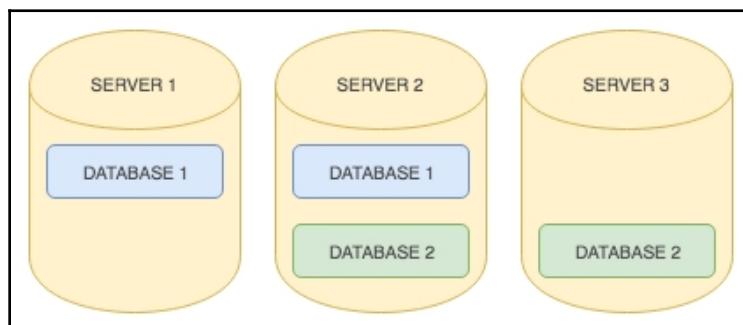
Sharding is the process of fragmenting the data across different locations. This enables us to split a large dataset into different servers, each one with a smaller set. This is comparable to a RAID 0 configuration, where, for example, you may need a 10 TB disk and do this by combining two 5 TB disks:



Doing so is considered scaling, but it comes at a cost. Each piece of data is important; it's not a copy, it's a fundamental part. Sharding adds complexity to your deployment but sometimes is inevitable if you have a very large dataset.

## Replicating approach

Replicating is the process of having copies of the data across different locations. This enables us to have a bigger throughput as we can have different services serving the same data. It also brings complexity, at least for the database servers, as they need to keep everything synchronized:



Doing replicas also gives you redundancy in case of any failures in regards to parts of the server nodes, depending on your configuration. If you use a database cluster for more than one application, you don't need to have replicas of a database in all cluster nodes; it all depends on your needs.

## Sharding and replicating

Looking at our service, now that it's using RethinkDB, we can have the best of both worlds and enable sharding and replication at the same time. Head to the Administration Console and click on the **Tables** tab at the top:

The screenshot shows the RethinkDB Administration Console interface. At the top, there are navigation links: Dashboard, Tables (which is the active tab), Servers, Data Explorer, Logs, and a search bar. Below the navigation, there are four status indicators: 'Connected to <proxy node>', 'Issues No issues', 'Servers 4 connected', and 'Tables 1/1 ready'. The main content area is titled 'Tables in the cluster' and lists a single database named 'imagini'. Underneath the database name are buttons for '+ Add Database' and '- Delete selected tables'. Below the database list, there is a table for the 'images' table, which has a checkbox, a thumbnail icon, the name 'images', a status indicator '1 shard, 1 replica', and a green circular progress bar labeled 'Ready 1/1'. At the bottom of the page, there are links for Documentation, API, Google Groups, '#rethinkdb on freenode', Github, and Community.

Click on our **images** table and scroll down:

The screenshot shows the RethinkDB Administration Console interface. At the top, it displays "1/1 replicas available" and a timeline from 10:03:00 to 10:04:00. Below this, there are two main sections: "Data distribution" and "Sharding and replication".

**Data distribution:** Shows a vertical hierarchy from "Docs" at the top to "Shards" at the bottom.

**Sharding and replication:** Shows "1 shard" and "1 replica per shard". There is a "Reconfigure" button.

**Secondary indexes:** Shows "No secondary indexes found." and a link to "Create a new secondary index →".

**Servers used by this table:** Shows "Shard 1" with "0 documents" and a "Primary replica" status. A node labeled "4da81fc343b3\_ron" is listed under Shard 1. A green circular icon indicates the server is "ready".

At the bottom, there are links to "Documentation", "API", "Google Groups", "#rethinkdb on freenode", "Github", and "Community".

You can see that our table is on one shard (the default), with only one replica per shard. Down the bottom, you can see the server (blue link) on our cluster that is holding the table. Click on **Apply configuration**:

The screenshot shows the RethinkDB Administration Console interface. At the top, it displays the URL `Not Secure | 192.168.99.101:8080/#tables/c3215d2a-31fb-40a3-b788-87c155ecac11`. Below the header, there's a timeline from 10:04:15 to 10:05:15. The main area has two tabs: "Data distribution" and "Sharding and replication". Under "Sharding and replication", it shows "1 shard" and "1 replica per shard", with a "Reconfigure" button. A modal dialog titled "Sharding and replication for imagini.images" is open. It contains a warning message: "Applying changes may cause the table to become temporarily unavailable." It shows fields for "shards" (set to 2, max: 4) and "replicas per shard" (set to 2, max: 4). Below the fields is a question "Where's my data going?". At the bottom of the dialog are "Cancel" and "Apply configuration" buttons. In the background, under "Servers used by this table", it shows "Shard 1" with a primary replica node `4da81fc343b3_ron` labeled as "ready". Navigation links at the bottom include Documentation, API, Google Groups, #rethinkdb on freenode, Github, and Community. The footer shows the IP address `192.168.99.101:8080/#`.

You can configure the number of shards and the number of replicas of every shard. The limitation on four shards and four replicas is because our cluster has only four nodes. Any number above that would not make sense.

Let's configure our table with two shards and two replicas per shard and apply the configuration. Wait a couple of seconds and look at how our table is distributed:

The screenshot shows the RethinkDB Administration Console interface. At the top, it displays the URL `192.168.99.101:8080/#tables/c3215d2a-31fb-40a3-b788-87c155ecac11`. The main area is divided into several sections:

- Data distribution:** Shows a vertical hierarchy from "Docs" at the top to "Shards" at the bottom.
- Sharding and replication:** Displays "2 shards" and "2 replicas per shard". A "Reconfigure" button is available.
- Secondary indexes:** States "No secondary indexes found." and includes a link to "Create a new secondary index →".
- Servers used by this table:** Lists servers grouped by shard.
  - Shard 1:** Contains nodes `4da81fc343b3_ron` (Primary replica, ready) and `c7b765373fff_yxm` (Secondary replica, ready).
  - Shard 2:** Contains nodes `fec31920892e_bfe` (Primary replica, ready) and `fc46b239242f_peu` (Secondary replica, ready).

At the bottom, there are links to Documentation, API, Google Groups, `#rethinkdb` on freenode, Github, and Community.

If one node restarts, the database keeps running without a problem. It will recover and synchronize everything when the node comes online again. Let's just try that. Head to a console of one of the Docker machines and restart one of the database containers.

If you keep the administration console open, you should see something like the following screenshot for just a few seconds:

The screenshot shows the RethinkDB Administration Console interface. At the top, there's a header bar with the title 'RethinkDB Administration Con' and a URL '192.168.99.101:8080/#tables/c3215d2a-31fb-40a3-b788-87c155ecac11'. Below the header, there are two main sections: 'Secondary indexes' and 'Servers used by this table'.

**Secondary indexes** section:

- No secondary indexes found.
- [Create a new secondary index →](#)

**Servers used by this table** section:

Shard	Documents	Primary replica	Status
Shard 1	~0 documents	4da81fc343b3_ron	ready (green)
		c7b765373fff_yxm	disconnected (red)
Shard 2	~0 documents	fec31920892e_bfe	ready (green)
		fc46b239242f_peu	ready (green)

At the bottom of the interface, there are links to 'Documentation', 'API', 'Google Groups', '#rethinkdb on freenode', 'Github', and 'Community'.

The server then reconnects, and everything recovers immediately. But this is a specific feature that will vary according to the database server and cluster type. Your operations' team should be comfortable with the database server chosen to run in production.

We now have a production-ready environment that we can tweak and scale to our needs.

## Moving to Kubernetes

Kubernetes began with a group of Google developers from a previous Google system called Borg. Its goal was, and is, to help in the deployment, scaling, and maintenance of applications. When it was first announced in 2014, there was no open source alternative. At the time, there was no Docker swarm, Docker networks, or Docker services.

Let's see what changes we need to make to our microservice in order to successfully run it using Kubernetes. But first, we need to clarify some of the concepts that are used in Kubernetes:

- **Pods:** A Pod consists of one or more containers that share some resources, and because of that, need to be located on the same host. A Pod is assigned a unique network address to avoid port collision. Note that several Pods of your deployment may be created, each with a different address.
- **Labels:** Kubernetes allows us to assign several labels to Pods in order to create groups of different kinds of components, such as frontend and backend, production and staging.
- **Services:** A Service is a group of Pods that work together, like our microservice and the database server. You can create a Service by defining a Label of Pods.

## Deploying with Kubernetes

That's not all of the concepts needed, but you may see some similarities and differences between Docker and Kubernetes. Actually, Kubernetes uses Docker and adds a few tools to enhance the deployment, scaling, and monitoring of containers.

The best way to start with Kubernetes is to install `minikube`. This is a tool to run Kubernetes locally on a single virtual machine:

**Minikube**

build passing codecov 28% go report A+



**What is Minikube?**

Minikube is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a VM on your laptop for users looking to try out Kubernetes or develop with it day-to-day.

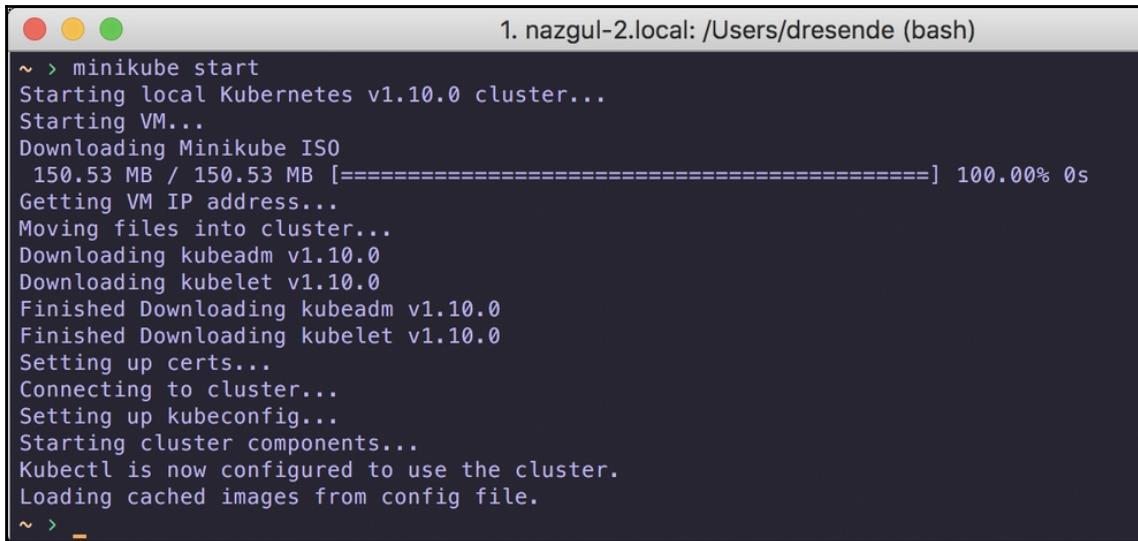
Follow the installation instructions. If you're on macOS and have `homebrew` installed, you just need to run:

```
brew cask install minikube
```

After installing `minikube`, you have to start it by running:

```
minikube start
```

You have to wait a minute or two before everything is set up:

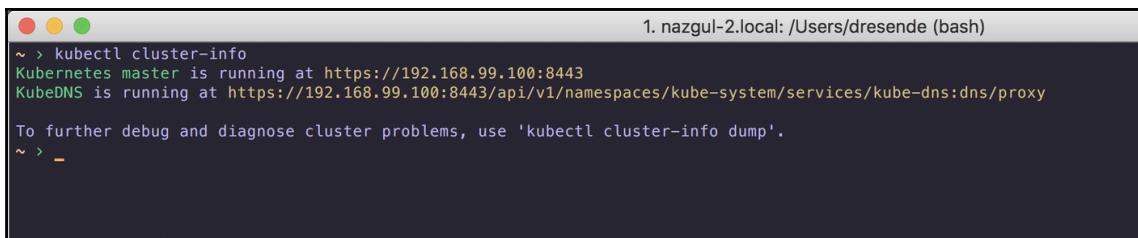


```
1. nazgul-2.local: /Users/dresende (bash)
~ > minikube start
Starting local Kubernetes v1.10.0 cluster...
Starting VM...
Downloading Minikube ISO
 150.53 MB / 150.53 MB [=====] 100.00% 0s
Getting VM IP address...
Moving files into cluster...
Downloading kubeadm v1.10.0
Downloading kubelet v1.10.0
Finished Downloading kubeadm v1.10.0
Finished Downloading kubelet v1.10.0
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
~ > -
```

We can see that our Kubernetes cluster is up and running by running:

```
kubectl cluster-info
```

This is a cluster of nodes that run containers, similar to Docker swarm. You should get a positive response as follows:



```
1. nazgul-2.local: /Users/dresende (bash)
~ > kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
~ > -
```

You're now able to access the Kubernetes dashboard by running:

```
minikube dashboard
```

Your browser should open something along the following lines:

The screenshot shows the Kubernetes Dashboard's 'Overview' page. On the left, there's a sidebar with sections for Cluster (Namespaces, Nodes, Persistent Volumes, Roles, Storage Classes), Namespace (default selected), and Workloads (Cron Jobs, Daemon Sets, Deployments, Jobs, Pods, Replica Sets, Replication Controllers). The main area has two tabs: 'Discovery and Load Balancing' and 'Config and Storage'. Under 'Discovery and Load Balancing', the 'Services' section lists a single service named 'kubernetes' with IP 10.96.0.1, port 443 TCP, and provider kubernetes. It was created 36 minutes ago. Under 'Config and Storage', the 'Secrets' section lists a secret named 'default-token-h972n' of type kubernetes.io/service-account-token, also created 36 minutes ago.

You can use the interface to monitor all of Kubernetes, but also to create services and deployments. Since our service depends on RethinkDB right now, we first need to ensure we have that up and running.

To begin, hit the **Create** button in the top right-hand corner of the page. On the text input that shows appears, write the following YAML configuration:

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: rethinkdb-master
spec:
  serviceName: rethinkdb-master
  replicas: 1
  template:
    metadata:
      labels:
        app: rethinkdb-master
  spec:
```

```
hostname: rethinkdb-master
containers:
- name: rethinkdb
  image: rethinkdb:2.3.6
  command: ["rethinkdb"]
  args:
  - --bind
  - "all"
  - --canonical-address
  - "rethinkdb-master:29015"
  - --canonical-address
  - "${MY_POD_IP}:29015"
  volumeMounts:
  - name: pdb-local-data
    mountPath: /data
  env:
  - name: MY_POD_NAME
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: MY_POD_IP
    valueFrom:
      fieldRef:
        fieldPath: status.podIP
  volumes:
  - name: pdb-local-data
    hostPath:
      path: /var/data/rethinkdb

---
apiVersion: v1
kind: Service
metadata:
  name: rethinkdb-master
  labels:
    app: rethinkdb-master
spec:
  ports:
  - port: 28015
    name: pdb-api
  - port: 29015
    name: pdb-cluster-api
  selector:
    app: rethinkdb-master
```

Hit **Upload** and wait a few seconds. You should have a running RethinkDB database in no time:

The screenshot shows the Kubernetes Dashboard's Overview page. On the left, there's a sidebar with 'Cluster' and 'Namespaces' sections, and a dropdown for the namespace set to 'default'. The main area has a 'Workloads' section with two green circular charts: one for 'Pods' at 100.00% and one for 'Stateful Sets' at 100.00%. Below this are two tables: 'Pods' and 'Stateful Sets'. The 'Pods' table shows a single row for 'rethinkdb-master-0' which is running on 'minikube'. The 'Stateful Sets' table shows a single row for 'rethinkdb-master' with 1 pod, labeled 'app: rethinkdb-master', created 'a minute' ago and using the image 'rethinkdb:2.3.5'. At the bottom, there's a 'Discovery and Load Balancing' section.

To check that our RethinkDB server is running correctly, we can add port forwarding from the outside of the cluster. First, let's list our services:

```
1. nazgul-2.local: /Users/dresende (bash)
~ > kubectl get service
NAME      TYPE      CLUSTER-IP     EXTERNAL-IP   PORT(S)      AGE
kubernetes   ClusterIP  10.96.0.1    <none>        443/TCP     1h
rethinkdb-master   ClusterIP  10.109.8.171  <none>        28015/TCP,29015/TCP  8m
~ > -
```

Our service is there. There are some endpoints, but the Administration Console isn't actually there. Let's forward that local port, which is 5000, to port 8080:

```
1. nazgul-2.local: /Users/dresende (kubectl)
~ > kubectl get service
NAME          TYPE      CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
kubernetes    ClusterIP  10.96.0.1    <none>       443/TCP       1h
rethinkdb-master ClusterIP  10.109.8.171  <none>       28015/TCP,29015/TCP   8m
~ > kubectl port-forward rethinkdb-master-0 5000:8080
Forwarding from 127.0.0.1:5000 -> 8080
Forwarding from [::1]:5000 -> 8080
-
```

While keeping the command running, open a new browser tab and head to port 5000 of the localhost. You should see the RethinkDB Administration Console:

The screenshot shows the RethinkDB Administration Console interface. At the top, there are navigation links: Overview - Kubernetes Dashboard, RethinkDB Administration Con, and a user icon labeled Diogo. Below the header, there is a search bar and a dashboard summary.

**Dashboard Summary:**

- Connected to rethinkdb\_master...
- Issues: No issues
- Servers: 1 connected
- Tables: 0/0 ready

**Cluster Performance:**

Cluster performance graph showing Reads/sec: 0 and Writes/sec: 0. The Y-axis ranges from 10K to 20K.

Servers	Tables	Indexes	Resources
1 server connected	0 tables ready	0 secondary indexes	0% cache used
0 servers missing	0 tables with issues	0 indexes building	0 Bytes disk used

We can now head to our microservice and try to deploy it, similar to what we did previously with Docker. First, create the `imagini` database on the Administration Console:

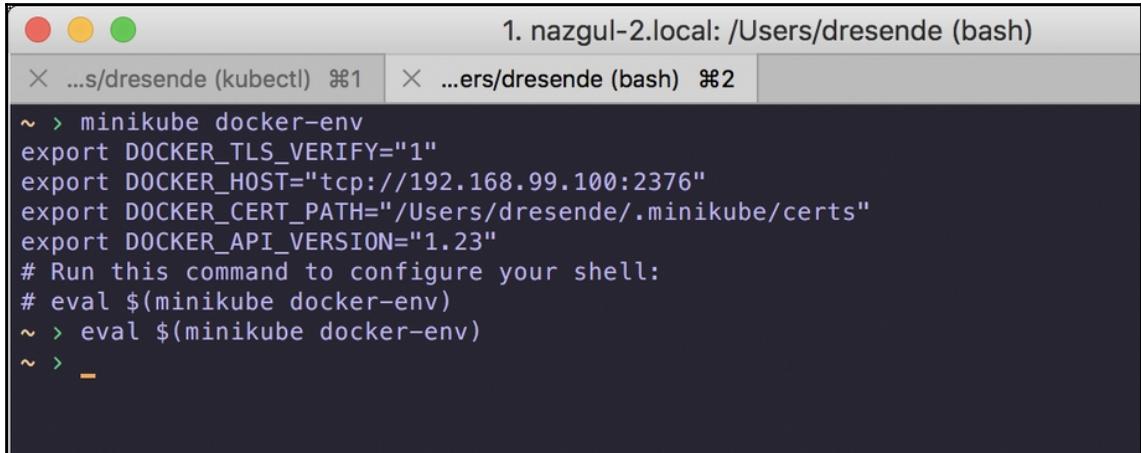
The screenshot shows the RethinkDB Administration Console interface. At the top, there are navigation links: Overview - Kubernetes Dashboard, RethinkDB Administration Console, and a search bar. Below the header, there are four status indicators: 'Connected to rethinkdb\_master...', 'Issues No issues', 'Servers 1 connected', and 'Tables 0/0 ready'. The main area is titled 'Tables in the cluster' and lists two databases: 'imagini' and 'test'. Each database has a 'DATABASE' button, a name field ('imagini' or 'test'), and buttons for '+ Add Table' and 'Delete Database'. Below each database, a message states 'There are no tables in this database.' At the bottom of the page, there are links for Documentation, API, Google Groups, #rethinkdb on freenode, Github, and Community.

To keep it simple, we're keeping the RethinkDB connection hardcoded on our code. I'll leave it up to you to try it out later on with Kubernetes to see how you can have a persistent volume, have your settings there, and the database.

We just have to change our line to:

```
rethinkdb.connect({ host: "rethinkdb-master", db: "imagini" }, (err, db) =>
{
```

Since we have a local image, we need to build it just like we did before. Let's build it inside our minikube. First, let's make sure that we're going to push our commands to minikube and not our Docker base:



The screenshot shows a terminal window with two tabs. The active tab is titled "1. nazgul-2.local: /Users/dresende (bash)". It contains the following command and its output:

```
~ > minikube docker-env
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/dresende/.minikube/certs"
export DOCKER_API_VERSION="1.23"
# Run this command to configure your shell:
# eval $(minikube docker-env)
~ > eval $(minikube docker-env)
~ >
```

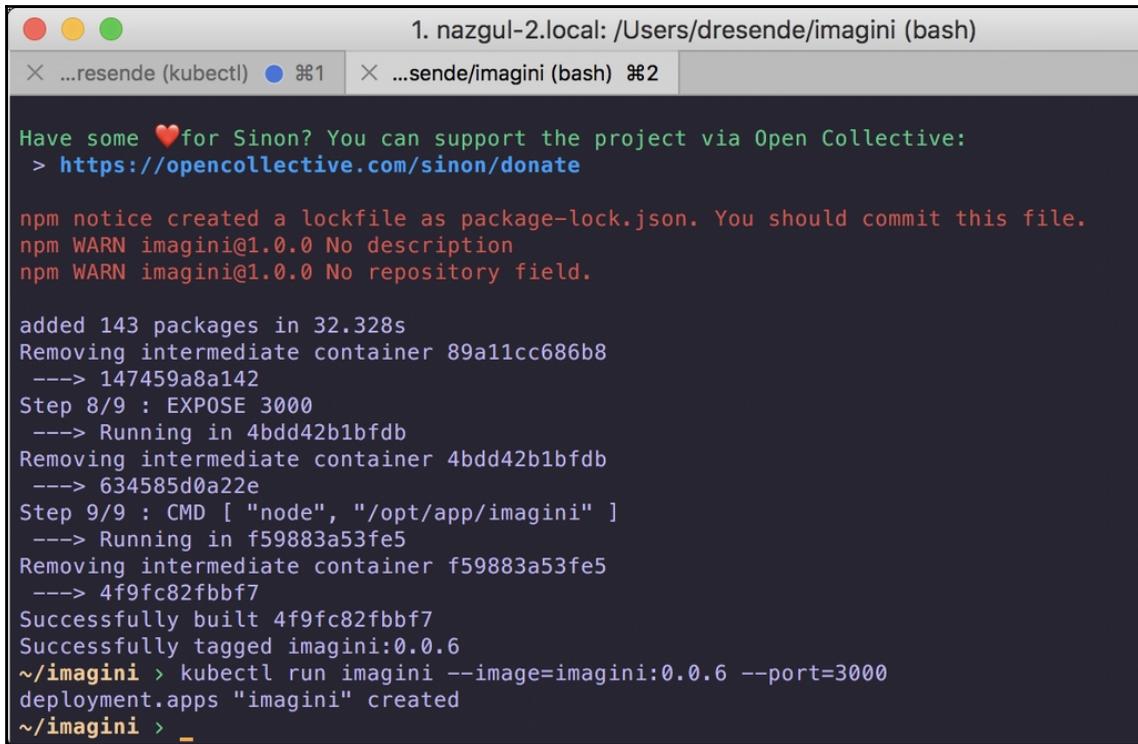
Then, let's build our image using the same Docker command:

```
docker build -t imaginini:0.0.6 .
```

After building it, let's begin the deployment:

```
kubectl run imaginini --image=imaginini:0.0.6 --port=3000
```

If everything runs smoothly, you should see a message like the following:



The screenshot shows a terminal window titled "1. nazgul-2.local: /Users/dresende/imagini (bash)". It has two tabs: "...resende (kubectl)" (active) and "...sende/imagini (bash)". The terminal output is as follows:

```
Have some ❤️ for Sinon? You can support the project via Open Collective:  
> https://opencollective.com/sinon/donate  
  
npm notice created a lockfile as package-lock.json. You should commit this file.  
npm WARN imagini@1.0.0 No description  
npm WARN imagini@1.0.0 No repository field.  
  
added 143 packages in 32.328s  
Removing intermediate container 89a11cc686b8  
--> 147459a8a142  
Step 8/9 : EXPOSE 3000  
--> Running in 4bdd42b1bfdb  
Removing intermediate container 4bdd42b1bfdb  
--> 634585d0a22e  
Step 9/9 : CMD [ "node", "/opt/app/imagini" ]  
--> Running in f59883a53fe5  
Removing intermediate container f59883a53fe5  
--> 4f9fc82fbbf7  
Successfully built 4f9fc82fbbf7  
Successfully tagged imagini:0.0.6  
~/imagini > kubectl run imagini --image=imagini:0.0.6 --port=3000  
deployment.apps "imagini" created  
~/imagini > _
```

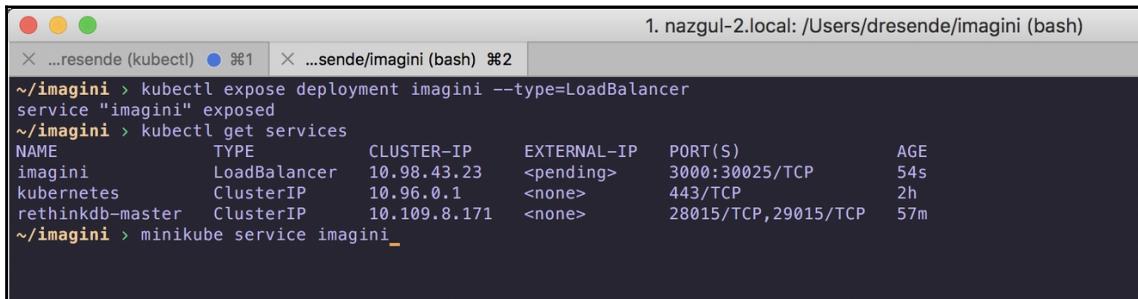
If you refresh the Administration Console, you now have **Deployments** and a new Pod:

The screenshot shows the Kubernetes Administration Console interface. On the left, there's a sidebar with 'Cluster' and 'Namespace' sections. Under 'Cluster', options like Namespaces, Nodes, Persistent Volumes, Roles, and Storage Classes are listed. Under 'Namespace', 'default' is selected. The main area is titled 'Overview' and contains two main sections: 'Workloads' and 'Deployments'. The 'Workloads' section has a chart titled 'Workloads Statuses' showing 100.00% for Deployments, Pods, Replica Sets, and Stateful Sets. Below it, the 'Deployments' section lists one deployment named 'imaginini' with 1 pod, 26 seconds age, and image 'imaginini:0.0.6'. The 'Pods' section lists two pods: 'imaginini-865f64947c-gk4p5' and 'rethinkdb-master-0', both running on 'minikube' with 0 restarts and 26 seconds and 51 minutes age respectively.

But Pods are, by default, only visible on the internal network. We need to expose our `imaginini` to the exterior. Do this by running:

```
kubectl expose deployment imaginini --type=LoadBalancer
```

If you check the services, you'll see that our service will expose on port 3000, but it's currently pending. This is because we're using minikube and it has no LoadBalancer. For example, on a cloud provider, this would launch a real load balancer with an external address and would route traffic to the inside:



```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~ ...resende (kubectl)  ● %1 | X ...sende/imagini (bash) ⌘2
~/imagini > kubectl expose deployment imagini --type=LoadBalancer
service "imagini" exposed
~/imagini > kubectl get services
NAME           TYPE      CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
imagini        LoadBalancer  10.98.43.23  <pending>    3000:30025/TCP  54s
kubernetes     ClusterIP   10.96.0.1    <none>       443/TCP       2h
rethinkdb-master ClusterIP  10.109.8.171  <none>       28015/TCP,29015/TCP  57m
~/imagini > minikube service imagini
```

In our case, we can use the direct port, 30025, which was assigned randomly. We can use the last command you see in the preceding screenshot in order to launch a browser window and point to our service.

You could do all of this using the administration tools from the browser or from the console. Some may be easier on the console, but it's up to you.

This is not an extensive introduction to Kubernetes. The objective was actually to see how easy it was to just make a few changes to our service and get it up and running on a new environment. That's one of the powers of containers.

## Summary

Scaling an application used to be hard and complex. Today, with more information passing through our applications, and with a constantly increasing number of connected users, scaling is now imperative.

Thankfully, containers showed up a couple of years ago and helped solve this problem in a simpler way. There are still obstacles, but with the tools provided by Docker, Kubernetes, and others, replicating our applications and microservices became something that you can do almost without changing your code.

In the next chapter, we'll see what we need to do to have our service be cloud native while deploying our microservice to the **Google Cloud Platform (GCP)**.

# 10

## Cloud-Native Microservices

Let's recap what we have done so far. We started by looking at the advantages of building applications based on microservices. Then, we looked at several tools that can help us start building our microservices. We stood with Express and embarked on building a simple microservice.

We learned the basics of state and security by interconnecting our microservice and a database server. We've chosen a test suite and added tests until we had a very good code coverage.

Finally, we learned how to deploy our microservice using containers. We then explored how we could scale our microservice using replicas across different sites.

When we deploy our microservice inside a container, we're opening a new world of possibilities. There are plenty of cloud providers that support containers. This means that as soon as we have our microservice running on a local container, we can replicate and do the same on many of these providers in no time.

You might have found that developing using containers gives you consistent and predictable behavior with the application. This also extends to the operations when they need to deploy.

Containers give them a much simpler and faster method to deploy any application. It's almost like your application is already native to the cloud environment, and they just need to specify a few details and that's it, it's deployed.

In this chapter, to make our microservice cloud-native, we're going to use the **Google Cloud Platform (GCP)**, and:

- Create a new project
- Deploy a database service
- Create a Kubernetes cluster
- Create our microservice files on the cluster
- Deploy our microservice to the cluster

## Preparing for cloud-native

But what is cloud-native? What makes an application cloud-native? We say an application runs in the cloud when it's running outside our premises. More than that, it denotes when an application is supposedly running and spread across different locations and is resilient to failures.

A cloud-native application or microservice is designed from scratch to run in the cloud and take advantage of that computing model. It is able to scale and operate even if parts of its infrastructure fail. Being cloud-native is about how we deploy our microservice, not where.

And, we're already doing that! Since we first deployed using Docker, we have already started and restarted our microservice, and rebuilt new images and replicated without the worry of where, but instead, how.

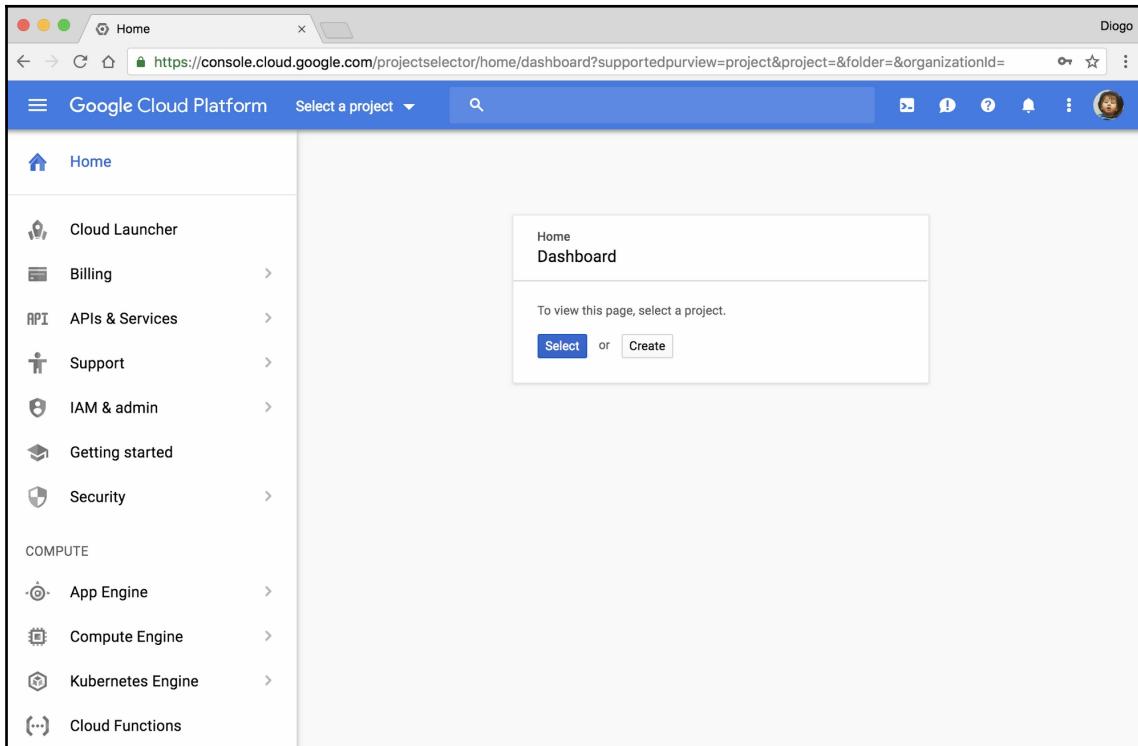
Thinking generically, there are a few points to take into account when designing our microservice, or anything that will be cloud-native, for that matter:

- Understand what data your microservice will handle, and what kind of structures and relations will there be. Does a relational database such as MySQL or PostgreSQL suffice? Do you need a more loose database server that can handle documents with different information structures, such as MongoDB or RethinkDB?  
These questions must be answered to narrow the options. There are plenty of other types of databases. After narrowing your choices, you should pick the one that gives you more resilience and that can be replicated the same way as your microservice, to handle some degree of failure.
- Find out what other dependencies you might need. If you have choices, go for the ones that are not dependent on a specific operative system. Below the cloud servers, there are different operative systems and you don't want to be forced to choose any type.
- Avoid going non-standard. More specifically, try to take the most out of current standard protocols. Because you can deploy to geographically distant hosts, connectivity and policy rules may introduce restrictions to traffic, so keep with standards such as HTTP in order to communicate between microservices.
- Design your microservices to be able to scale horizontally. It's not uncommon for microservices to not scale linearly just by adding replicas, but try to design them to be able to distribute loads more evenly. One easy path is to have a layer of proxies on top.

## Going cloud-native

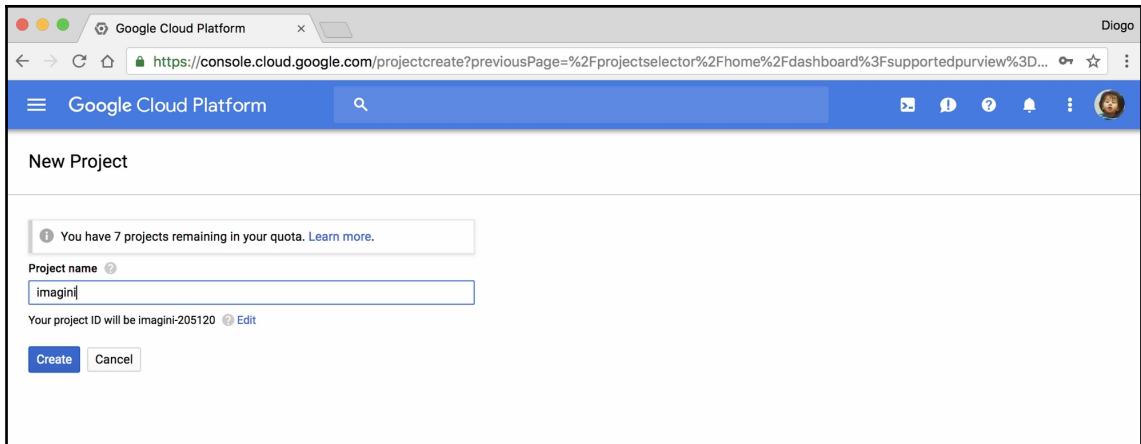
Now that we're able to deploy using containers, and we have used Kubernetes already on a local virtual machine, let's see how we can do the same on a cloud environment. You can choose any type of cloud provider, as long as you follow the considerations we just discussed in the previous section.

We're going to use GCP. There's a one-year trial with free credit available, so you can try it out while you read this. Head to the website and register if you haven't done it before. You'll probably be asked to enter a payment method, but don't worry, they won't charge unless you exceed the free credit:

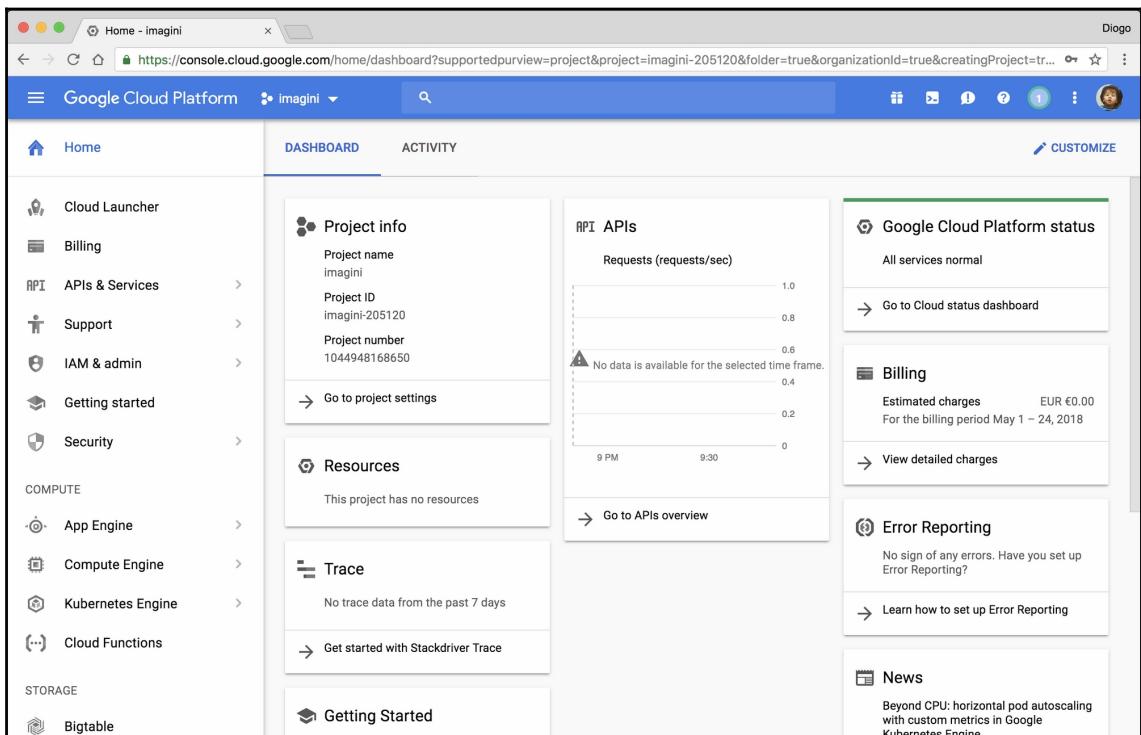


## Creating a new project

You now need to create a project. Just follow the instructions, set a name, and hit the **Create** button:

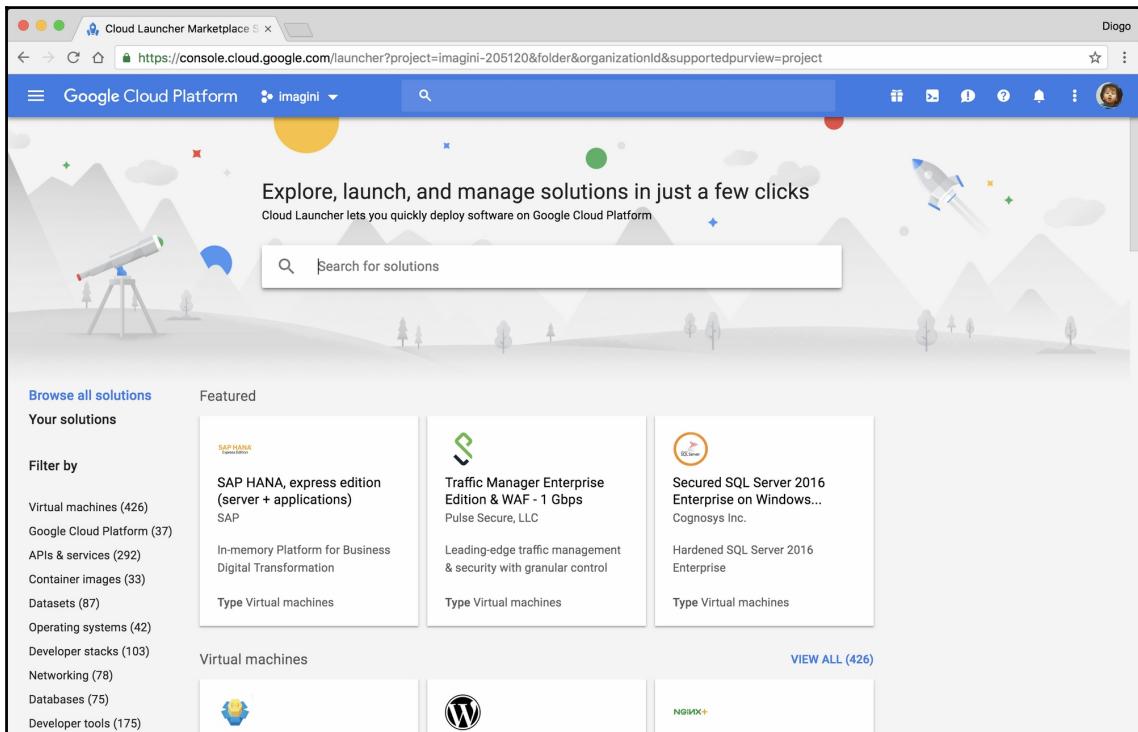


You'll then have access to the project dashboard. You can check resource usage and project status, as well as billing information:



## Deploying a database service

On the left-hand side, there are the main navigation options, one of which, Cloud Launcher, lets us easily set up services that we need. Since we used RethinkDB in the previous chapter, let's now try MySQL again. We need a database server, so let's head to Cloud Launcher:



On the sidebar, there are a couple of service categories to choose from. Let's use the filter on top to search for **mysql**:

The screenshot shows the Google Cloud Platform Cloud Launcher interface. The search bar at the top contains the query "mysql". Below the search bar, the results are displayed under the heading "43 results". The results are categorized by type and include various MySQL offerings from Google Cloud Platform and Cognosys Inc. The categories listed on the left are:

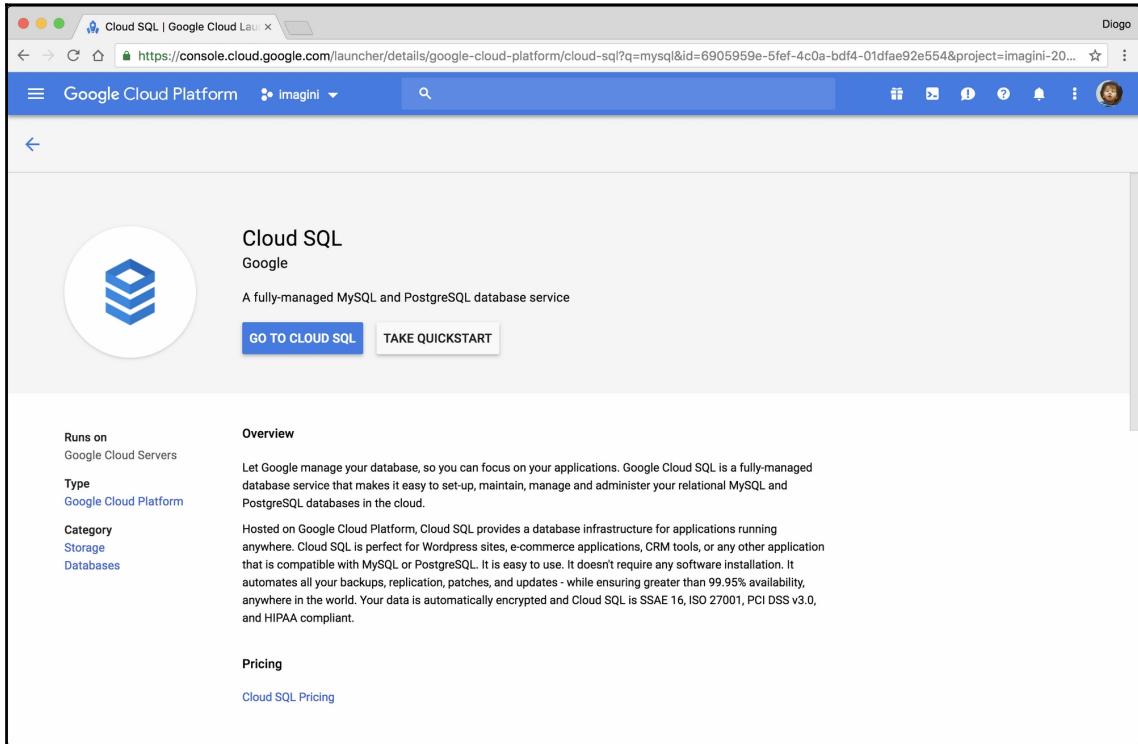
- TYPE**
  - Container images (1)
  - Google Cloud Platform (2)
  - APIs & services (2)
  - Virtual machines (38)
- CATEGORY**
  - Analytics (1)
  - Big data (1)
  - Blog & CMS (4)
  - CRM (8)
  - Databases (16)
  - Developer stacks (8)
  - Developer tools (14)
  - Security (1)
  - Social (12)
  - Storage (2)
  - Other (2)

The results are as follows:

Type	Description	Provider	Type	
Cloud SQL	A fully-managed MySQL and PostgreSQL database service	Google	Secured MySQL 5.6 on Red Hat Enterprise Linux 7	Cognosys Inc.
Cloud SQL	A fully-managed MySQL and PostgreSQL database service	Google	Secured MySQL 5.7 on Red Hat Enterprise Linux 7	Cognosys Inc.
Secured MySQL 5.6 on Windows 2012 R2	Hardened MySQL 5.6 on Windows 2012 R2	Cognosys Inc.	Secured MySQL 5.7 on Windows 2012 R2	Cognosys Inc.
Secured MySQL 5.6 on Windows 2012 R2	Hardened MySQL 5.6 on Windows 2012 R2	Cognosys Inc.	Secured MySQL with IIS on Windows 2012 R2	Cognosys Inc.
Secured MySQL 5.7 on Windows 2012 R2	Hardened MySQL 5.7 on Windows 2012 R2	Cognosys Inc.	Secured MySQL with IIS on Windows 2012 R2	Cognosys Inc.

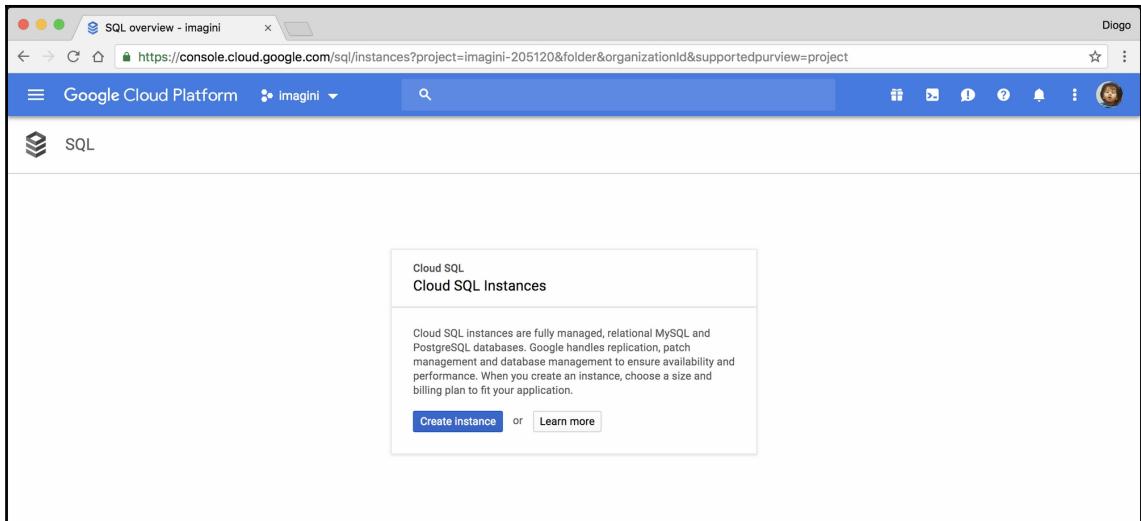
The URL shown in the browser is <https://console.cloud.google.com/launcher/browse?q=mysql&project=imagini-205120&folder&organizationId&supportedpurview=project>.

One of the options, in my case the first, is **Cloud SQL**, which is a Google service used to run a fully-managed MySQL or PostgreSQL database service. It alleviates the burden of configuration and maintenance. Click on that option:

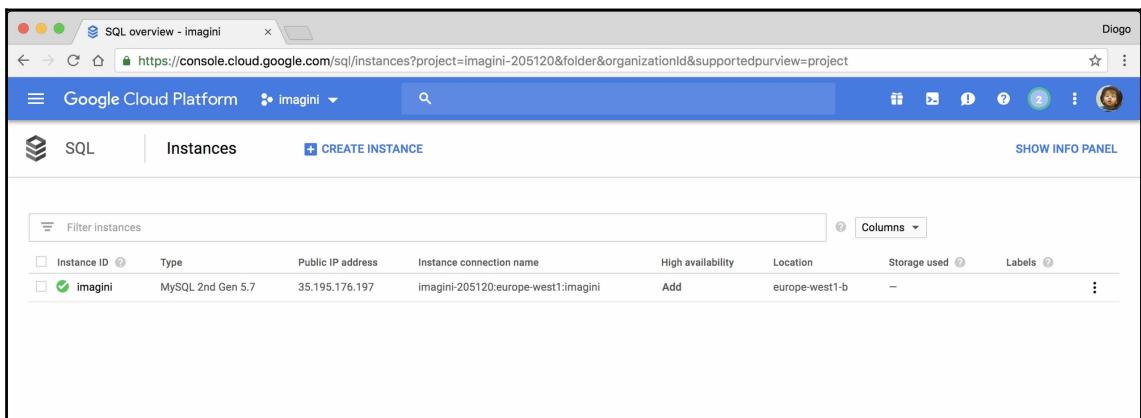


The screenshot shows a browser window for the Google Cloud Platform Cloud SQL page. The URL is https://console.cloud.google.com/launcher/details/google-cloud-platform/cloud-sql?q=mysql&id=6905959e-5fef-4c0a-bdf4-01dfaef2e554&project=imagini-20... The page title is "Cloud SQL | Google Cloud Launcher". The main content area features a large circular icon with a blue 3D cube representing Cloud SQL. To its right, the text "Cloud SQL" and "Google" is displayed, followed by a subtitle "A fully-managed MySQL and PostgreSQL database service". Below this are two buttons: "GO TO CLOUD SQL" (blue) and "TAKE QUICKSTART" (white). On the left side, there's a sidebar with categories: "Runs on Google Cloud Servers", "Type Google Cloud Platform", "Category Storage", and "Databases". On the right side, under the "Overview" section, there's a detailed description of the service, mentioning it's hosted on Google Cloud Platform and provides a database infrastructure for various applications. It also highlights its compatibility with MySQL and PostgreSQL, ease of use, and automation of backups, replication, patches, and updates.

There's an overview with an explanation of the service, as well as pricing if you want to consult it. Because we're using free credit, that is not a concern for now. Click on **GO TO CLOUD SQL**:



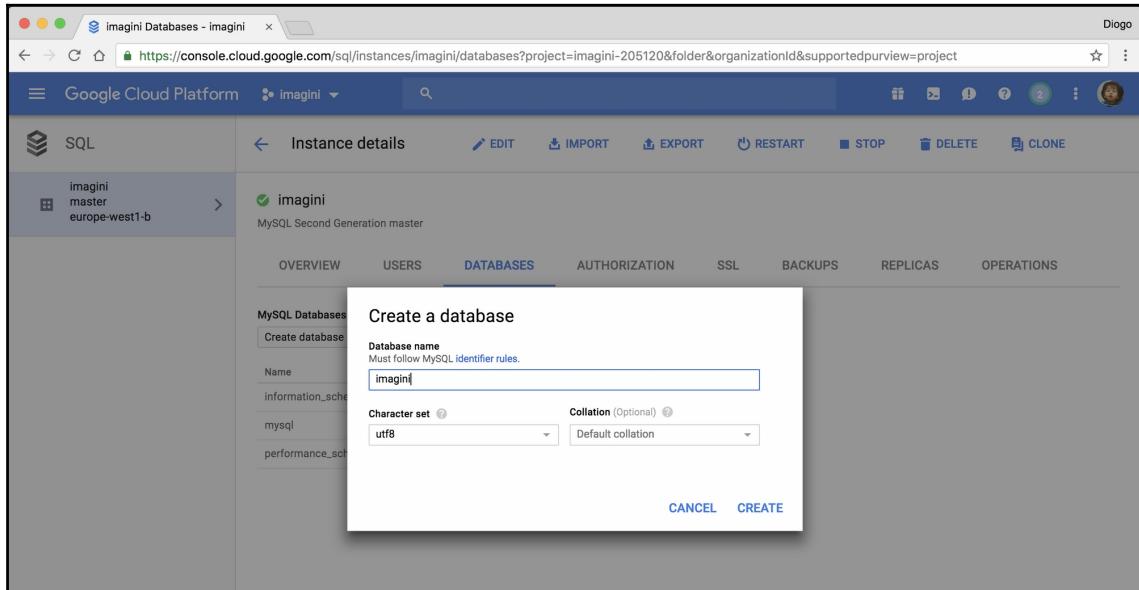
Because we're using the service for the first time, there's no service instance. Click on "Create Instance" to create a new one, follow the instructions, and create the instance. You'll be asked for a password for the default root account; don't forget it. At the end, you should see an instance list. Wait until your instance is fully up and running:



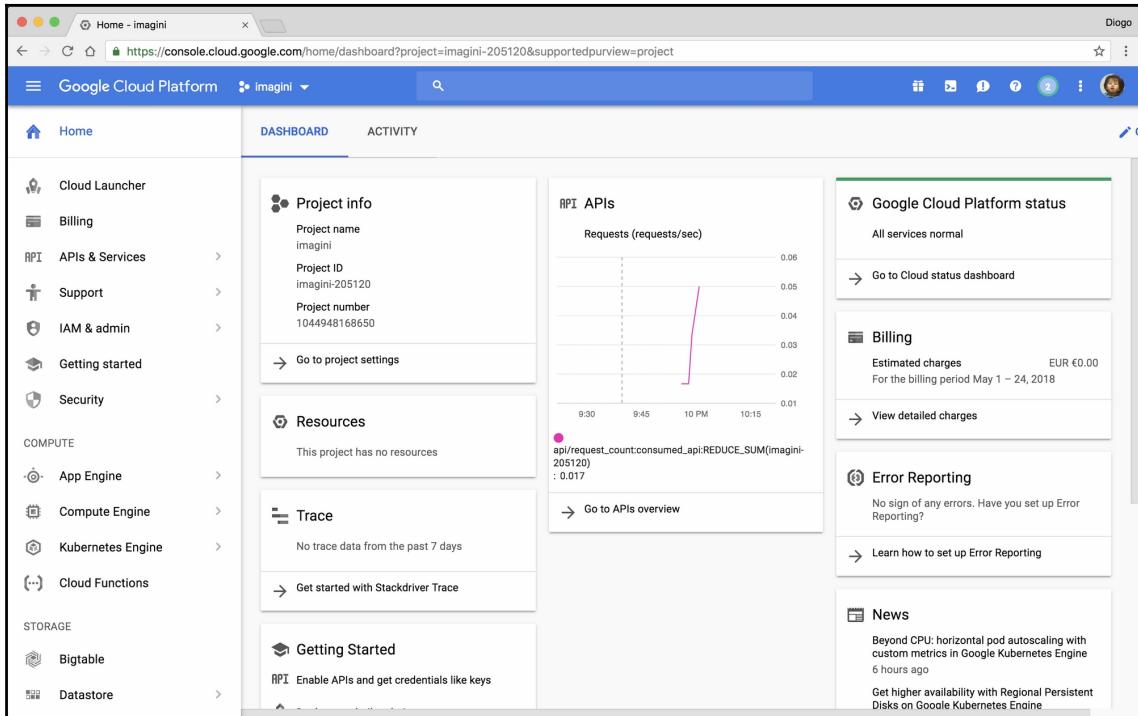
You can click on the instance ID and see more information about it. You'll have access to a service dashboard, as well as be able to import and export:

The screenshot shows the Google Cloud Platform SQL Instances Overview page. The instance 'imagini' is selected. The 'OVERVIEW' tab is active. A chart titled 'Storage usage' shows storage usage over time, with a red line indicating a peak at 10:05 on May 24, 2018, reaching 1164MB. Below the chart, there are sections for 'Connect to this instance' (Primary IPv4 address: 35.195.176.197, Instance connection name: imaginii-205120:europe-west1:imagini) and 'Configuration' (vCPUs: 1, Memory: 3.75 GB, HDD storage: 10 GB). The database version is listed as MySQL 5.7.

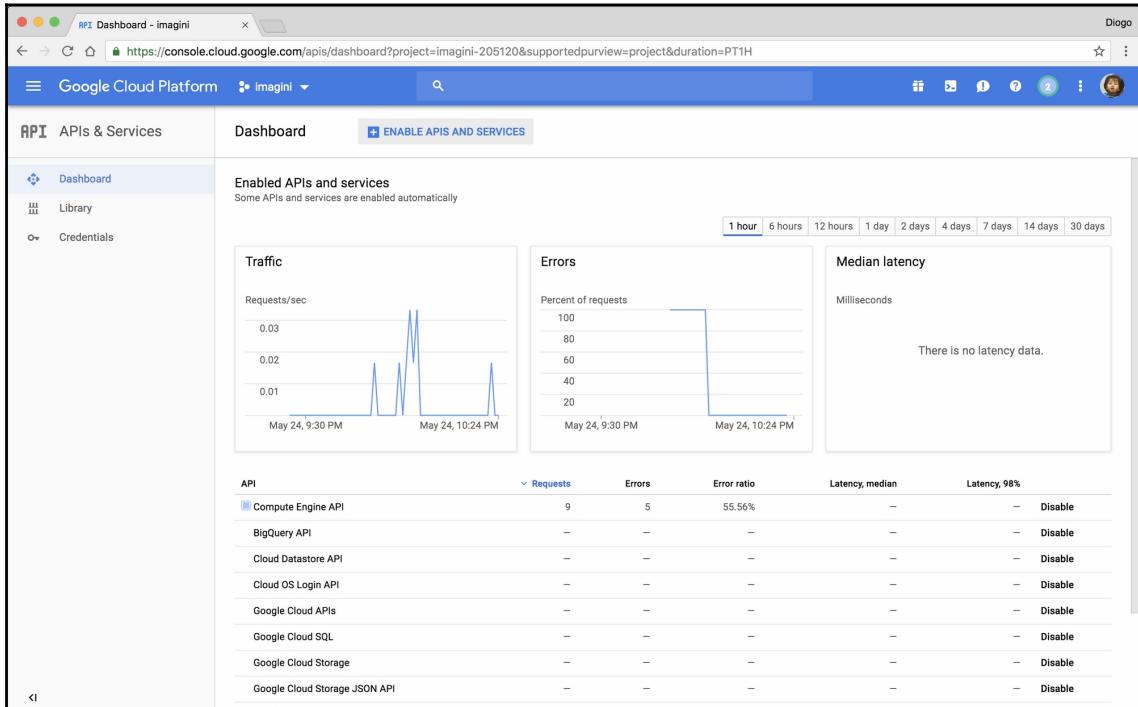
There's a **DATABASES** tab where we can create our microservice database. Head there and create it. Retain defaults on the character set and collation:



Our database is now ready. Let's head back to the project dashboard. Click on **Google Cloud Platform** in the top-left corner of the screen:



You'll notice that the **APIs** block on the middle column now has a line showing some activity. APIs are part of the infrastructure and, since we just created a service instance, there's some activity going on. Click on **Go to APIs overview**:



Here, you can monitor activity, traffic, and errors of all enabled and used APIs. Heading back to our project dashboard, and looking at the sidebar, you'll notice there are plenty of options available.

Scrolling down a bit, you'll find an **SQL** section, where you can find our previously created instance. If you need to go there in the future, this is how you find it:

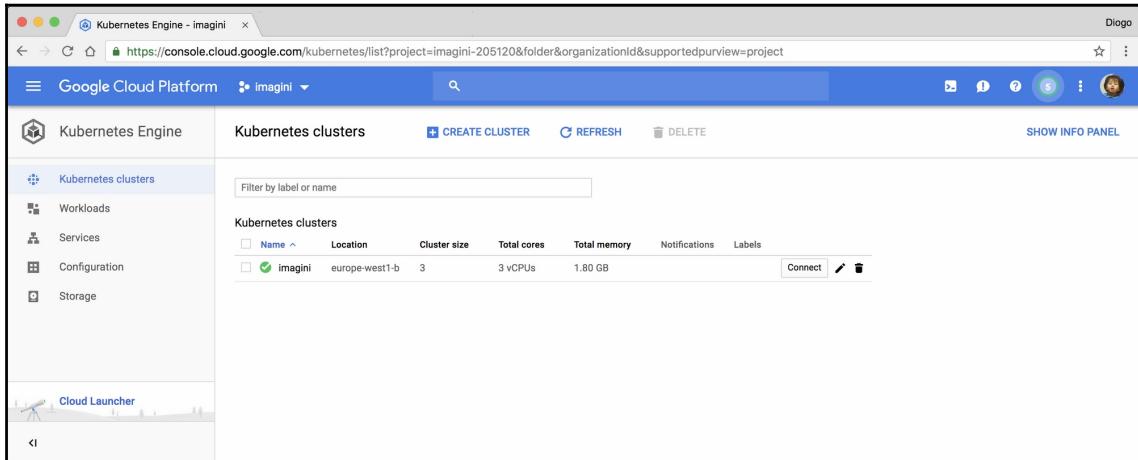
The screenshot shows the Google Cloud Platform dashboard for a project named 'imagini'. The left sidebar lists various services under 'COMPUTE', 'STORAGE', 'NETWORKING', and 'SQL'. The 'SQL' section is highlighted. The main content area displays 'Project info' (Project name: imagini, Project ID: imagini-205120, Project number: 1044948168650), 'API APIs' (Requests (requests/sec) chart showing two spikes at 10 PM and 10:30 PM), 'Google Cloud Platform status' (All services normal), 'Billing' (Estimated charges EUR €0.00 for May 1 - 24, 2018), 'Error Reporting' (No errors), and 'News' (Beyond CPU: horizontal pod autoscaling with custom metrics in Google Kubernetes Engine). A URL at the bottom of the dashboard is <https://console.cloud.google.com/sql?project=imagini-205120&folder&organizationId&supportedpurview=project>.

# Creating a Kubernetes cluster

Previously, for a few items, you may have noticed the **Kubernetes Engine** option. That's where we're heading to prepare our cluster:

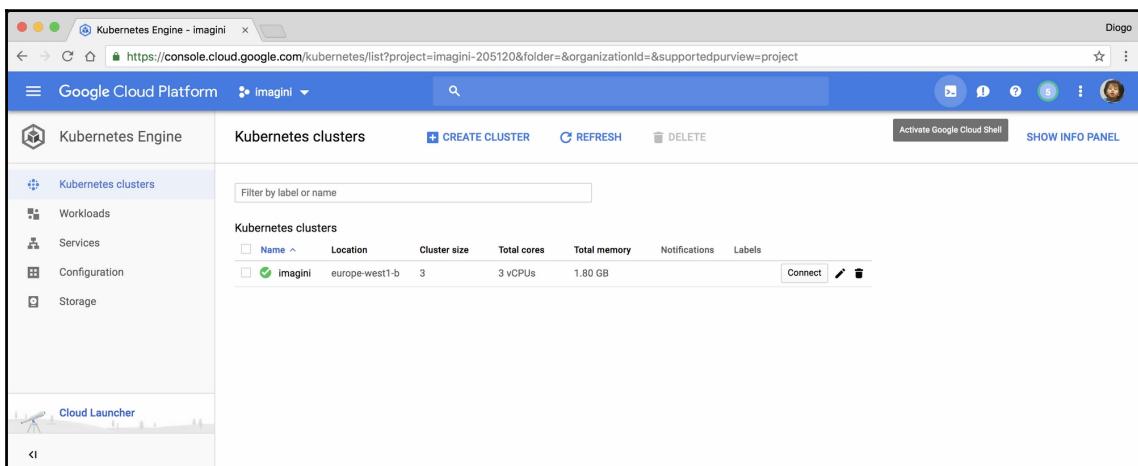
The screenshot shows the Google Cloud Platform dashboard for a project named 'imagini'. The left sidebar has a 'Kubernetes Engine' section with a dropdown menu open, showing 'Kubernetes clusters' as the selected option. Under 'Kubernetes clusters', it says 'Workloads' and 'Services', both showing 'has no resources'. There is also a note 'No trace data from the past 7 days' and a link 'Get started with Stackdriver Trace'. Below this is a 'Getting Started' section with a link 'Enable APIs and get credentials like keys'. The main content area on the right includes 'Project info' (Project name: imagini, Project ID: imagini-205120, Project number: 1044949186850), a 'API APIs' chart showing requests per second over time, and sections for 'Google Cloud Platform status' (All services normal), 'Billing' (Estimated charges: EUR €0.00 for May 1 - 24, 2018), 'Error Reporting' (No sign of any errors), and 'News' (Recent news items).

You'll be asked to create a new cluster. Choose the default options, or change the ones you want to try out, and create a cluster for our microservice to run on:



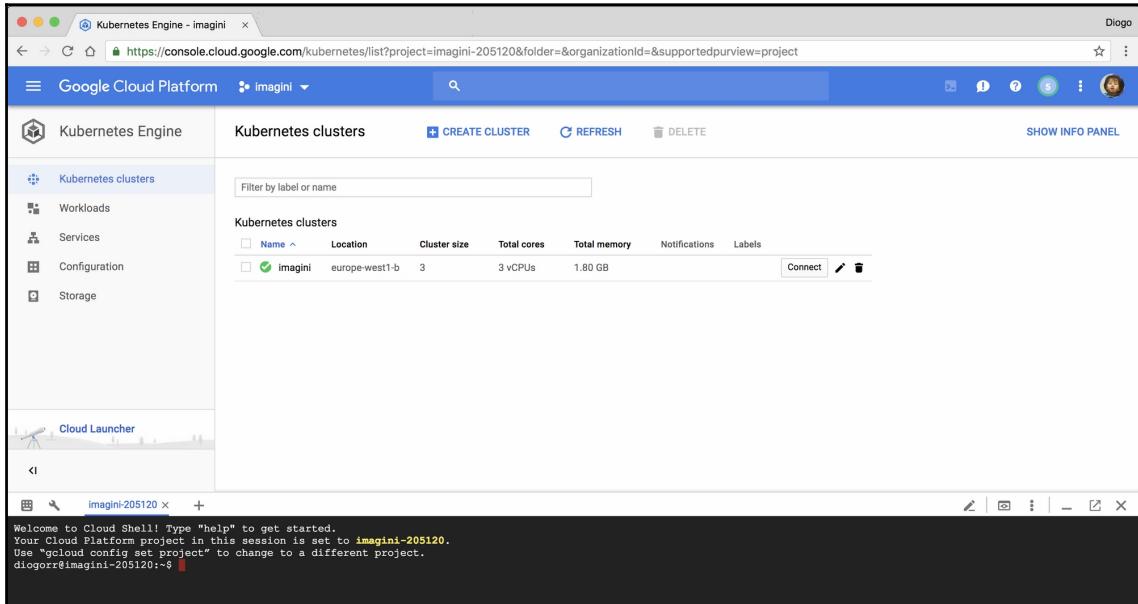
The screenshot shows the Google Cloud Platform interface for the Kubernetes Engine. The left sidebar has a 'Kubernetes Engine' section with 'Kubernetes clusters' selected. The main area displays a table of 'Kubernetes clusters' with one entry: 'imagine' located in 'europe-west1-b' with a cluster size of 3 nodes, 3 vCPUs, and 1.80 GB of memory. There are 'Connect', 'Edit', and 'Delete' buttons for this cluster. A 'Cloud Launcher' button is at the bottom of the sidebar.

We now have a Kubernetes cluster of three nodes to run our microservice. In the top-right corner of the screen, you'll find one important icon, which is the **Google Cloud Shell icon**:



This screenshot is identical to the one above, showing the same 'imagine' cluster in the Kubernetes Engine interface. However, a 'Activate Google Cloud Shell' button has been highlighted with a red box in the top right corner of the main content area. The rest of the interface, including the sidebar and cluster table, remains the same.

If you click on it, a console will open that gives you access to your platform using a terminal. You can do all kinds of stuff, such as monitoring containers, and checking connectivity:



This is a normal console, but with the `gcloud` and `kubectl` commands already installed. You can have those installed locally and manage your clusters remotely, or just access this console to make a quick change or checkup.

## Creating our microservice

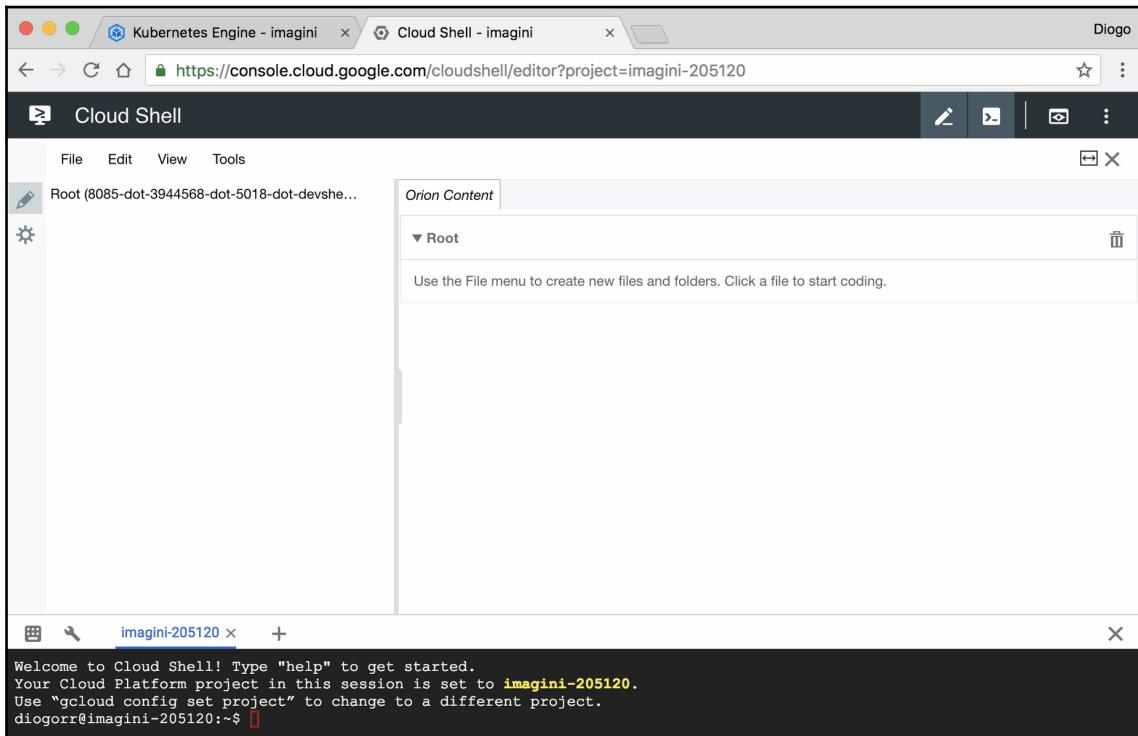
Similar to what we did before, these are the steps we need to perform before successfully deploying to our new cluster:

1. Create our docker image that will use our MySQL instance.
2. Create credentials to access the instance.
3. Create the configuration for our service.
4. Deploy it.

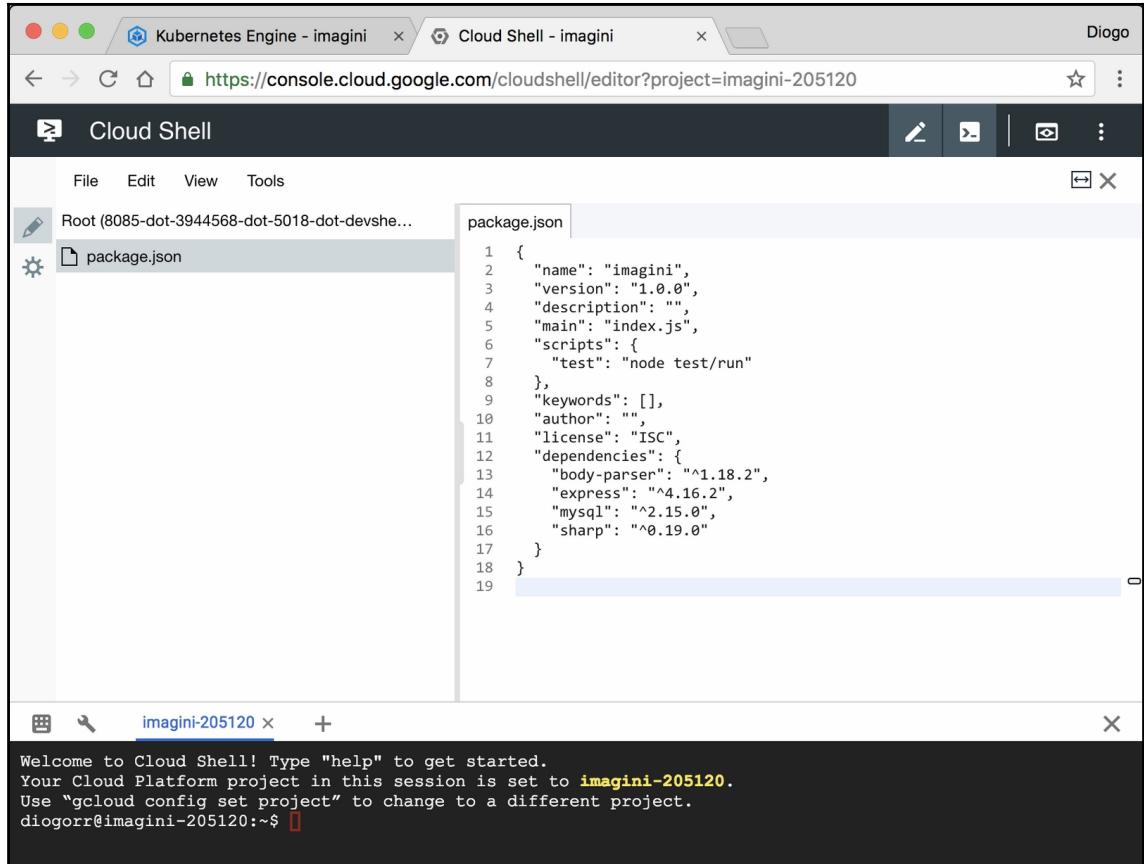
First, notice the pencil icon in the top-right corner of the console. It's an online editor we can use to create the files we need:

The screenshot shows the Google Cloud Platform interface for the Kubernetes Engine. On the left, there's a sidebar with icons for Kubernetes clusters, Workloads, Services, Configuration, and Storage. The main area is titled 'Kubernetes clusters' and shows a table with one row. The row for 'imagin1' has a green checkmark next to it, indicating it's selected. The table columns are Name, Location, Cluster size, Total cores, Total memory, Notifications, and Labels. The 'imagin1' row shows 'europe-west1-b' as the location, '3' as the cluster size, '3 vCPUs' as total cores, '1.80 GB' as total memory, and 'Pods unschedulable' under Notifications. There are 'Connect', 'Edit', and 'Delete' buttons at the end of the row. Below the table, a message says 'Welcome to Cloud Shell! Type "help" to get started.' and 'Your Cloud Platform project in this session is set to **imagin1-205120**. Use "gcloud config set project" to change to a different project.' A 'Launch code editor BETA' button is also visible.

Click on it. It should open a new tab with an empty editor. This represents the root folder of the console you have on the bottom:



Let's start by creating our package.json file. Go to **File** and click **New**, and then **File**. Type `package.json` and insert our dependencies. Remember, we're using MySQL this time:



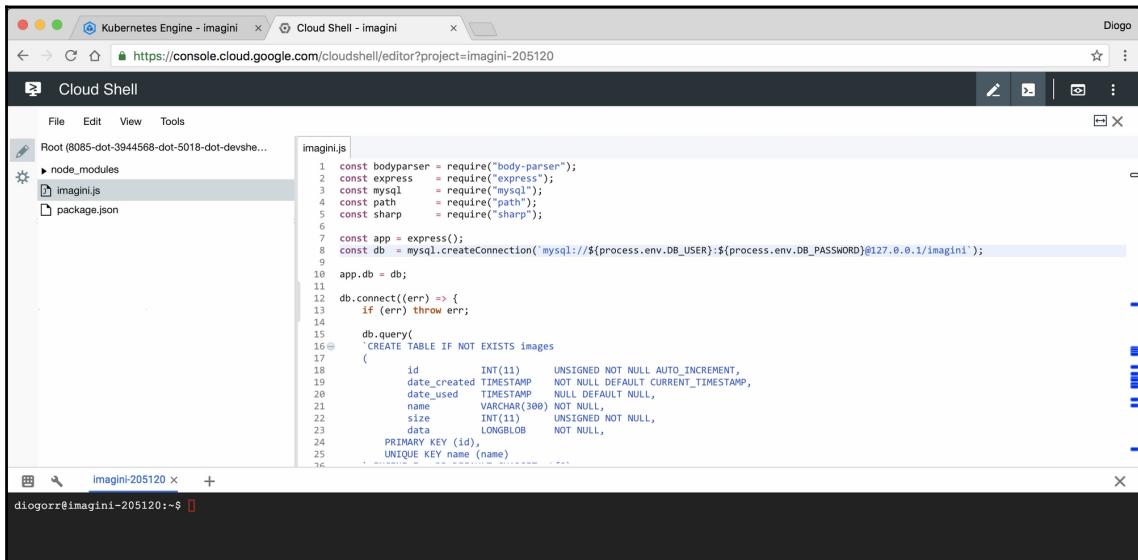
The screenshot shows the Google Cloud Platform Cloud Shell interface. At the top, there are two tabs: "Kubernetes Engine - imaginii" and "Cloud Shell - imaginii". The URL in the browser bar is <https://console.cloud.google.com/cloudshell/editor?project=imaginii-205120>. Below the tabs is a toolbar with icons for file operations. The main area is titled "Cloud Shell" and contains a "File" menu. On the left, there's a sidebar with icons for "Root (8085-dot-3944568-dot-5018-dot-devshe...)" and "package.json". The central workspace shows the "package.json" file open in a code editor. The file content is as follows:

```
1  {
2    "name": "imaginii",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "node test/run"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "body-parser": "^1.18.2",
14     "express": "^4.16.2",
15     "mysql": "^2.15.0",
16     "sharp": "^0.19.0"
17   }
18 }
19 }
```

At the bottom of the screen, there is a terminal window titled "imaginii-205120" with the following text:

```
Welcome to Cloud Shell! Type "help" to get started.  
Your Cloud Platform project in this session is set to imaginii-205120.  
Use "gcloud config set project" to change to a different project.  
diogorr@imaginii-205120:~$
```

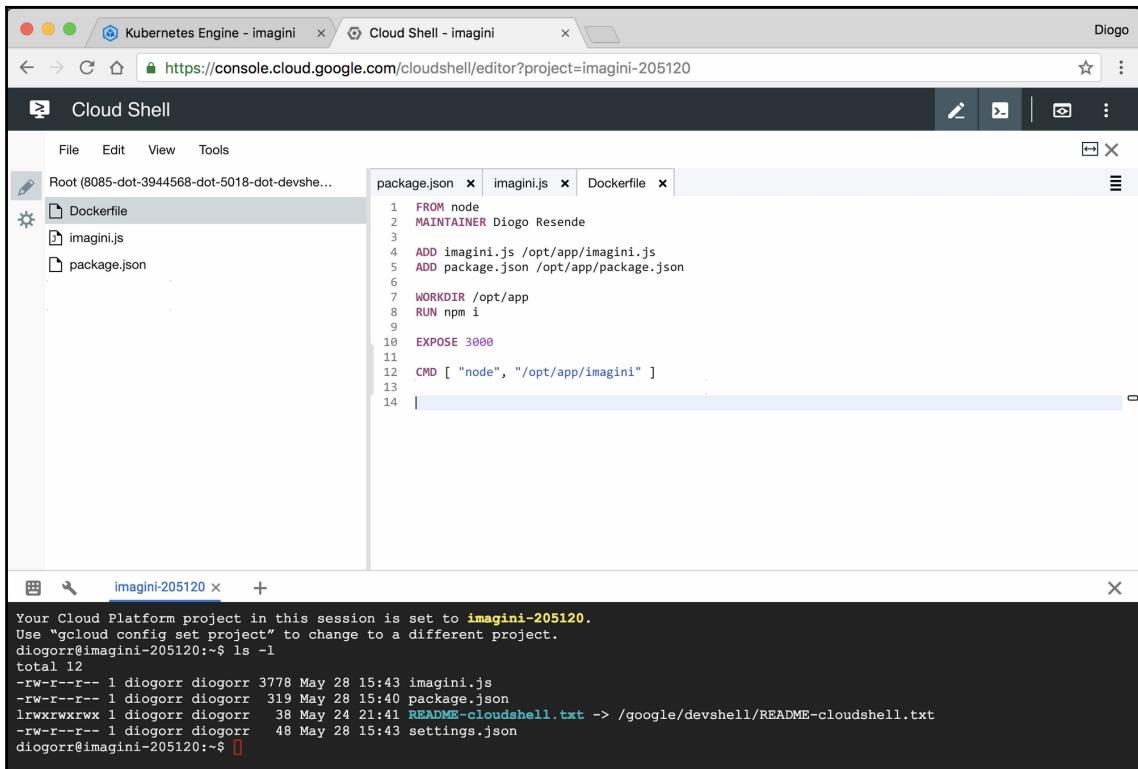
Then, create our service file. We're not using a `settings.json` file; we're going to use credentials passed to the container using an environment variable:



The screenshot shows a Cloud Shell interface with a tab titled "Kubernetes Engine - imagin1" and a sub-tab "Cloud Shell - imagin1". The URL is `https://console.cloud.google.com/cloudshell/editor?project=imagin1-205120`. The code editor window is titled "Cloud Shell" and contains the file "imagine.js". The code defines an Express application that connects to a MySQL database and creates a table named "images". The MySQL connection string uses environment variables `DB_USER` and `DB_PASSWORD`.

```
1 const bodyParser = require("body-parser");
2 const express = require("express");
3 const mysql = require("mysql");
4 const path = require("path");
5 const sharp = require("sharp");
6
7 const app = express();
8 const db = mysql.createConnection(`mysql://${process.env.DB_USER}:${process.env.DB_PASSWORD}@127.0.0.1/imagin1`);
9
10 app.db = db;
11
12 db.connect((err) => {
13   if (err) throw err;
14
15   db.query(
16     `CREATE TABLE IF NOT EXISTS images
17     (
18       id      INT(11)      UNSIGNED NOT NULL AUTO_INCREMENT,
19       date_created TIMESTAMP    NOT NULL DEFAULT CURRENT_TIMESTAMP,
20       date_used   TIMESTAMP    NULL DEFAULT NULL,
21       name        VARCHAR(300) NOT NULL,
22       size        INT(11)      UNSIGNED NOT NULL,
23       data        LONGBLOB     NOT NULL,
24
25       PRIMARY KEY (id),
26       UNIQUE KEY name (name)
27     )`
```

Then, create the Dockerfile for our service:



The screenshot shows a Google Cloud Shell interface. At the top, there are tabs for 'Kubernetes Engine - imagin' and 'Cloud Shell - imagin'. The URL in the address bar is <https://console.cloud.google.com/cloudshell/editor?project=imagin-205120>. The main area is titled 'Cloud Shell' and contains a code editor with three tabs: 'package.json', 'imagini.js', and 'Dockerfile'. The 'Dockerfile' tab is active and displays the following content:

```
1 FROM node
2 MAINTAINER Diogo Resende
3
4 ADD imagin.js /opt/app/imagini.js
5 ADD package.json /opt/app/package.json
6
7 WORKDIR /opt/app
8 RUN npm i
9
10 EXPOSE 3000
11
12 CMD [ "node", "/opt/app/imagini" ]
13
14 |
```

Below the code editor, there's a terminal window titled 'imagin-205120'. It shows the following output:

```
Your Cloud Platform project in this session is set to imagin-205120.
Use "gcloud config set project" to change to a different project.
diogorr@imagin-205120:~$ ls -l
total 12
-rw-r--r-- 1 diogorr diogorr 3778 May 28 15:43 imagin.js
-rw-r--r-- 1 diogorr diogorr 319 May 28 15:40 package.json
lrwxrwxrwx 1 diogorr diogorr   38 May 24 21:41 README-cloudshell.txt -> /google/devshell/README-cloudshell.txt
-rw-r--r-- 1 diogorr diogorr  48 May 28 15:43 settings.json
diogorr@imagin-205120:~$
```

You can see on the bottom console that we're creating the files in the root folder. We're now going to use the console to create our container image. Type the following command:

```
docker build . -t gcr.io/imagin-205120/imagin
```

We're using a more composed name because we're publishing to the **Google Container Registry (GCR)**, and so it's good practice to use the project ID (as I did) and then the name of the service:

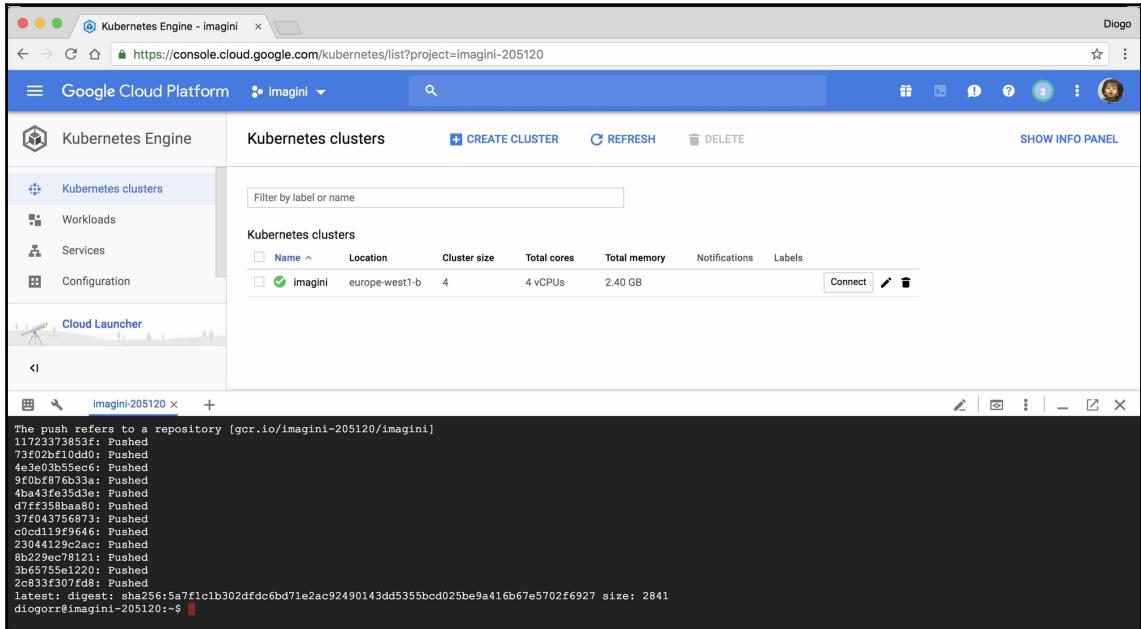
The screenshot shows a Cloud Shell interface with a file tree on the left containing a Dockerfile, imagini.js, and package.json. The terminal window below shows the output of a Docker build command:

```
--> d107338baca7
Step 8/9 : EXPOSE 3000
--> Using cache
--> c3370158c234
Step 9/9 : CMD node /opt/app/imagini
--> Using cache
--> 6aba2ec6ec31
Successfully built 6aba2ec6ec31
Successfully tagged gcr.io/imagini-205120/imagini:latest
diogorr@imagini-205120:~$
```

Then, let's push it to the registry using the following command:

```
gcloud docker -- push gcr.io/imagini-205120/imagini
```

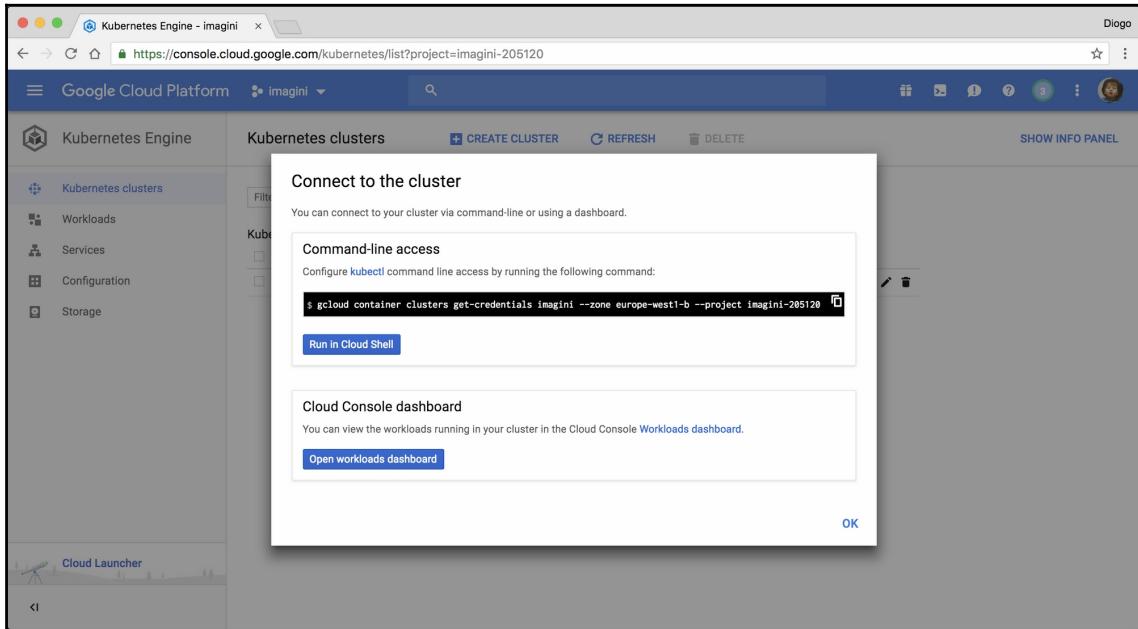
If everything goes right, you should see a list of layers of the image being pushed (published to the registry) and, in the end, a digest and the size of the image, as shown here:



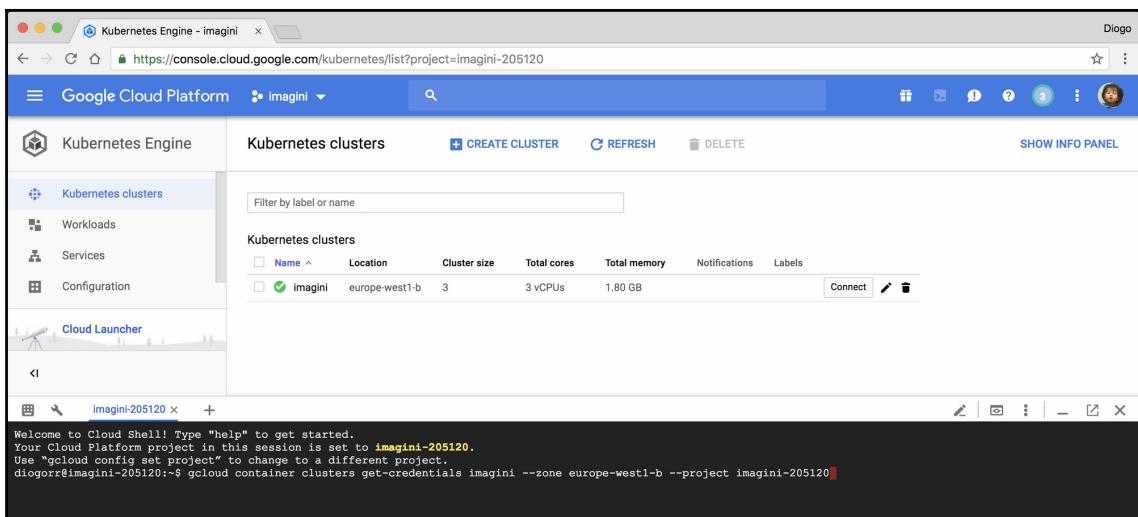
You may notice that I closed the editor between commands. You can use any console, and can even have more than one open.

We now need to create the credentials. We're not going to have passwords on text files. We're going to use an SQL proxy, and we'll expose credentials when the containers are executed, using an environment variable.

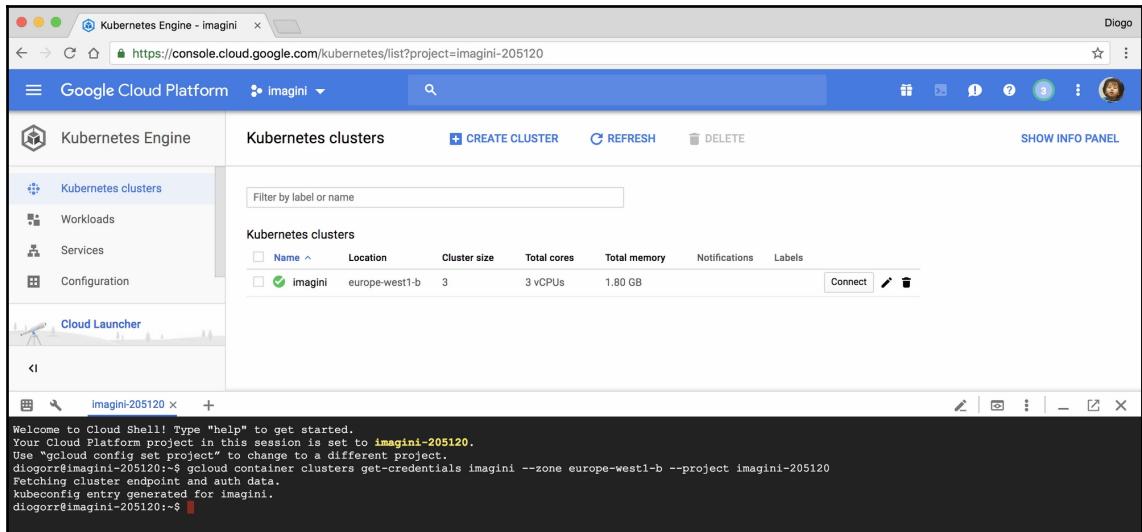
First, close the console and click the **Connect** button that is located on the right-hand side of the cluster name:



Click **Run in Cloud Shell**. This will reopen the console, but will type in the command to connect to our cluster:



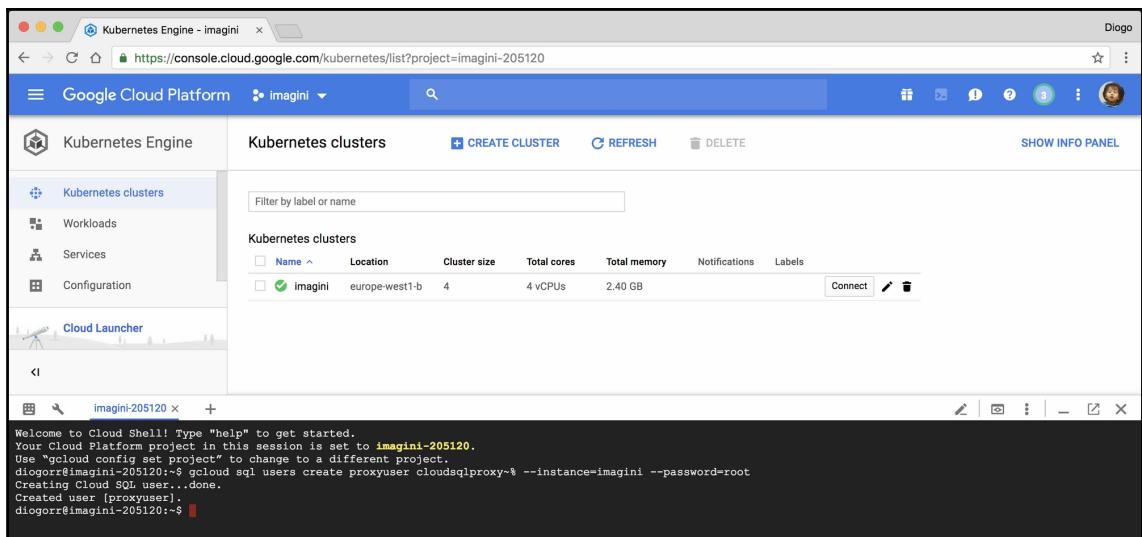
Just hit the *Enter* key in order to have access to the cluster:



The screenshot shows the Google Cloud Platform interface for the Kubernetes Engine. On the left, there's a sidebar with 'Kubernetes Engine' selected. The main area is titled 'Kubernetes clusters' and shows one cluster named 'imagin1'. Below the cluster table is a terminal window titled 'imagin1-205120'. The terminal output is as follows:

```
Welcome to Cloud Shell! Type "help" to get started.  
Your Cloud Platform project in this session is set to imagin1-205120.  
Use "gcloud config set project" to change to a different project.  
diogor@imagin1-205120:~$ gcloud container clusters get-credentials imagin1 --zone europe-west1-b --project imagin1-205120  
Fetching cluster endpoint and auth data.  
kubeconfig entry generated for imagin1.  
diogor@imagin1-205120:~$
```

Then, we'll use the `gcloud` command to create the proxy user account that will be used by the SQL proxy to access our Cloud SQL instance:



The screenshot shows the Google Cloud Platform interface for the Kubernetes Engine. On the left, there's a sidebar with 'Kubernetes Engine' selected. The main area is titled 'Kubernetes clusters' and shows one cluster named 'imagin1'. Below the cluster table is a terminal window titled 'imagin1-205120'. The terminal output is as follows:

```
Welcome to Cloud Shell! Type "help" to get started.  
Your Cloud Platform project in this session is set to imagin1-205120.  
Use "gcloud config set project" to change to a different project.  
diogor@imagin1-205120:~$ gcloud sql users create proxyuser cloudsqlproxy --instance=imagin1 --password=root  
Creating Cloud SQL user...done.  
Created user [proxyuser].  
diogor@imagin1-205120:~$
```

Now, type the following command:

```
gcloud sql instances describe imaginis
```

Scroll up the console log until you see a line that starts with **connectionName**:. Take a look at the following screenshot and note the highlighted text in the console.

The screenshot shows the Google Cloud Platform interface for the Kubernetes Engine. On the left, there's a sidebar with 'Kubernetes Engine' selected. Under 'Workloads', 'Services', and 'Configuration' are listed. Below that is the 'Cloud Launcher'. In the main area, 'Kubernetes clusters' are listed with a 'CREATE CLUSTER' button. One cluster named 'imaginis' is selected, showing its details: 'europe-west1-b', '4', '4 vCPUs', and '2.40 GB'. A 'Connect' button is also present. To the right of the cluster list is a terminal window titled 'imaginis-205120'. It displays the output of the 'gcloud sql instances describe' command, specifically highlighting the 'connectionName' line which reads 'connectionName: imaginis-205120:europe-west1:imaginis'.

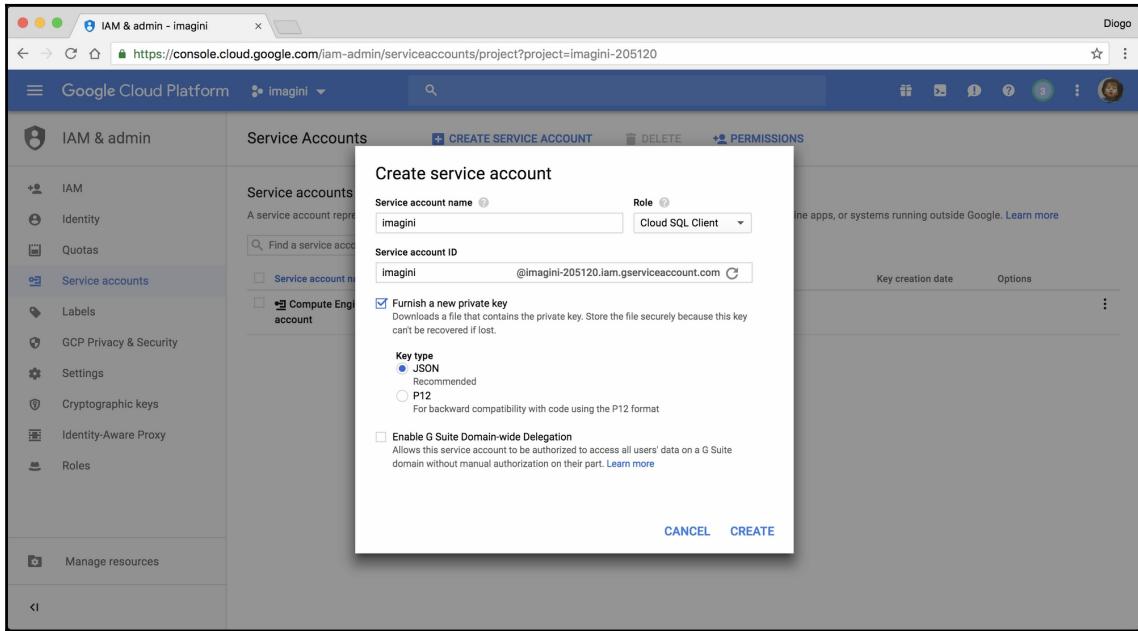
```
connectionName: imaginis-205120:europe-west1:imaginis
databaseVersion: MYSQL_5_7
etag: "A8B6a92871W_rBCwJZgLR2RH4M/MTA"
gceZone: europe-west1-b
instanceType: CLOUD_SQL_INSTANCE
ipAddresses:
- ip: 35.195.176.197
  type: PRIMARY
kind: sqlInstance
name: imaginis
project: imaginis-205120
region: europe-west1
selfLink: https://www.googleapis.com/sql/v1beta4/projects/imaginis-205120/instances/imaginis
serverCaCert:
  cert: |-
```

Copy the name down and save it for later. We'll need it when we create our deployment configuration.

Now, let's create a service account that is able to access our Cloud SQL instance. Head to **IAM & admin** and click **Service accounts**:

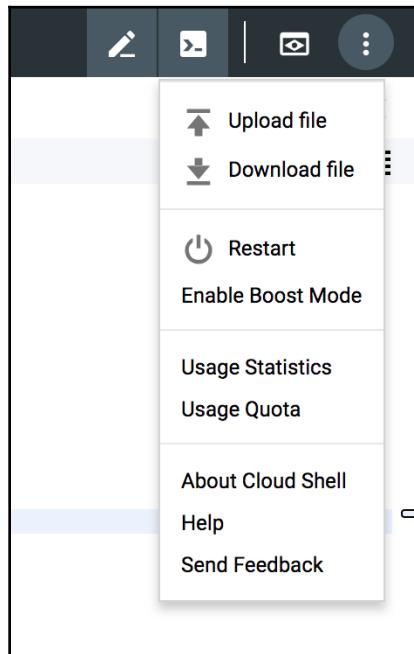
The screenshot shows the Google Cloud Platform dashboard for a project named 'imagin1'. The left sidebar has 'IAM & admin' selected, which is currently expanded. Under 'Service accounts', the 'Create new service account' button is visible. The main area displays 'Project info' (Project name: imagin1, Project ID: imagin1-205120), a 'Compute Engine' CPU usage chart, 'Google Cloud Platform status' (All services normal), 'Billing' information (Estimated charges: EUR €0.00), and 'Error Reporting' (Top errors in last 24 hours).

Create a service account with the **Cloud SQL Client** role, and check **Furnish a new private key**:

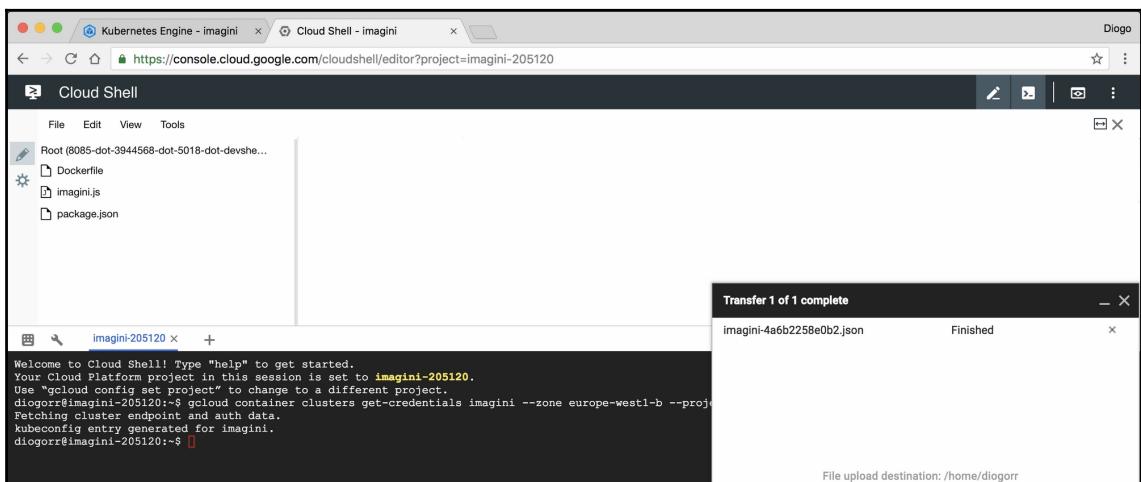


After hitting **Create**, the account will be created and a JSON file will be given to you to save in your local disk. Head back to the cluster, open the console, and then click to open the editor again.

You'll notice that there's a menu with several options in the top-right corner, one of which is **Upload file**:



Pick the JSON file you just saved and upload it using that option:



Now, let's use the console to create the instance secret using that credentials file. Use the following command (change the name to your JSON file):

```
kubectl create secret generic cloudsqli-instance-credentials --from-file=credentials.json=imagini-4a6b2258e0b2.json
```

The following screenshot shows the result of the preceding command executed in the console. You should see something like **secret "cloudsql-instance-credentials" created**.

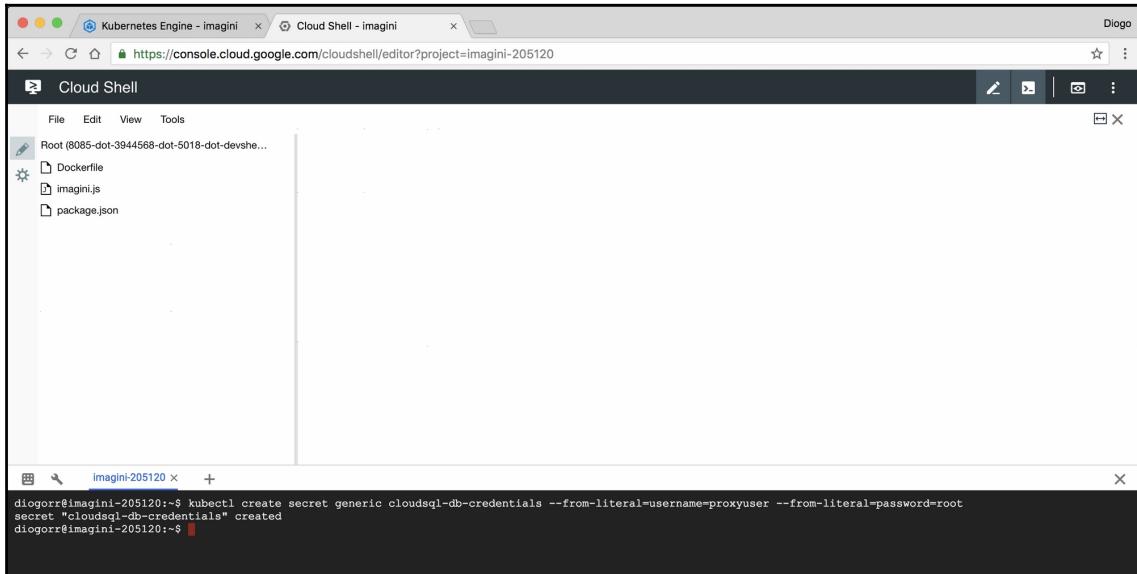
The screenshot shows a Cloud Shell window within a browser tab titled 'Cloud Shell - imagini'. The URL is <https://console.cloud.google.com/cloudshell/editor?project=imagini-205120>. The Cloud Shell interface has a file tree on the left containing 'Dockerfile', 'imagini.js', and 'package.json'. The main terminal window displays the following command and its output:

```
Welcome to Cloud Shell! Type "help" to get started.  
Your Cloud Platform project in this session is set to imagini-205120.  
Use "gcloud config set project" to change to a different project.  
diogorr@imagini-205120:~$ gcloud container clusters get-credentials imagini --zone europe-west1-b --project imagini-205120  
Fetching cluster endpoint and auth data.  
kubeconfig entry generated for imagini.  
diogorr@imagini-205120:~$ kubectl create secret generic cloudsqli-instance-credentials \  
>   --from-file=credentials.json=imagini-4a6b2258e0b2.json  
secret "cloudsql-instance-credentials" created  
diogorr@imagini-205120:~$
```

Now, let's create the database credentials secret:

```
kubectl create secret generic cloudsqli-db-credentials --from-literal=username=proxyuser --from-literal=password=root
```

The following screenshot shows the result of the preceding command executed in the console. You should see something like **secret "cloudsql-db-credentials" created.**



A screenshot of a Cloud Shell window titled "Cloud Shell". The window has a dark header bar with the title and a light gray body. In the top left of the body, there's a sidebar with a "Root" icon and the path "Root (8085-dot-3944568-dot-5018-dot-devshe...)" followed by a tree view with "Dockerfile", "imagini.js", and "package.json". The main area is a terminal window with a black background and white text. It shows the command "kubectl create secret generic cloudsqldb-credentials --from-literal=username=proxyuser --from-literal=password=root" being run, followed by the output "secret \"cloudsqldb-credentials\" created". The bottom right corner of the terminal window has a red border.

We now have everything ready to deploy our service. This is the configuration we'll be using:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: imagini-frontend
  labels:
    app: imagini
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: imagini
        tier: frontend
    spec:
      containers:
        - name: imagini-app
          image: gcr.io/imagini-205120/imagini
          imagePullPolicy: Always
          ports:
            - name: http-server
```