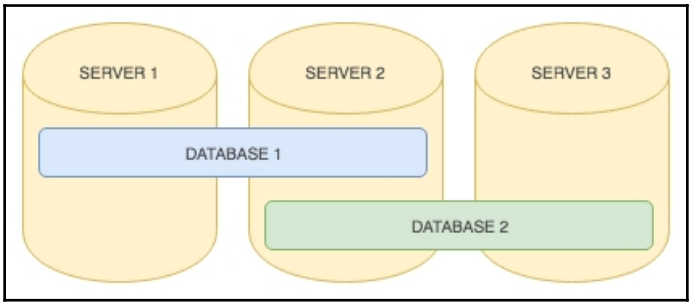
Sharding approach

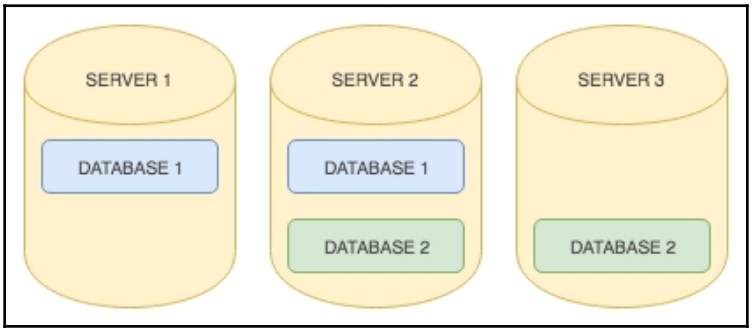
Sharding is the process of fragmenting the data across different locations. This enables us to split a large dataset into different servers, each one with a smaller set. This is comparable to a RAID 0 configuration, where, for example, you may need a 10 TB disk and do this by combining two 5 TB disks:



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

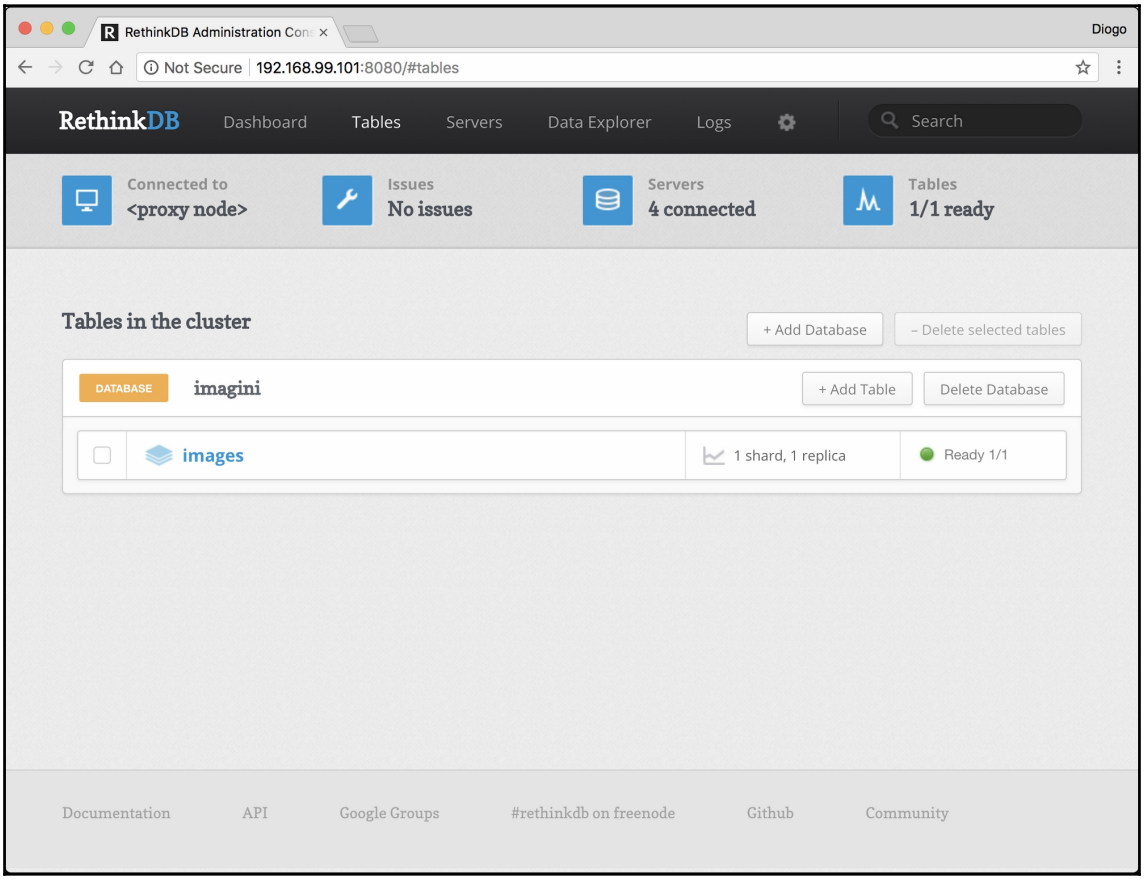
Replicating is the process of having copies of the data across different locations. This enables us to have a bigger throughput as we can have different services serving the same data. It also brings complexity, at least for the database servers, as they need to keep everything synchronized:



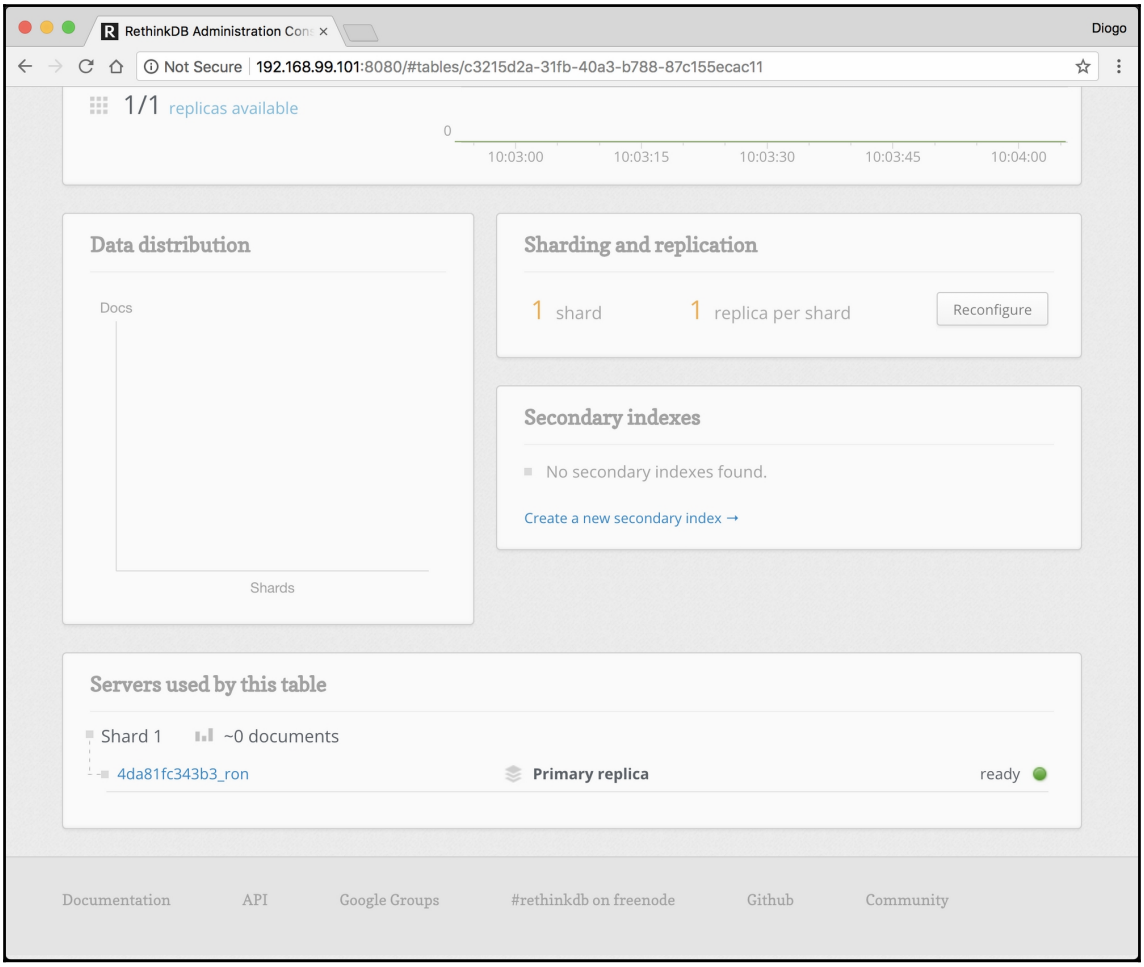
Doing replicas also gives you redundancy in case of any failures in regards to parts of the server nodes, depending on your configuration. If you use a database cluster for more than one application, you don't need to have replicas of a database in all cluster nodes; it all depends on your needs.

Sharding and replicating

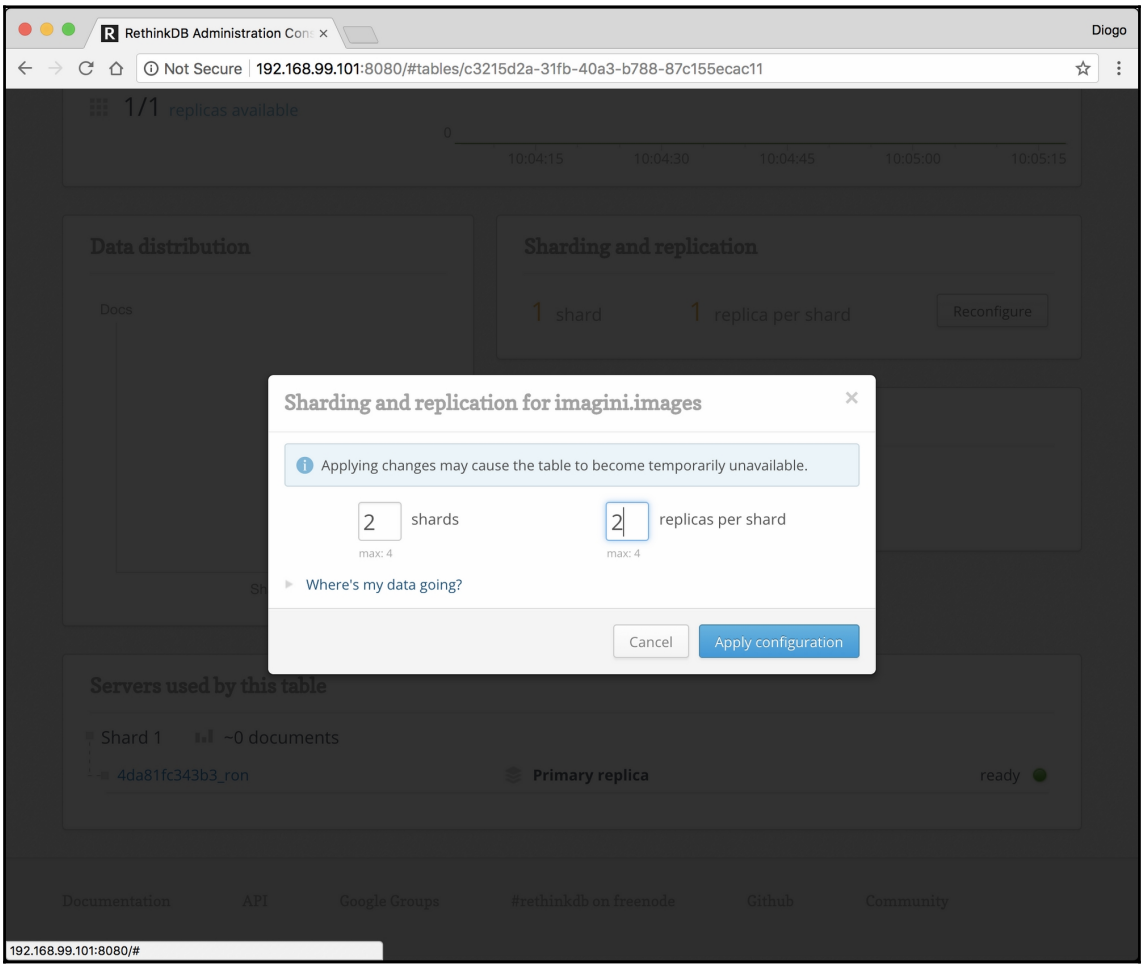
Looking at our service, now that it's using RethinkDB, we can have the best of both worlds and enable sharding and replication at the same time. Head to the Administration Console and click on the **Tables** tab at the top:



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

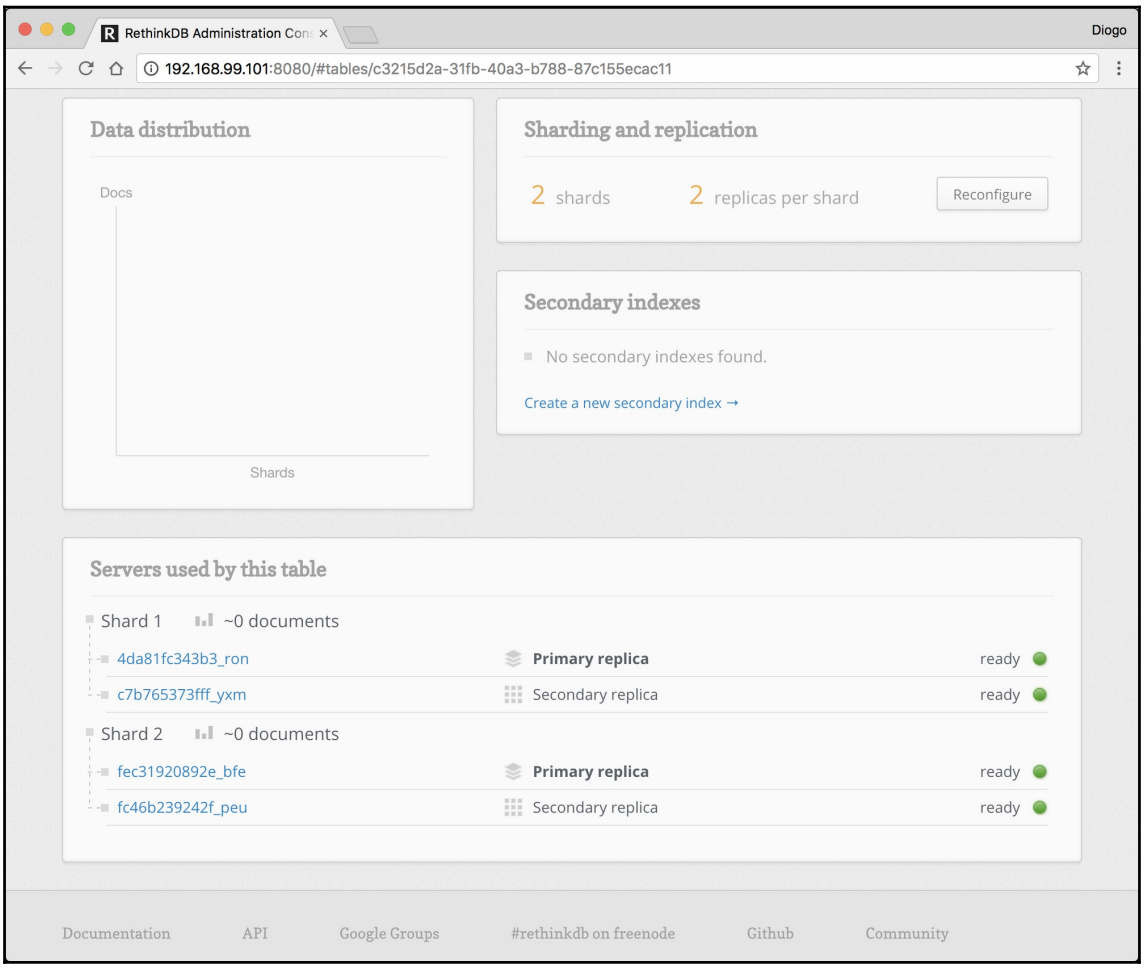


You can see that our table is on one shard (the default), with only one replica per shard. Down the bottom, you can see the server (blue link) on our cluster that is holding the table. Click on **Apply configuration**:



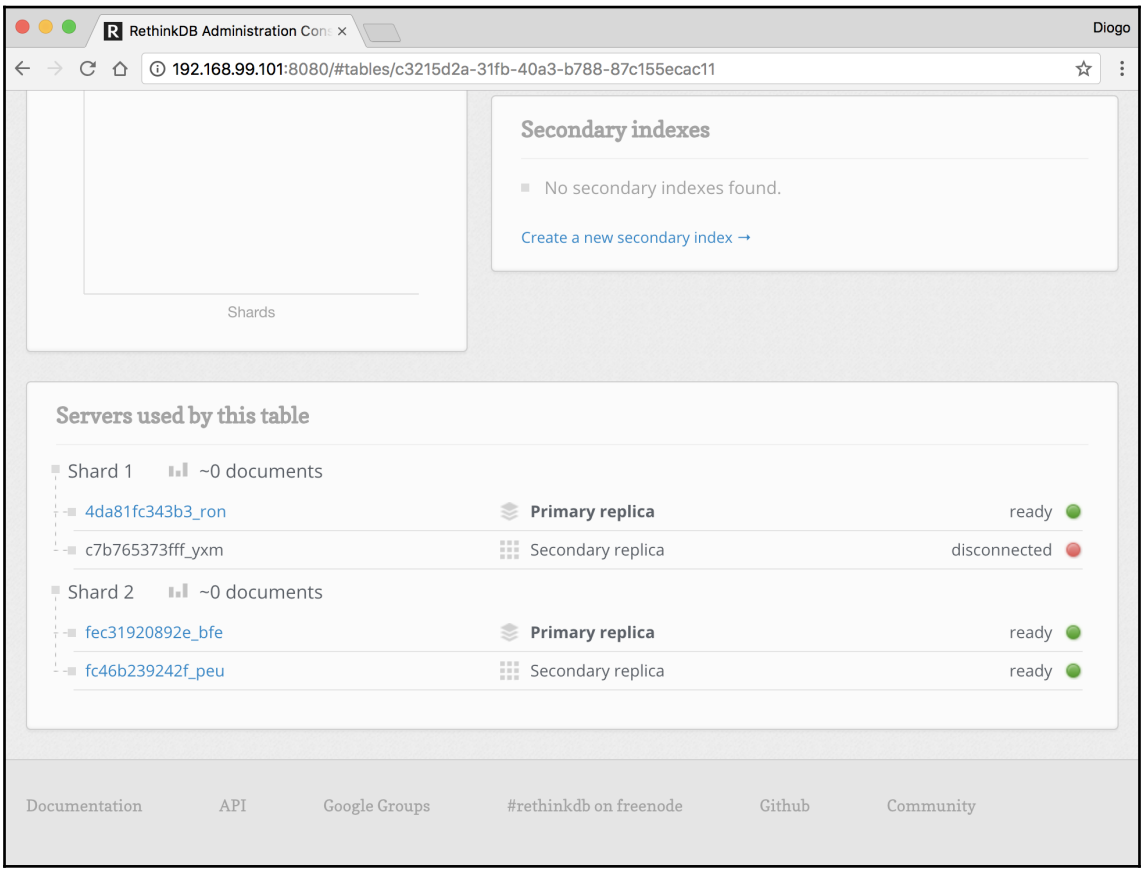
You can configure the number of shards and the number of replicas of every shard. The limitation on four shards and four replicas is because our cluster has only four nodes. Any number above that would not make sense.

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:



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 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.

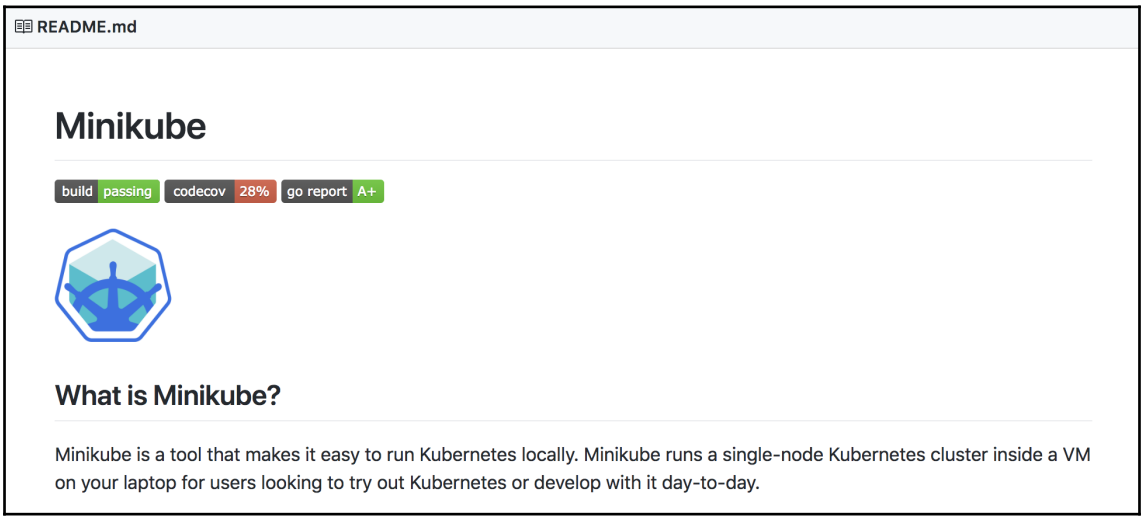
Let's see what changes we need to make to our microservice in order to successfully run it using Kubernetes. But first, we need to clarify some of the concepts that are used in Kubernetes:

- **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:


README.md

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.

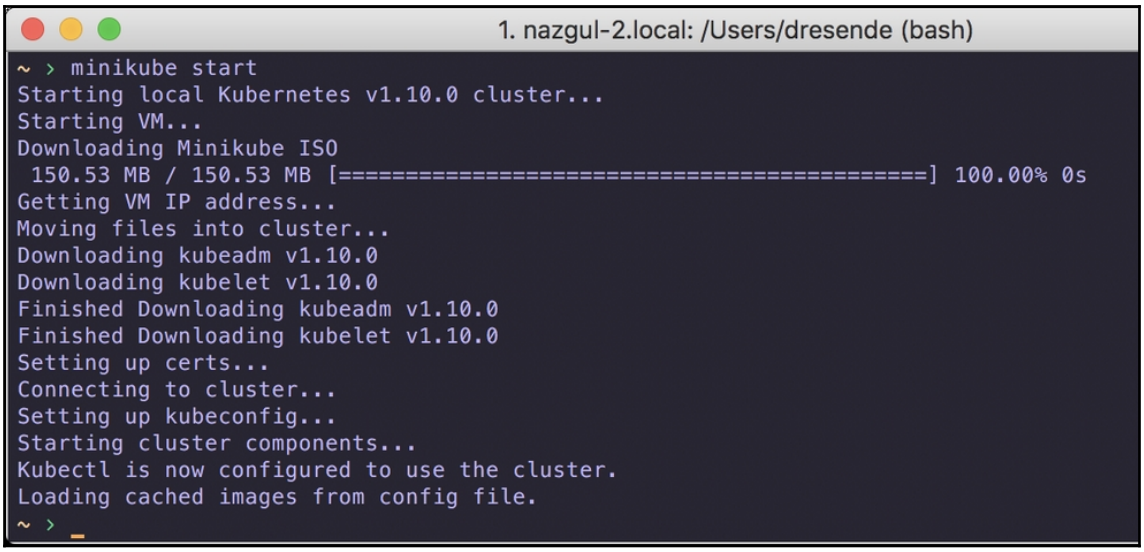
Follow the installation instructions. If you're on macOS and have `homebrew` installed, you just need to run:

```
brew cask install minikube
```

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

```
minikube start
```

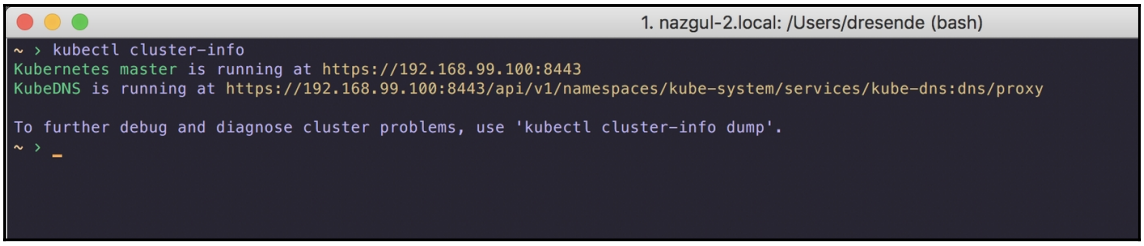
You have to wait a minute or two before everything is set up:



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

```
kubectl cluster-info
```

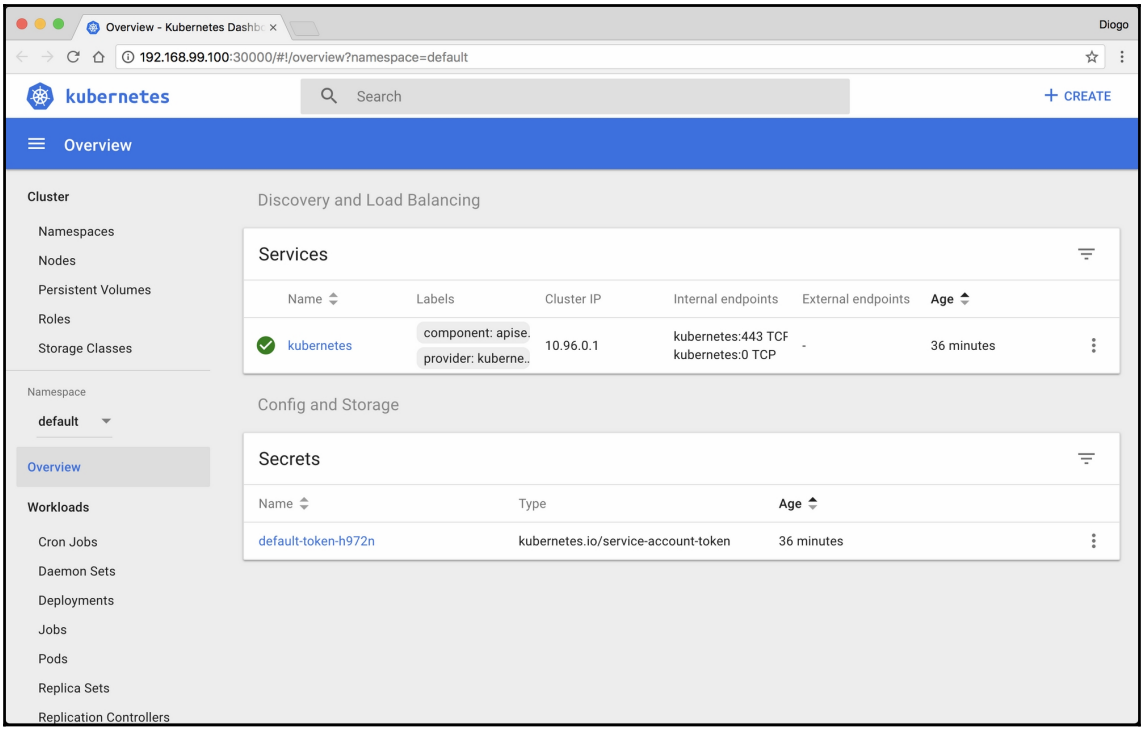
This is a cluster of nodes that run containers, similar to Docker swarm. You should get a positive response as follows:



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

```
minikube dashboard
```

Your browser should open something along the following lines:



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.

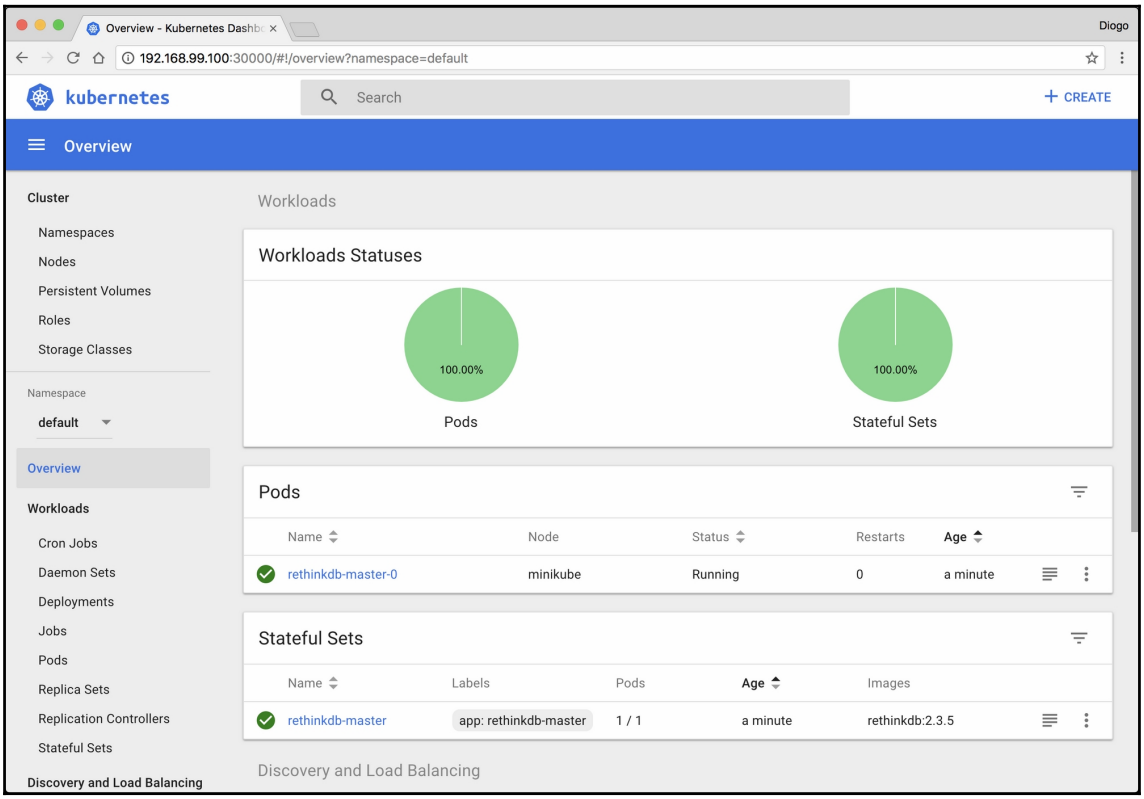
To begin, hit the **Create** button in the top right-hand corner of the page. On the text input that shows appears, write the following YAML configuration:

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

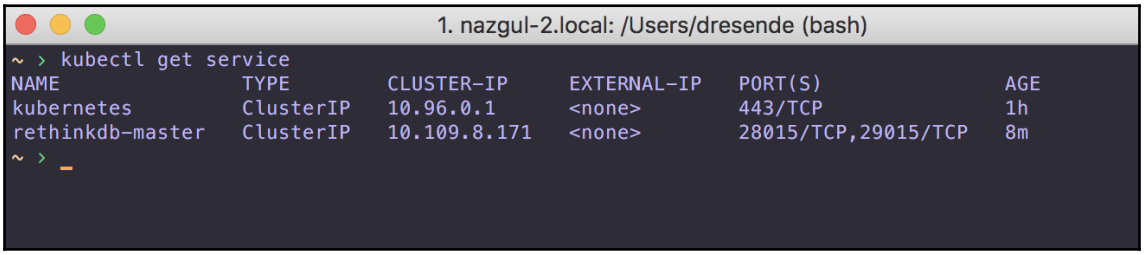
```
      hostname: rethinkdb-master
      containers:
      - name: rethinkdb
        image: rethinkdb:2.3.6
        command: ["rethinkdb"]
        args:
        - --bind
        - "all"
        - --canonical-address
        - "rethinkdb-master:29015"
        - --canonical-address
        - "$(MY_POD_IP):29015"
        volumeMounts:
        - name: rdb-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: rdb-local-data
        hostPath:
          path: /var/data/rethinkdb

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

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



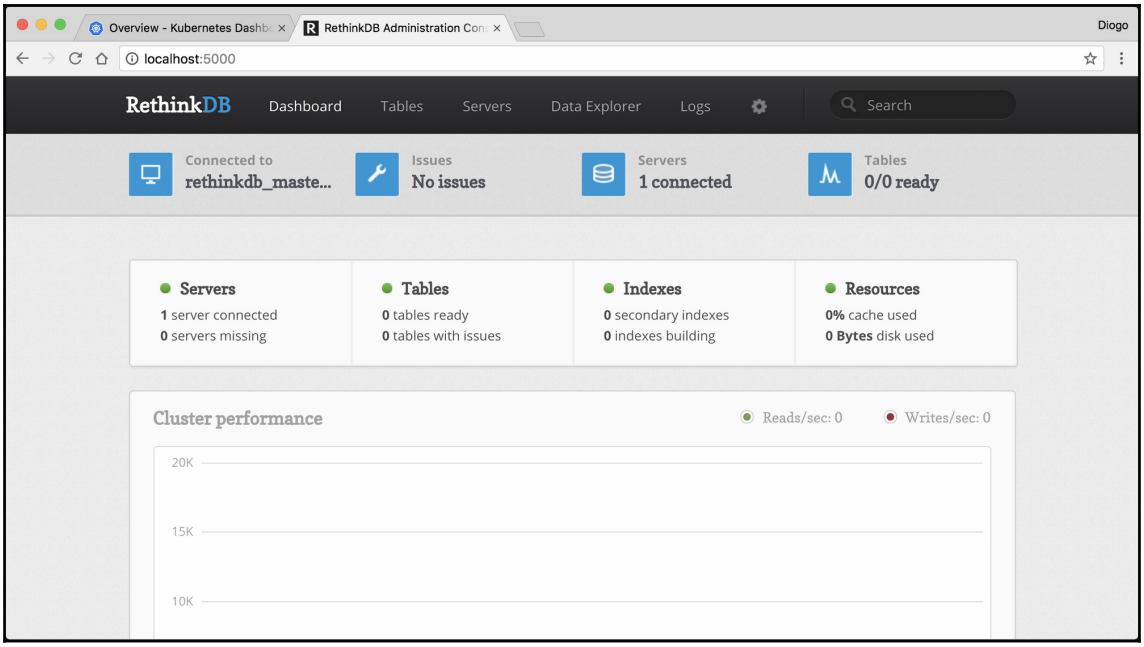
To check that our RethinkDB server is running correctly, we can add port forwarding from the outside of the cluster. First, let's list our services:



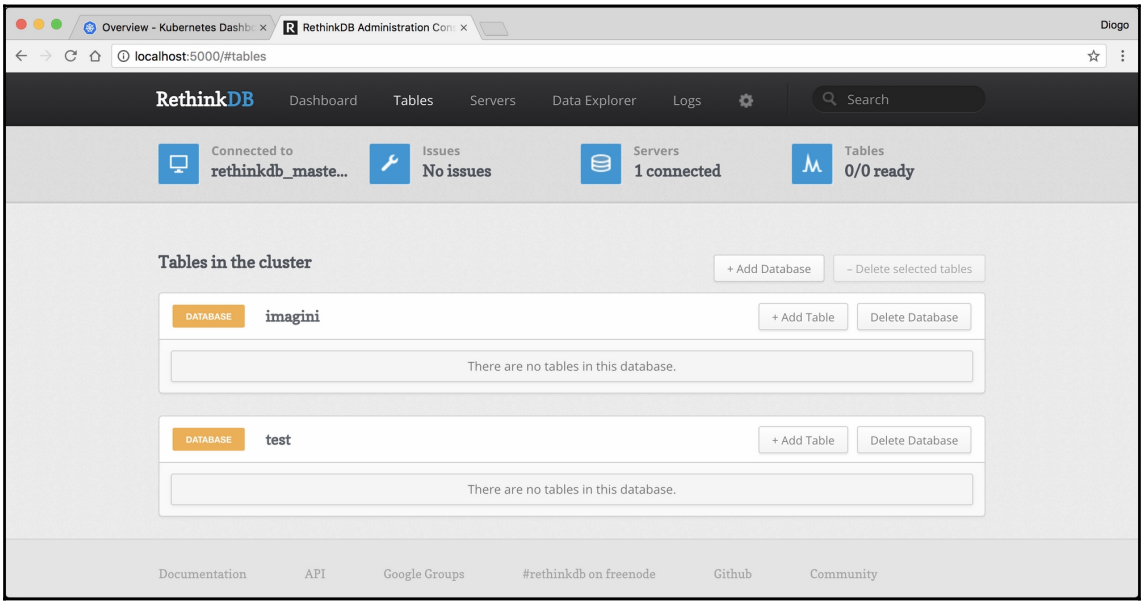
Our service is there. There are some endpoints, but the Administration Console isn't actually there. Let's forward that local port, which is 5000, to port `8080`:

```
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:



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:

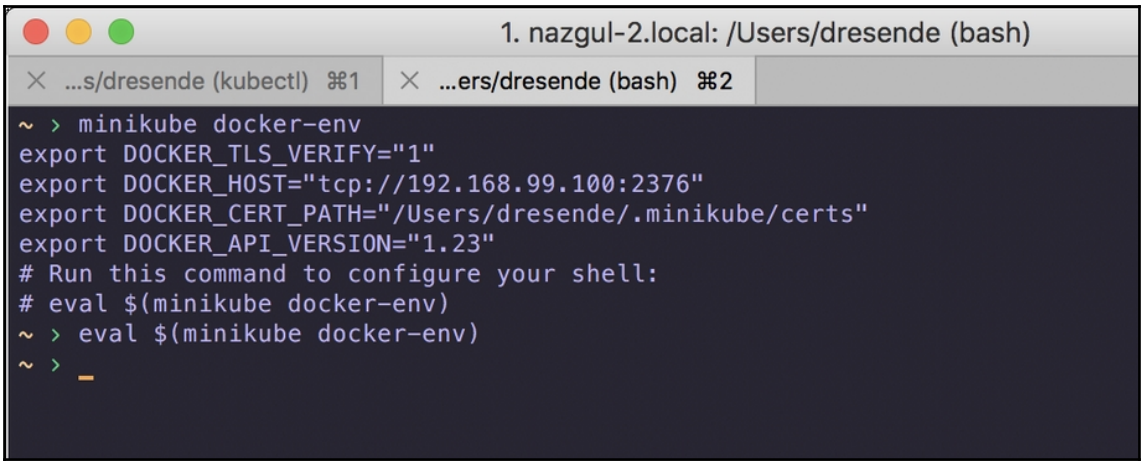


To keep it simple, we're keeping the RethinkDB connection hardcoded on our code. I'll leave it up to you to try it out later on with Kubernetes to see how you can have a persistent volume, have your settings there, and the database.

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:



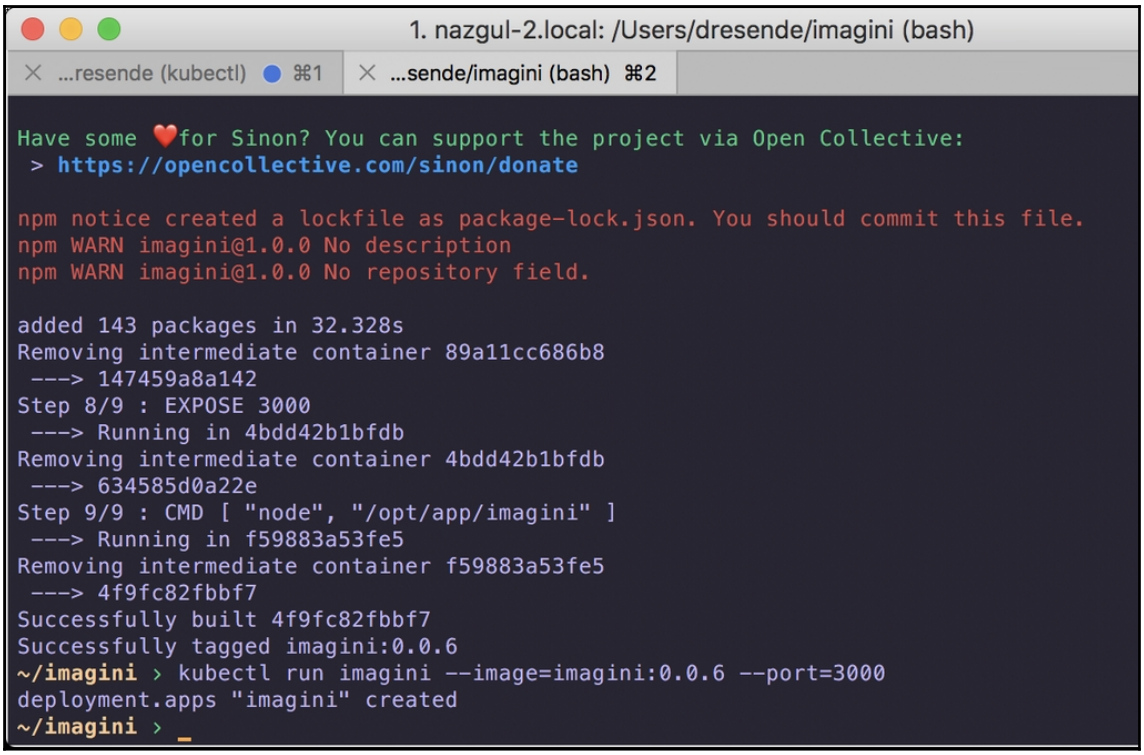
Then, let's build our image using the same Docker command:

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

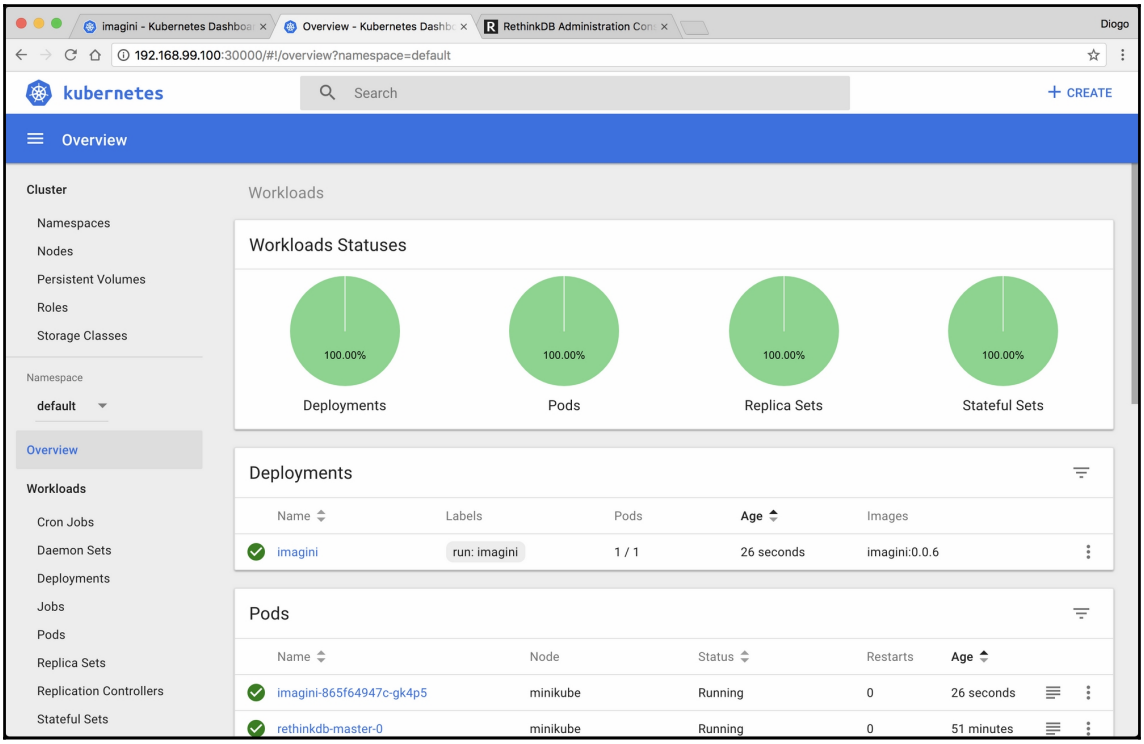
After building it, let's begin the deployment:

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

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



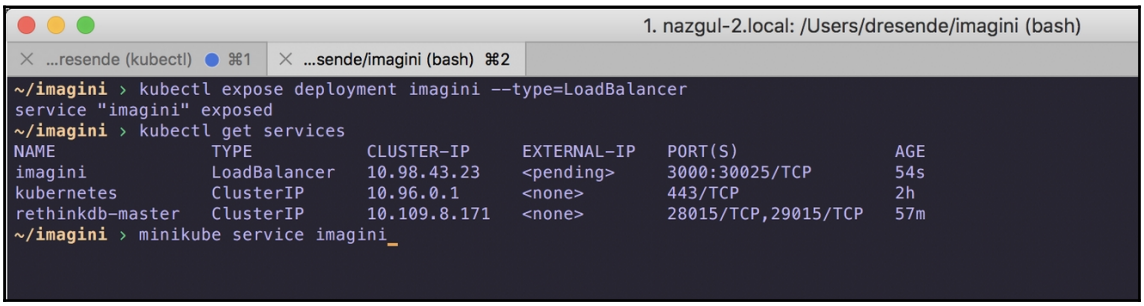
If you refresh the Administration Console, you now have **Deployments** and a new Pod:



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

```
kubectl expose deployment imagini --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:



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.

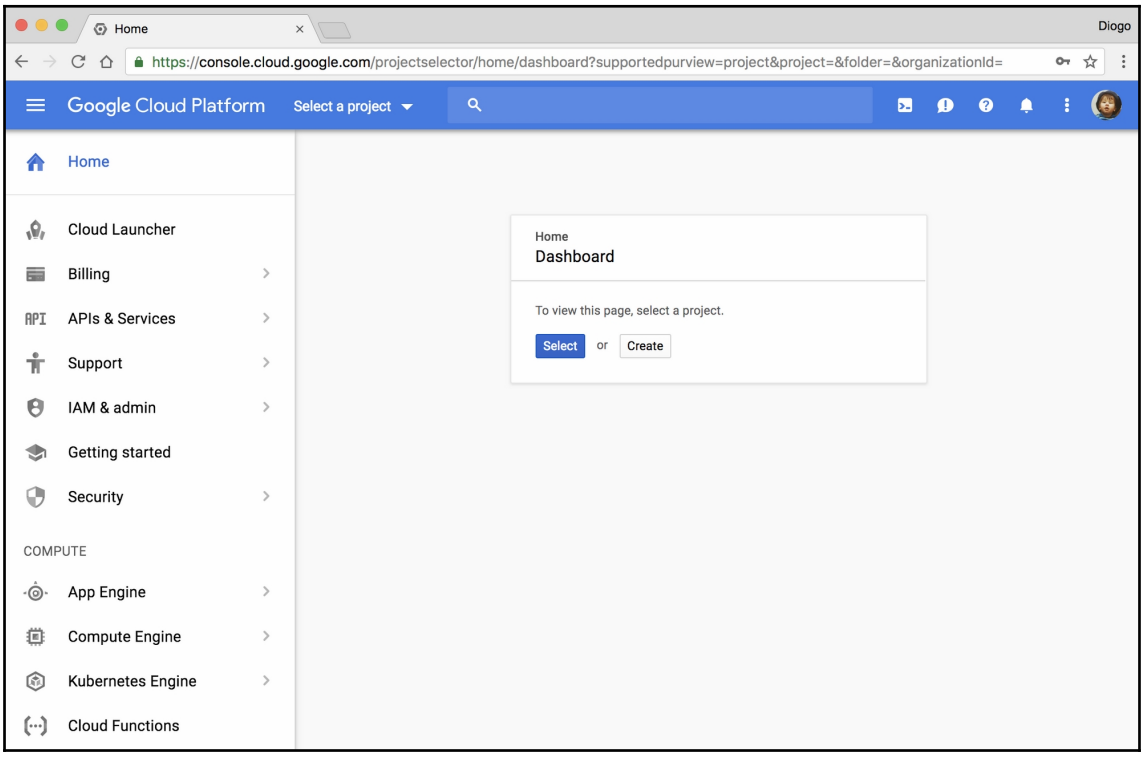
Thinking generically, there are a few points to take into account when designing our microservice, or anything that will be cloud-native, for that matter:

- 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

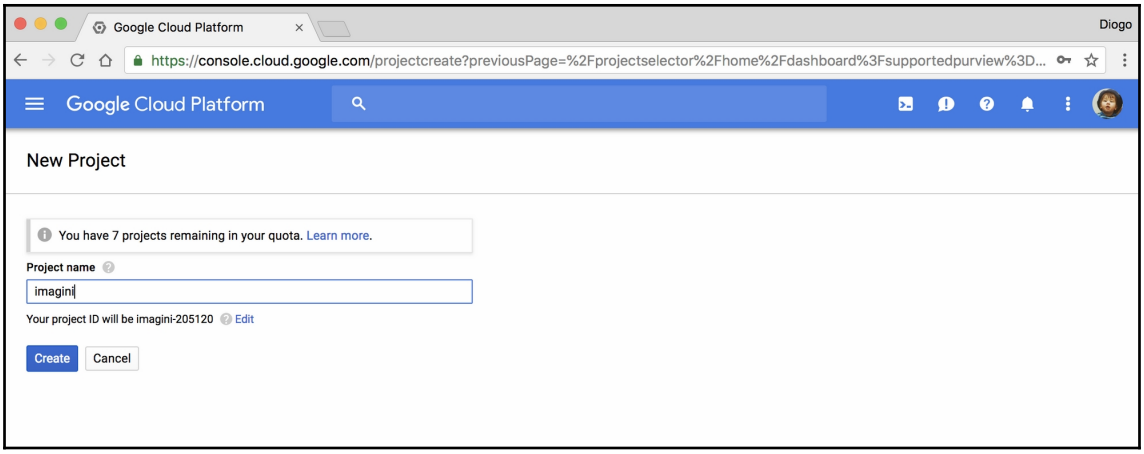
Now that we're able to deploy using containers, and we have used Kubernetes already on a local virtual machine, let's see how we can do the same on a cloud environment. You can choose any type of cloud provider, as long as you follow the considerations we just discussed in the previous section.

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:

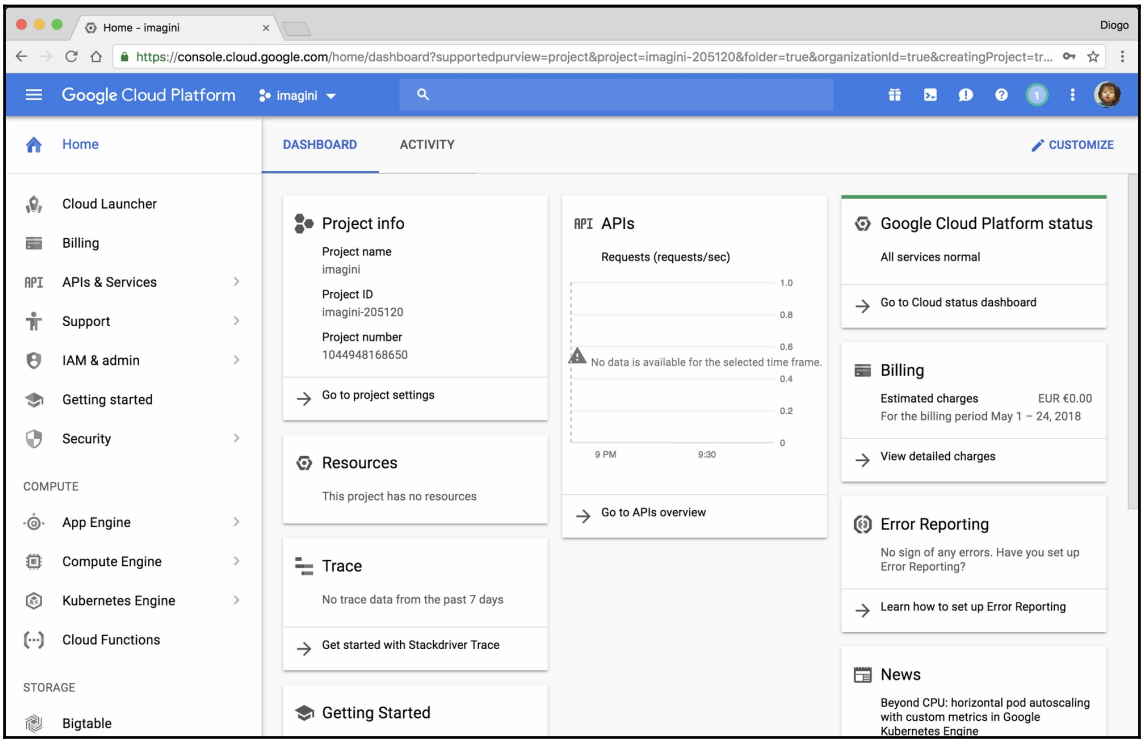


Creating a new project

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

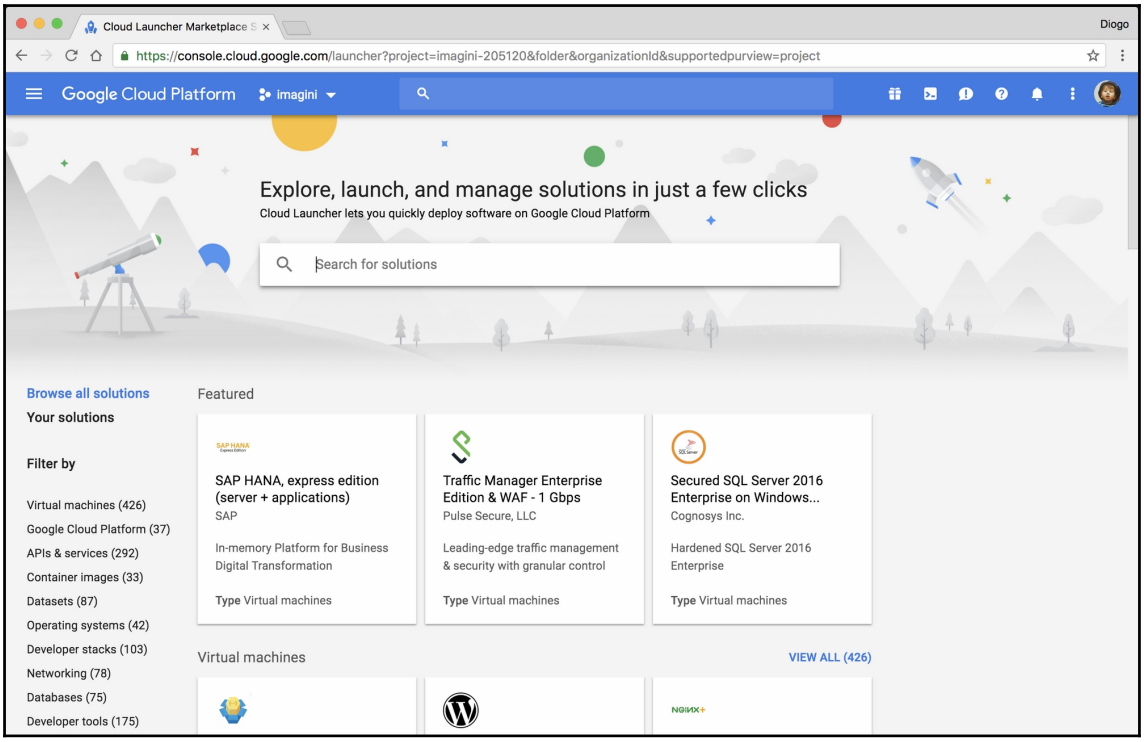


You'll then have access to the project dashboard. You can check resource usage and project status, as well as billing information:

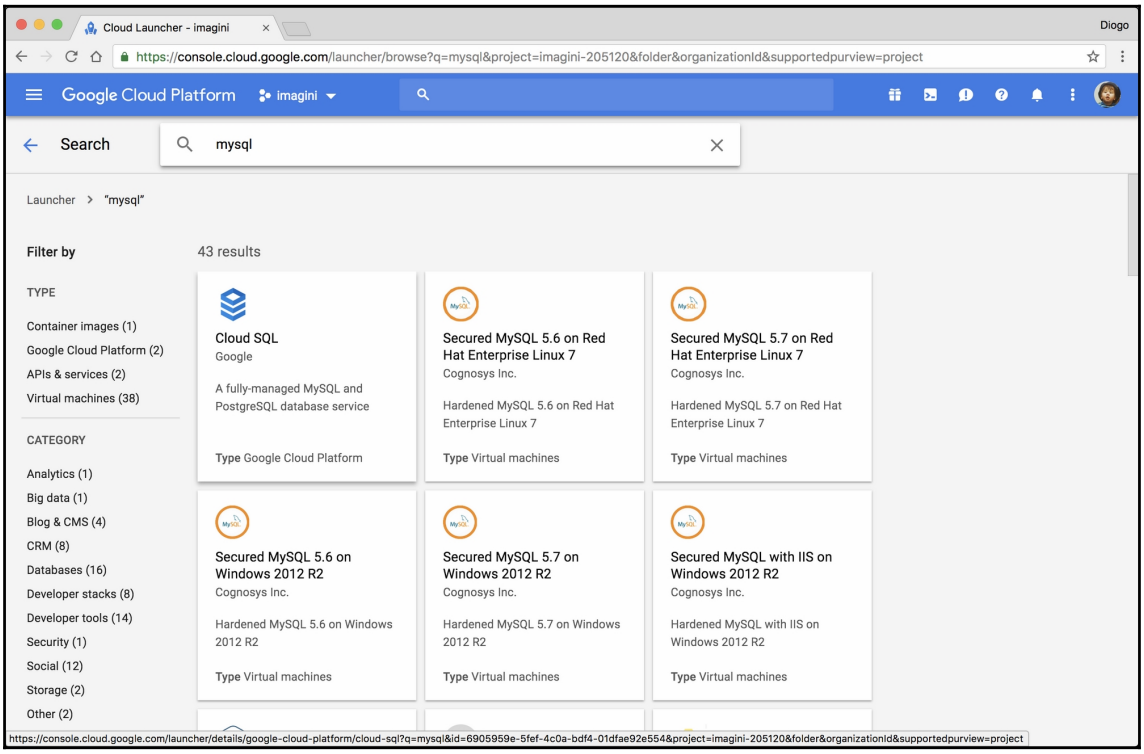


Deploying a database service

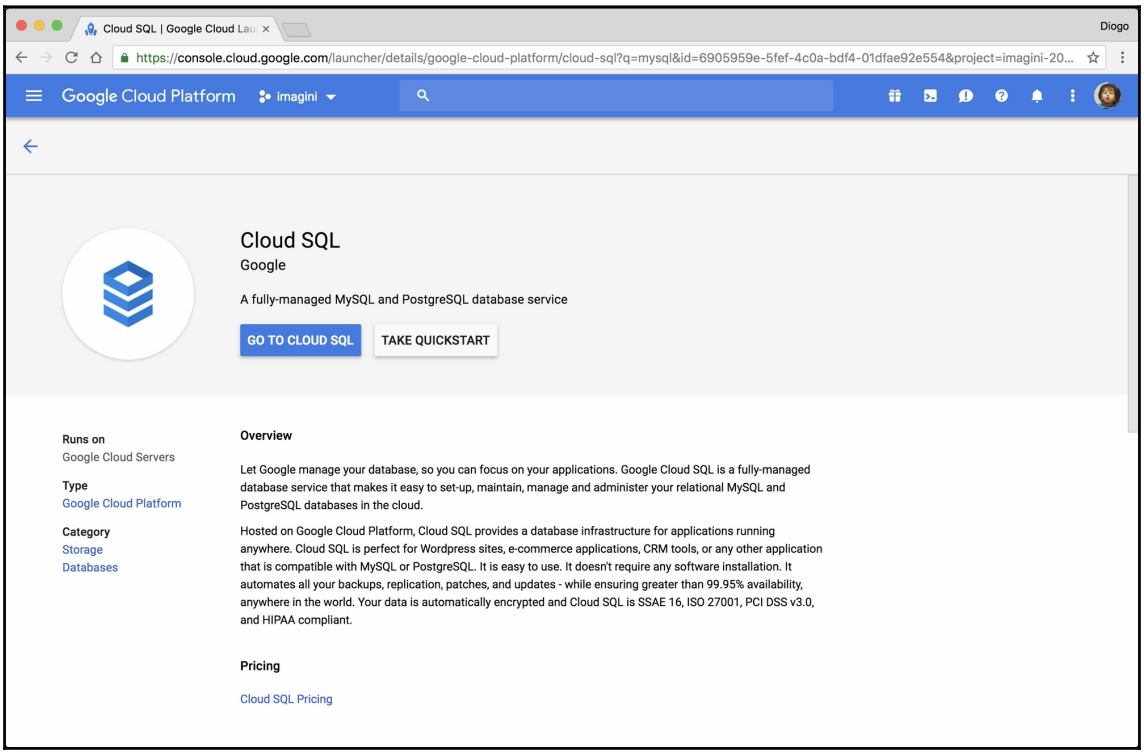
On the left-hand side, there are the main navigation options, one of which, Cloud Launcher, lets us easily set up services that we need. Since we used RethinkDB in the previous chapter, let's now try MySQL again. We need a database server, so let's head to Cloud Launcher:



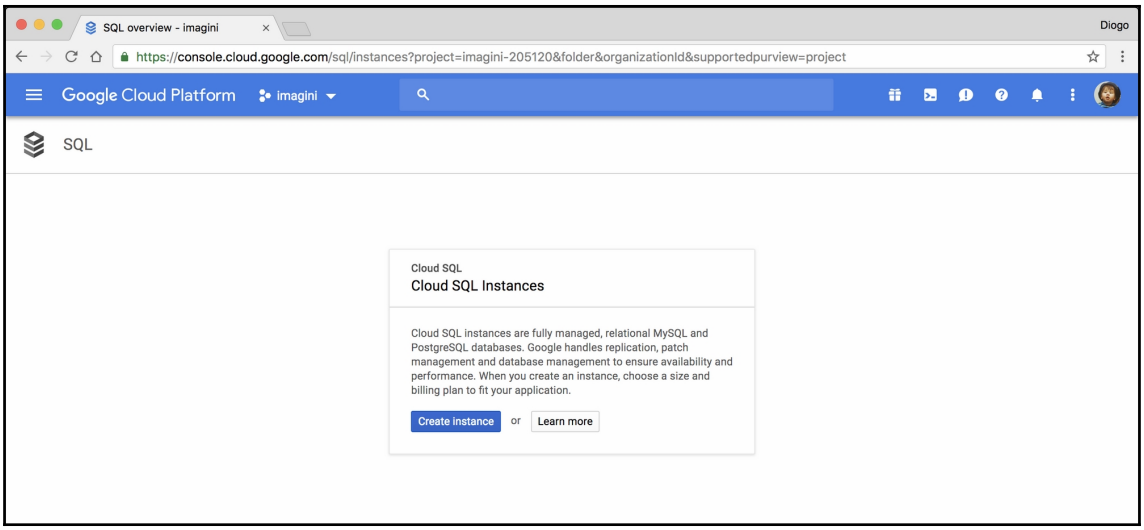
On the sidebar, there are a couple of service categories to choose from. Let's use the filter on top to search for **mysql**:



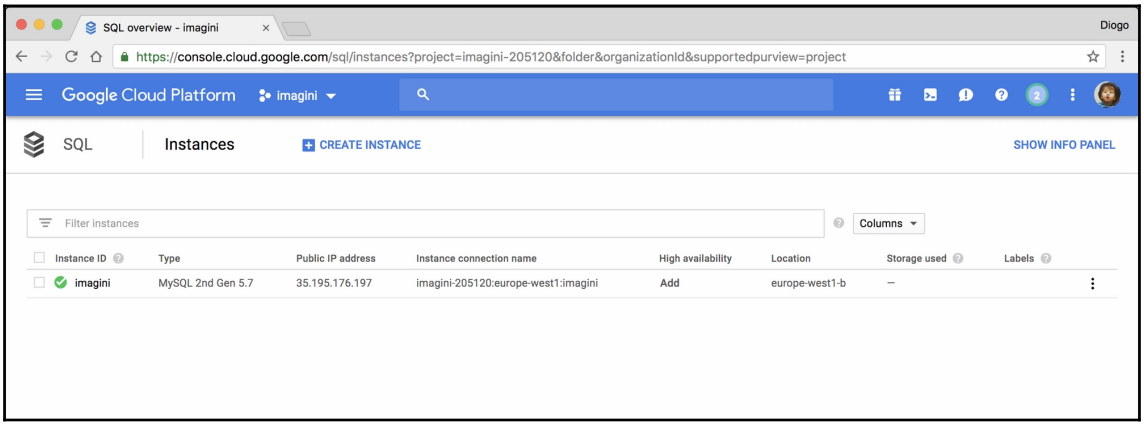
One of the options, in my case the first, is **Cloud SQL**, which is a Google service used to run a fully-managed MySQL or PostgreSQL database service. It alleviates the burden of configuration and maintenance. Click on that option:



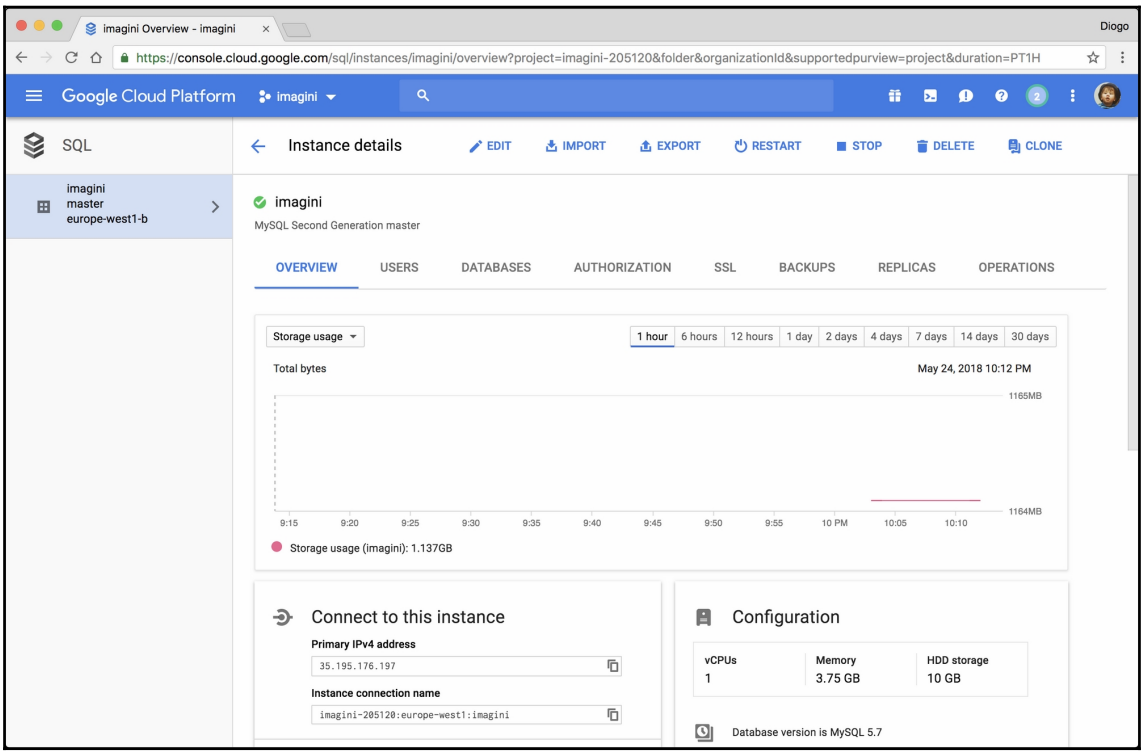
There's an overview with an explanation of the service, as well as pricing if you want to consult it. Because we're using free credit, that is not a concern for now. Click on **GO TO CLOUD SQL**:



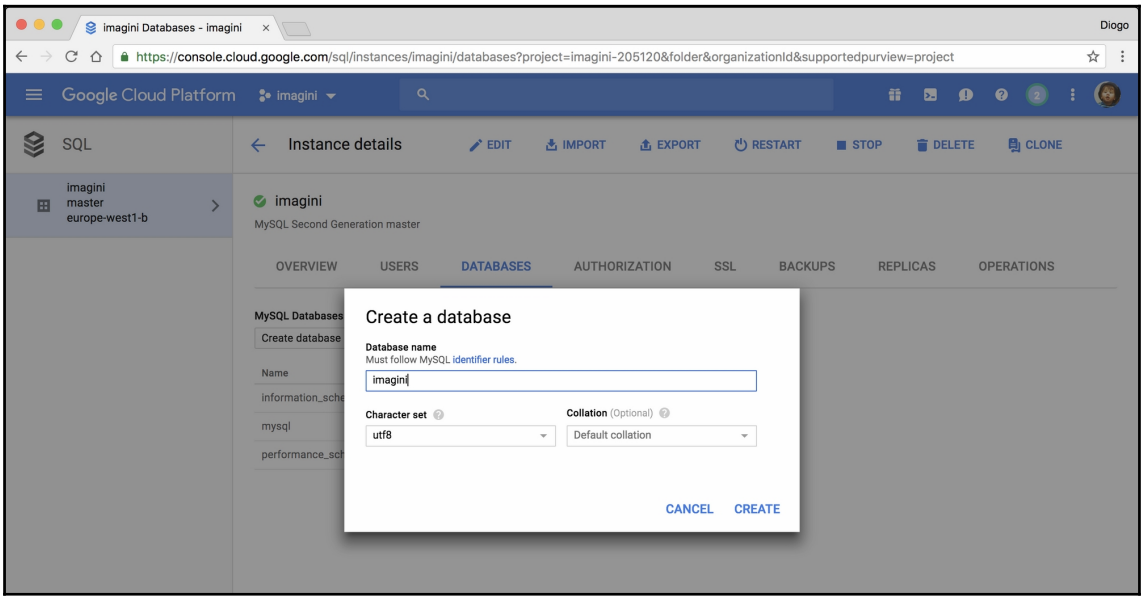
Because we're using the service for the first time, there's no service instance. Click on "Create Instance" to create a new one, follow the instructions, and create the instance. You'll be asked for a password for the default root account; don't forget it. At the end, you should see an instance list.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



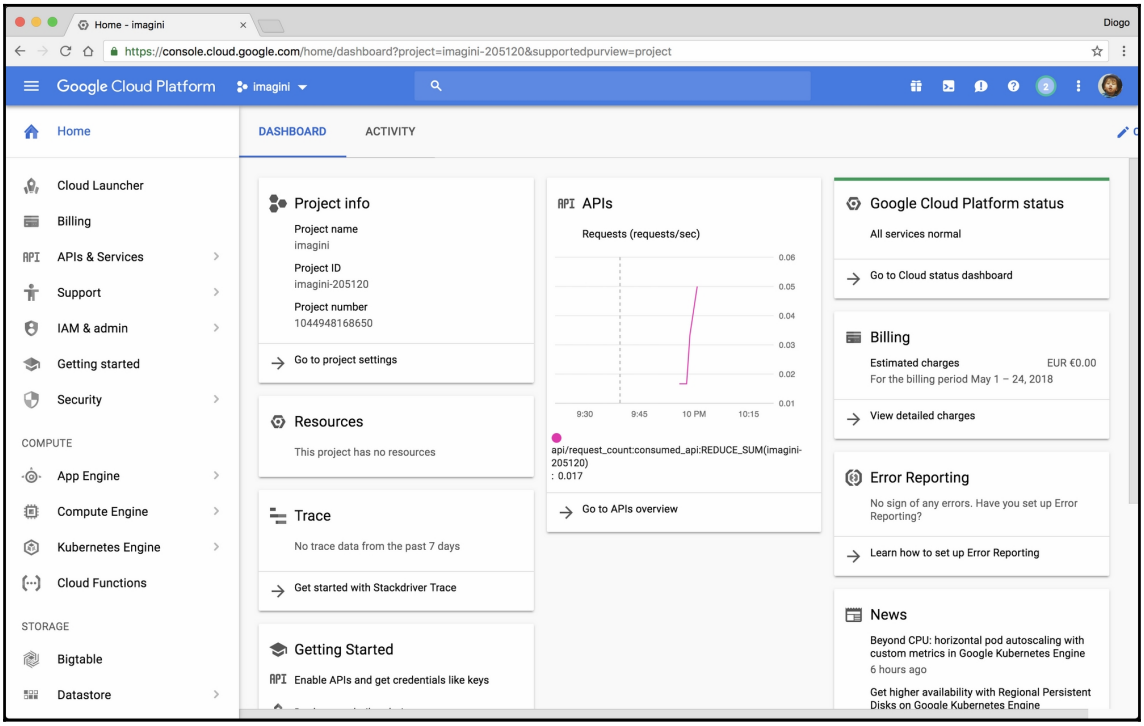
You can click on the instance ID and see more information about it. You'll have access to a service dashboard, as well as be able to import and export:



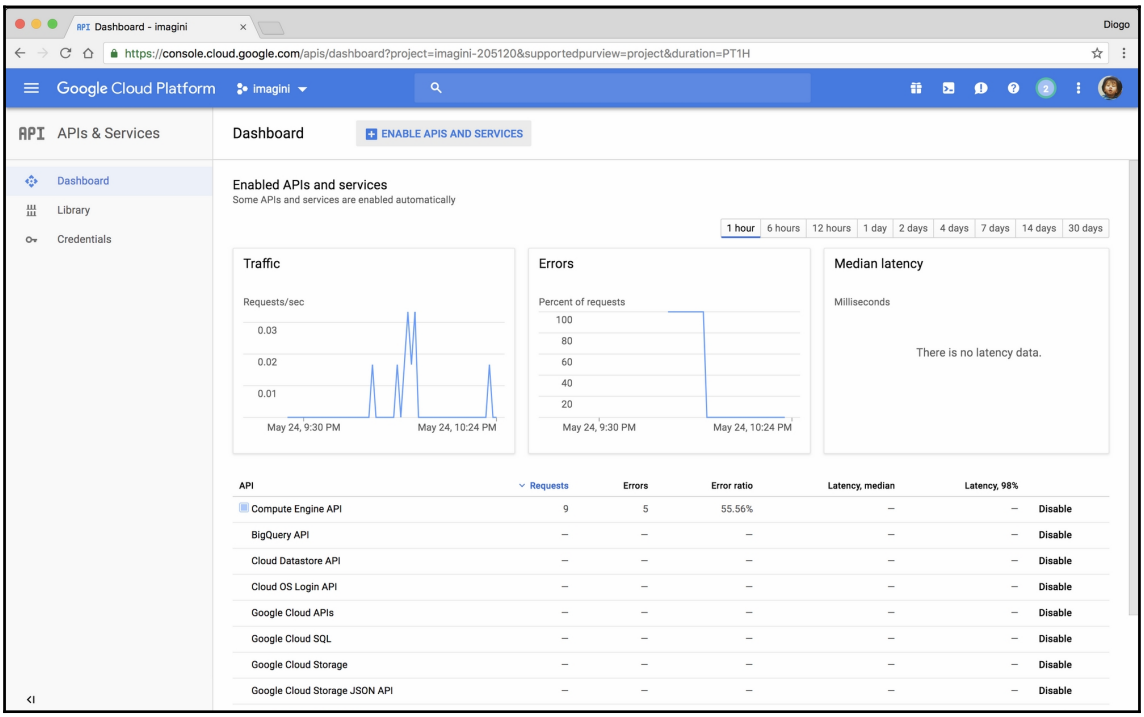
There's a **DATABASES** tab where we can create our microservice database. Head there and create it. Retain defaults on the character set and collation:



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:

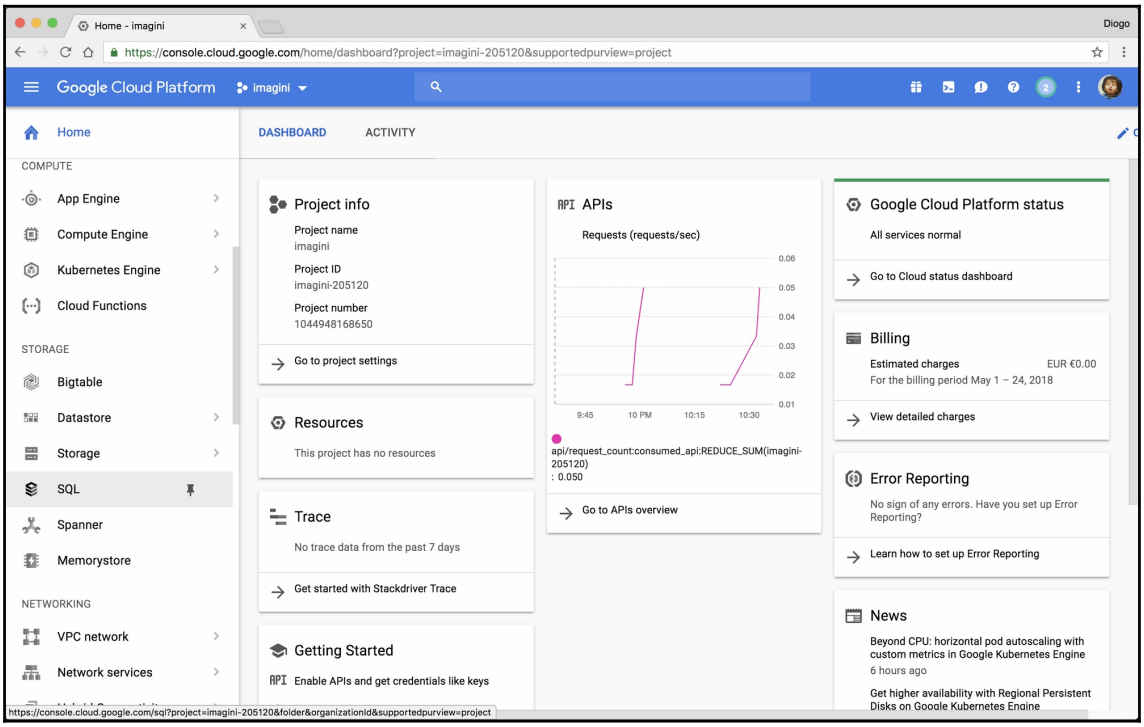


You'll notice that the **APIs** block on the middle column now has a line showing some activity. APIs are part of the infrastructure and, since we just created a service instance, there's some activity going on. Click on **Go to APIs overview**:



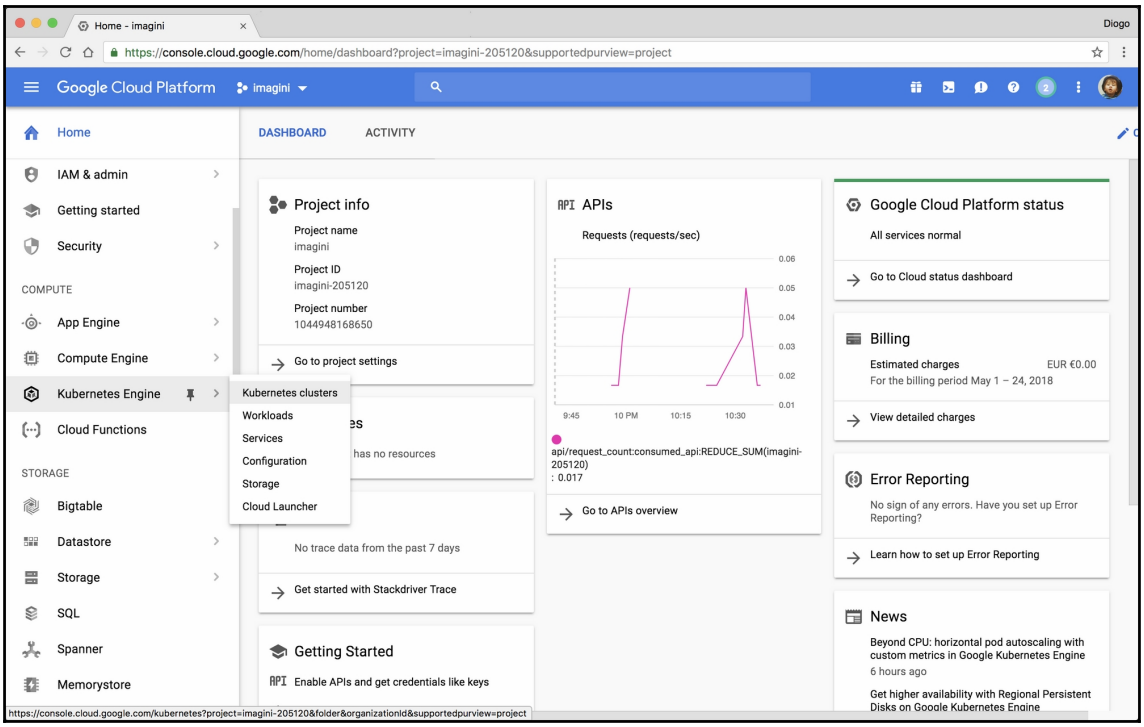
Here, you can monitor activity, traffic, and errors of all enabled and used APIs. Heading back to our project dashboard, and looking at the sidebar, you'll notice there are plenty of options available.

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:

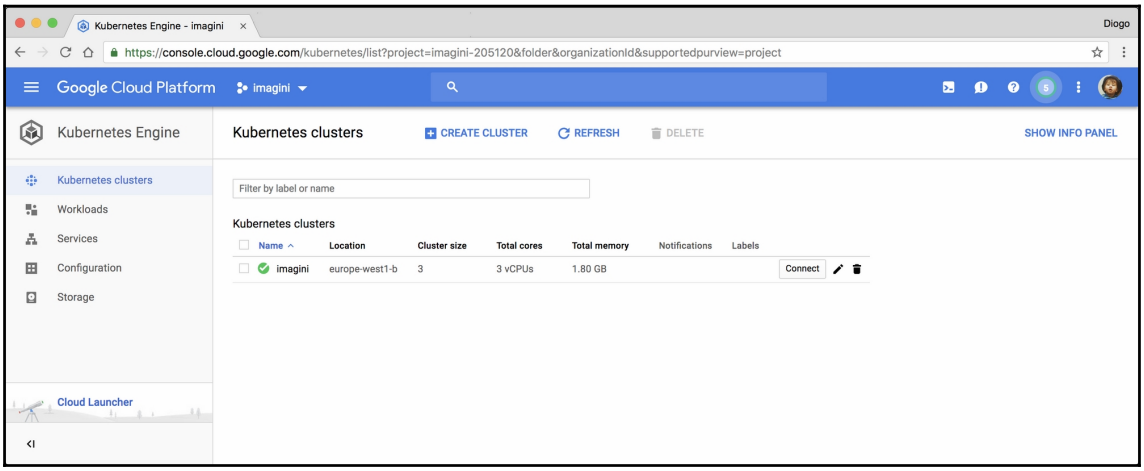


Creating a Kubernetes cluster

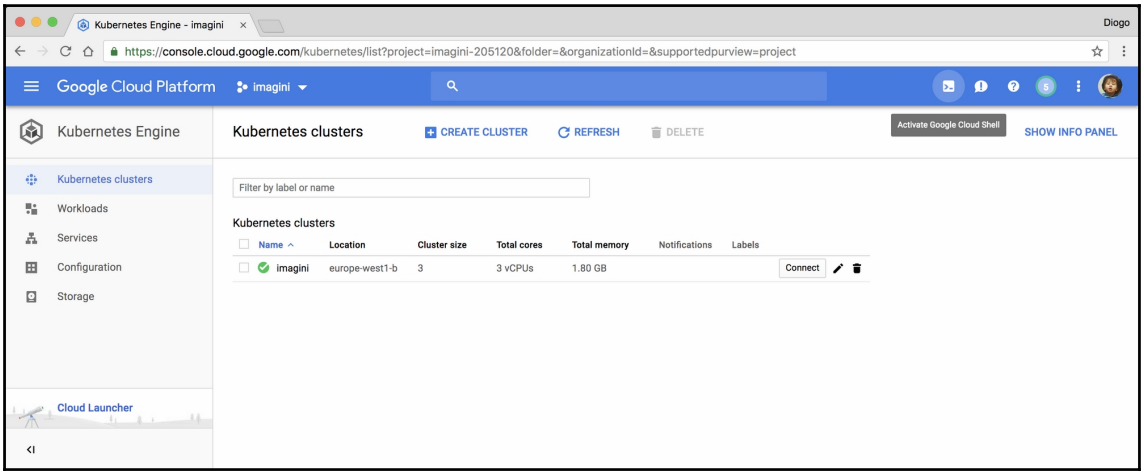
Previously, for a few items, you may have noticed the **Kubernetes Engine** option. That's where we're heading to prepare our cluster:



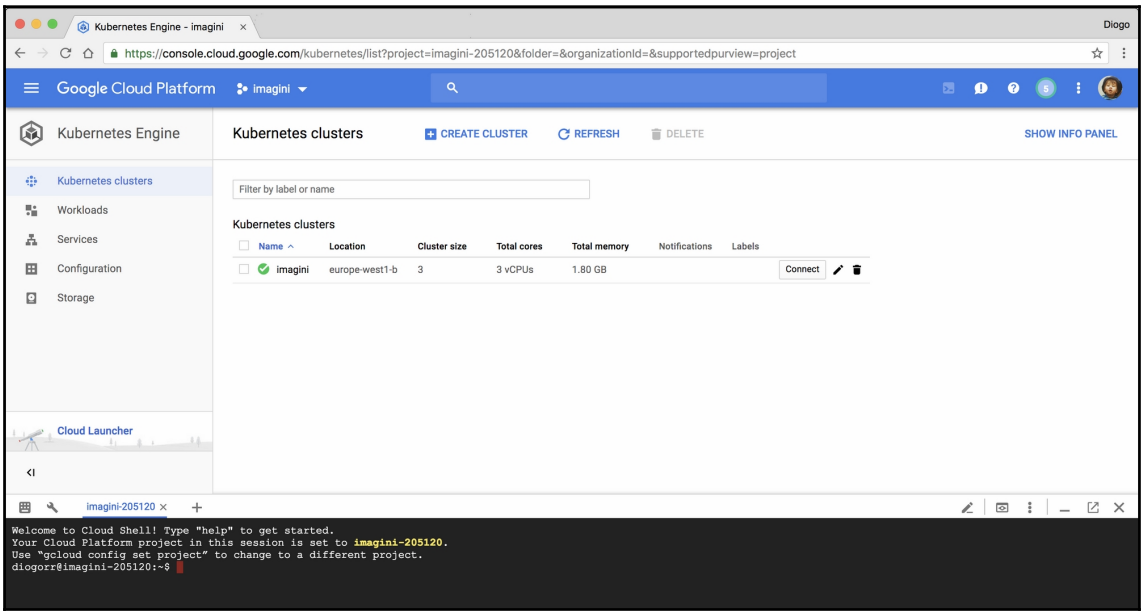
You'll be asked to create a new cluster. Choose the default options, or change the ones you want to try out, and create a cluster for our microservice to run on:



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**:



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:



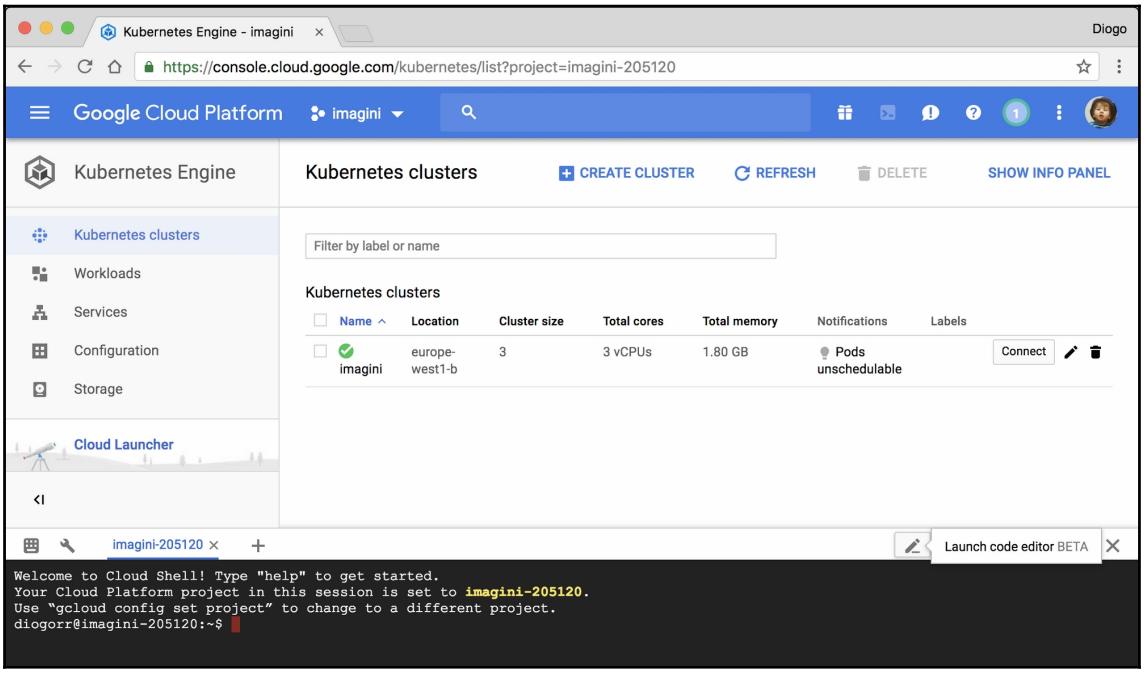
This is a normal console, but with the `gcloud` and `kubectl` commands already installed. You can have those installed locally and manage your clusters remotely, or just access this console to make a quick change or checkup.

Creating our microservice

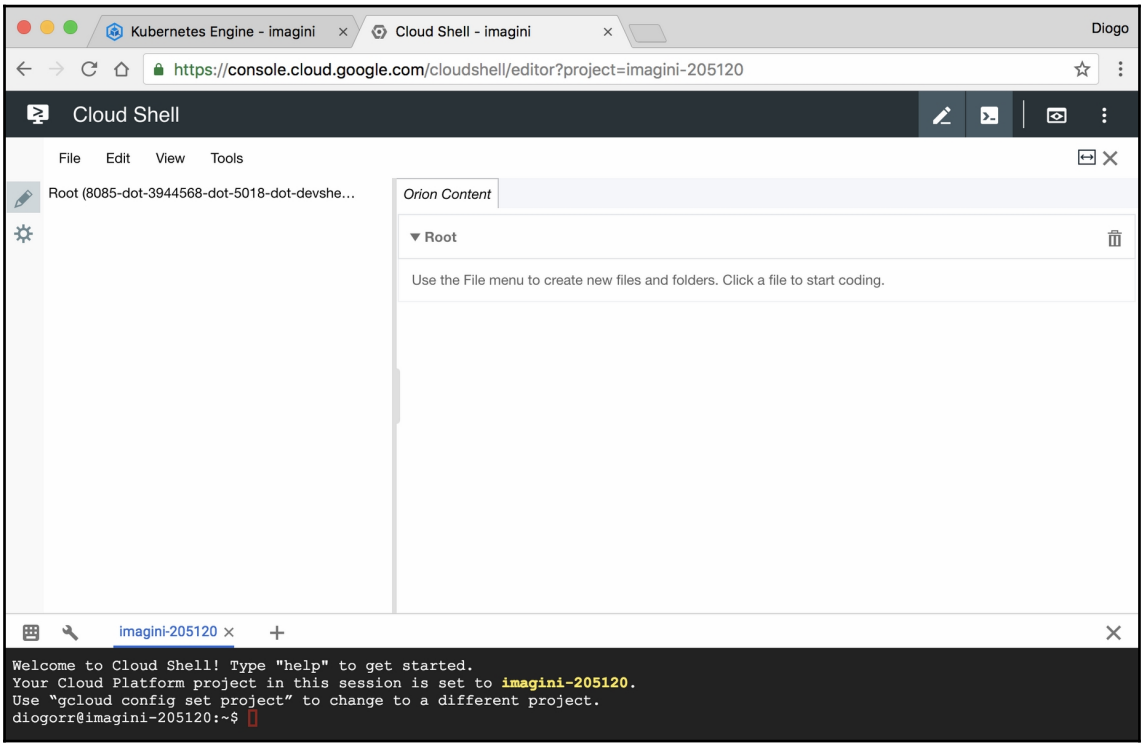
Similar to what we did before, these are the steps we need to perform before successfully deploying to our new cluster:

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.

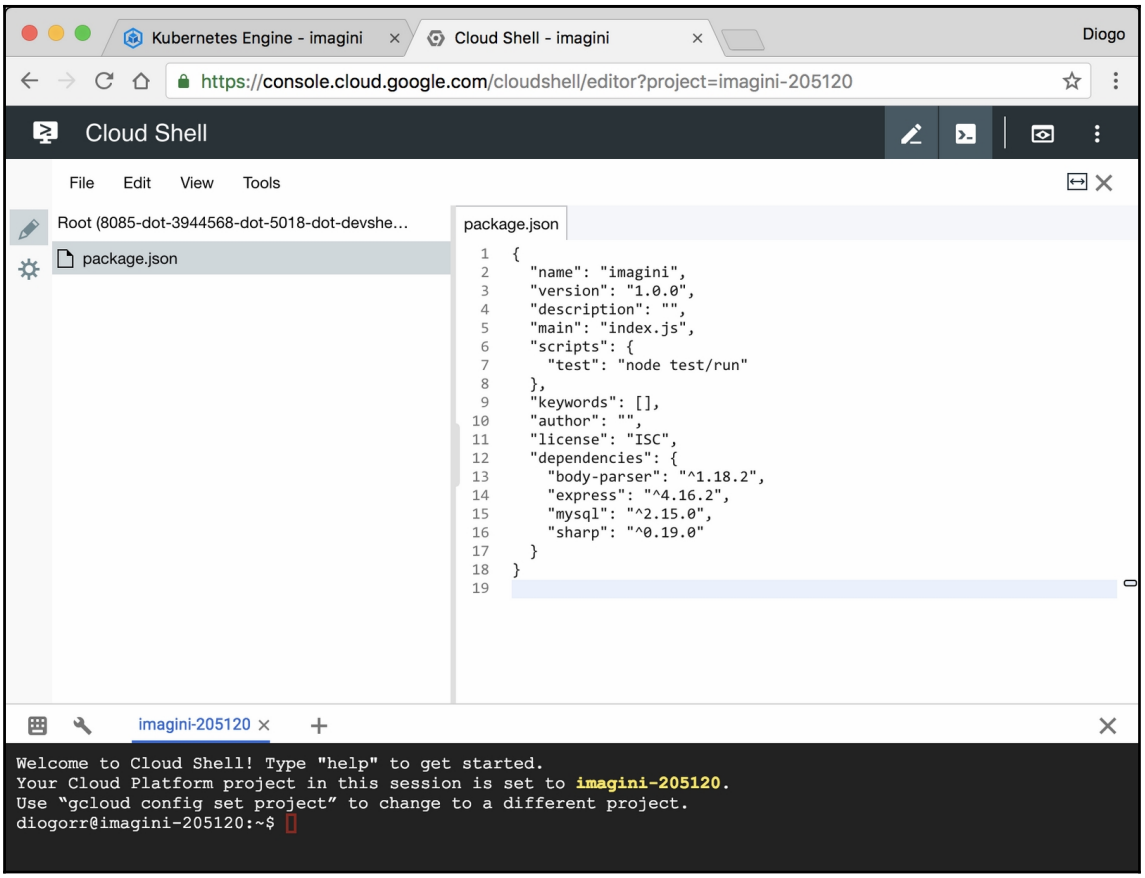
First, notice the pencil icon in the top-right corner of the console. It's an online editor we can use to create the files we need:



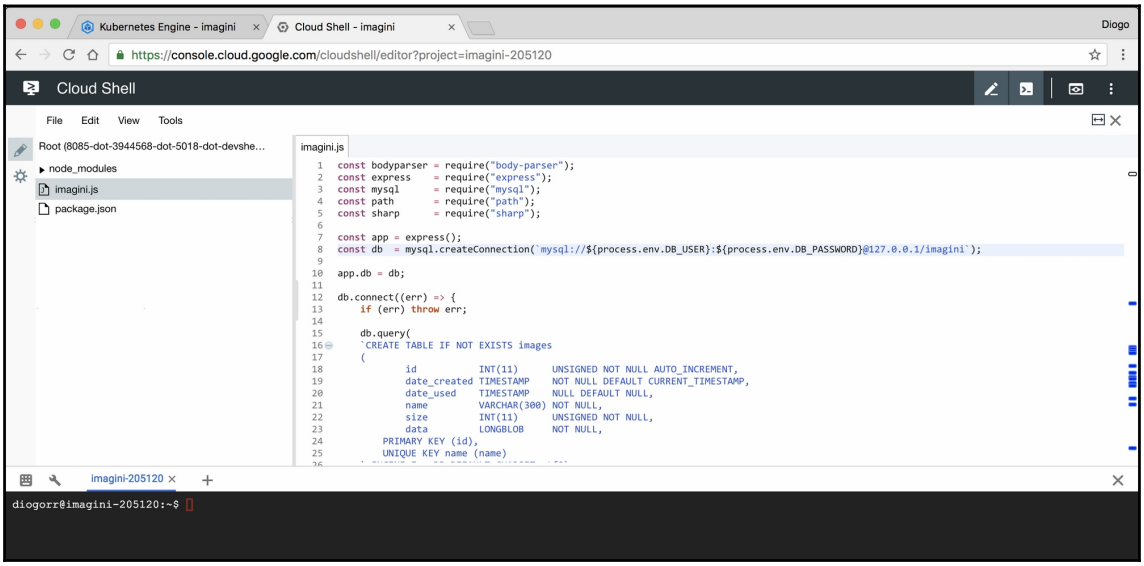
Click on it. It should open a new tab with an empty editor. This represents the root folder of the console you have on the bottom:



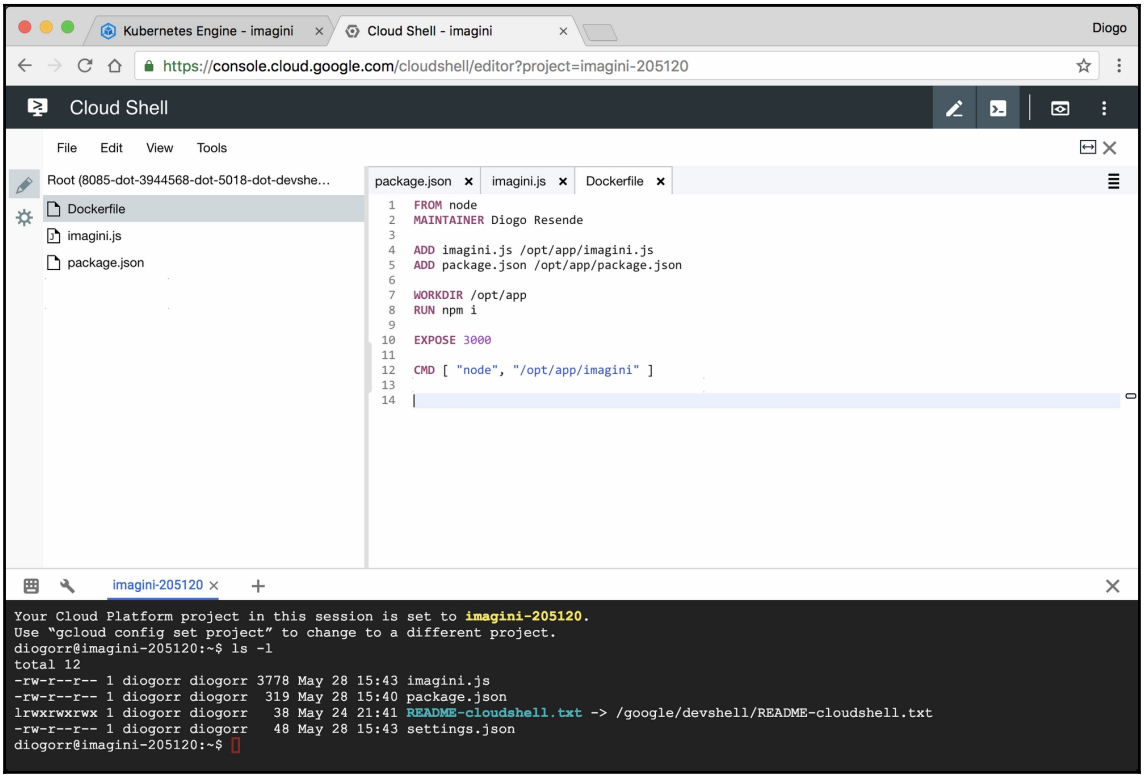
Let's start by creating our `package.json` file. Go to **File** and click **New**, and then **File**. Type `package.json` and insert our dependencies. Remember, we're using MySQL this time:



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:



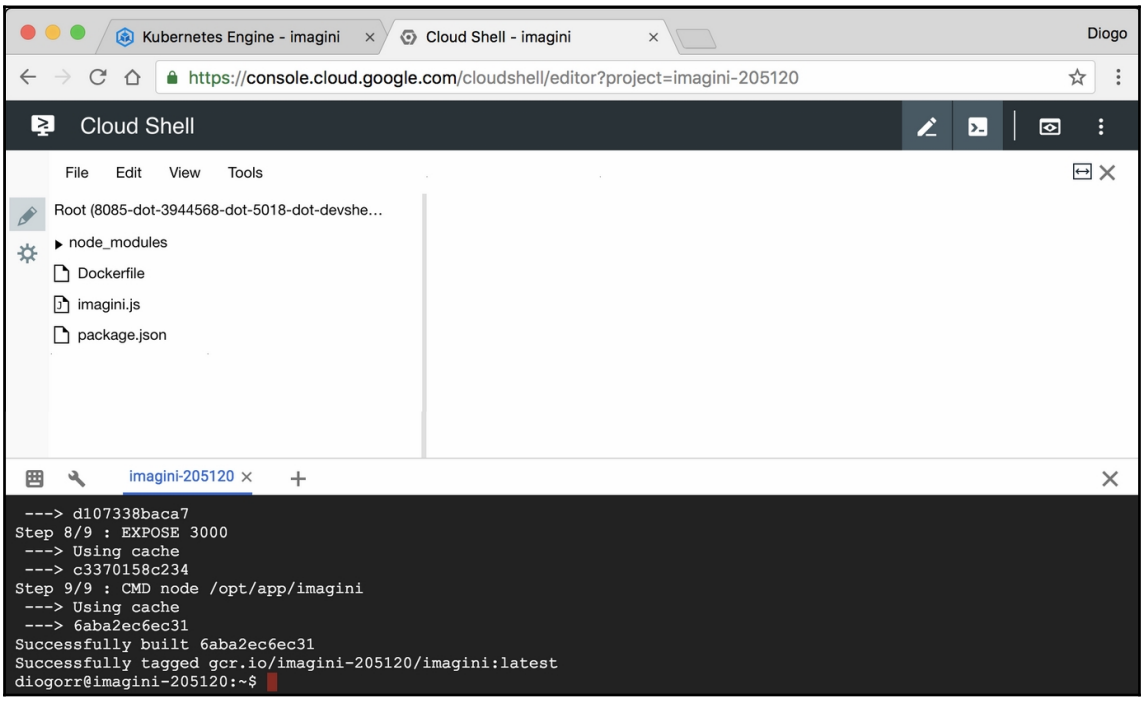
Then, create the `Dockerfile` for our service:



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/imagini-205120/imagini
```

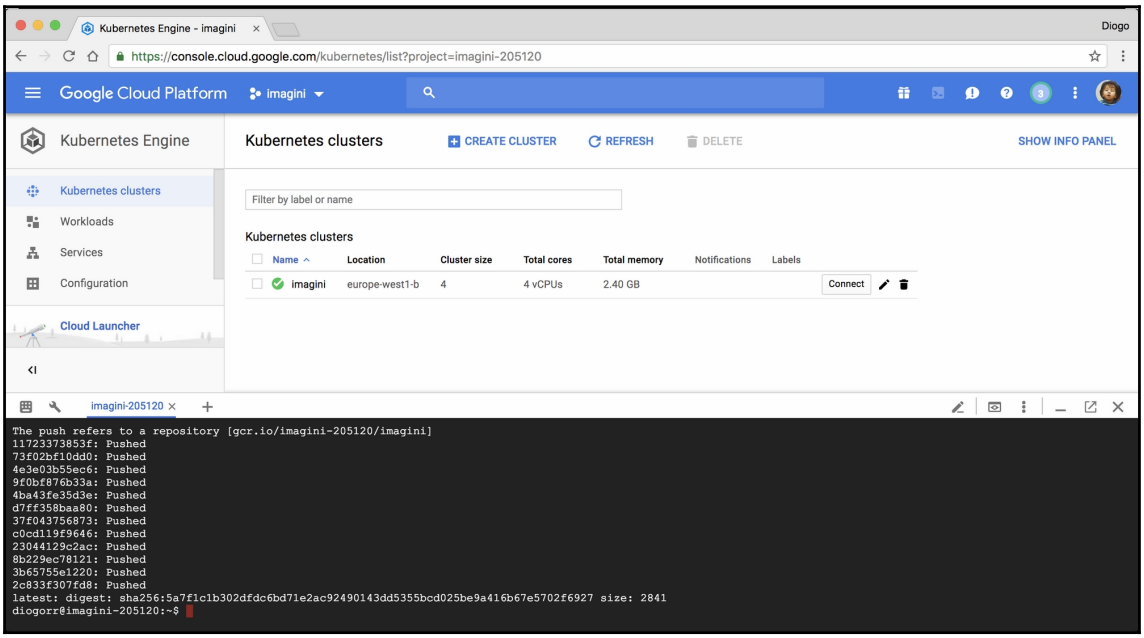
We're using a more composed name because we're publishing to the **Google Container Registry** (**GCR**), and so it's good practice to use the project ID (as I did) and then the name of the service:



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

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

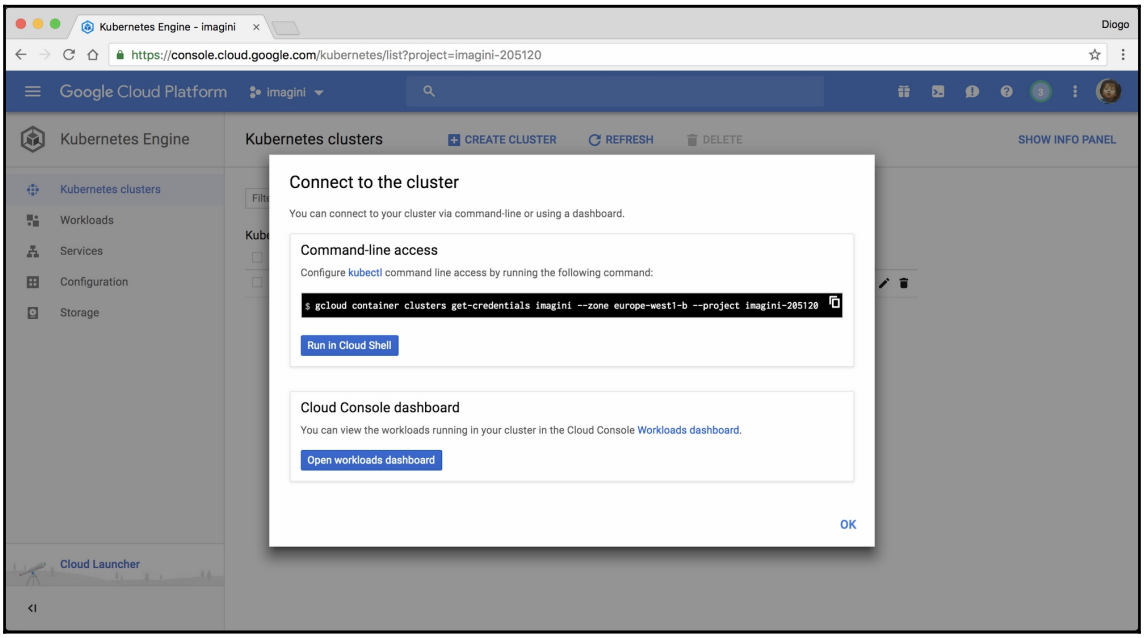
If everything goes right, you should see a list of layers of the image being pushed (published to the registry) and, in the end, a digest and the size of the image, as shown here:



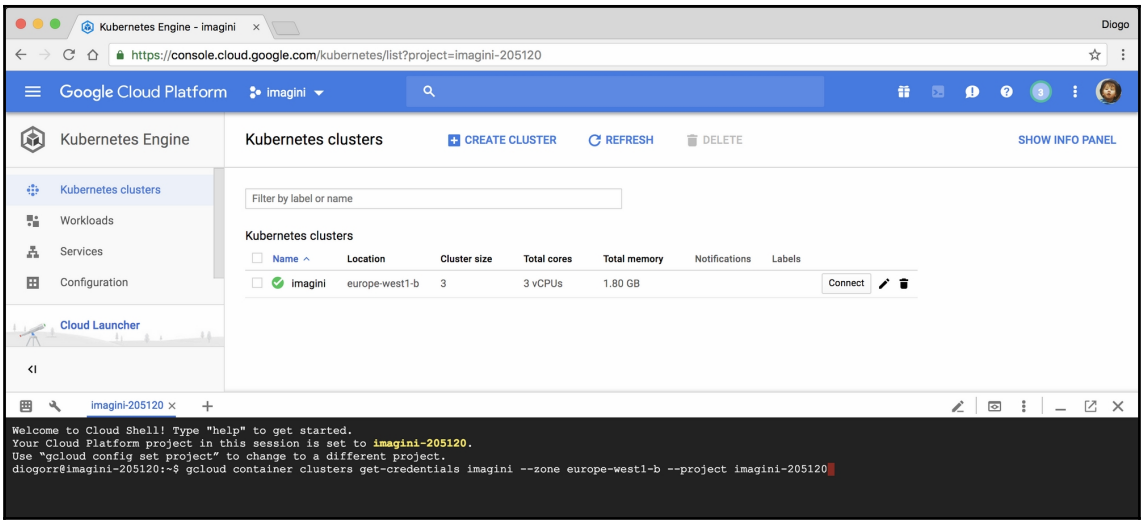
You may notice that I closed the editor between commands. You can use any console, and can even have more than one open.

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.

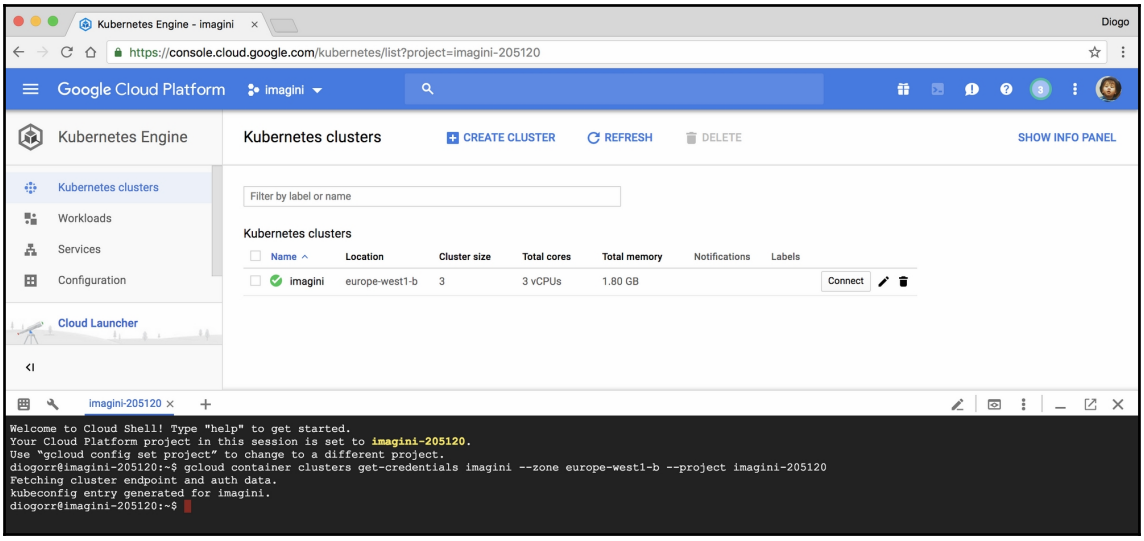
First, close the console and click the **Connect** button that is located on the right-hand side of the cluster name:



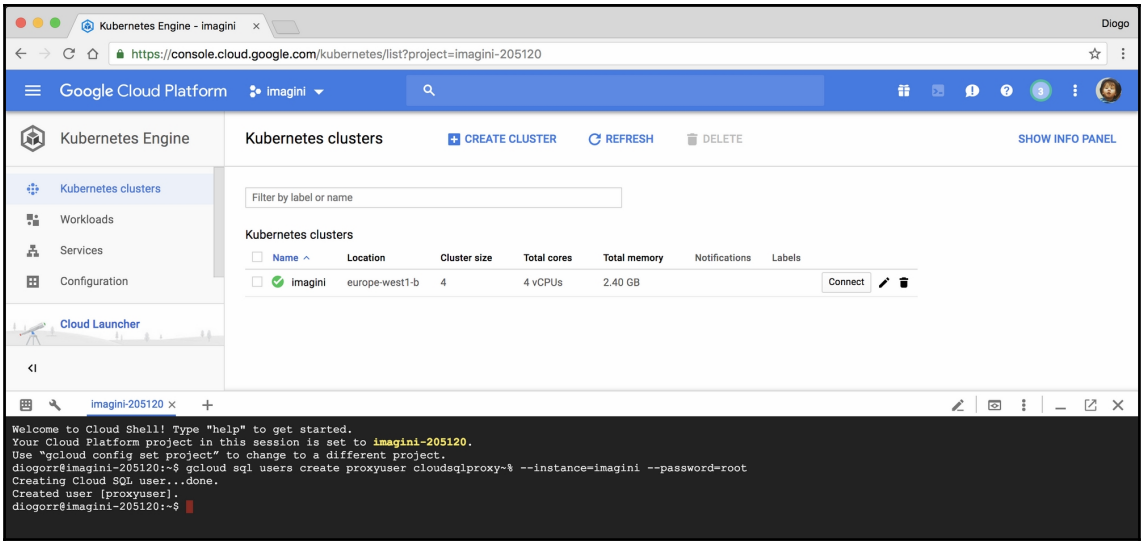
Click **Run in Cloud Shell**. This will reopen the console, but will type in the command to connect to our cluster:



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



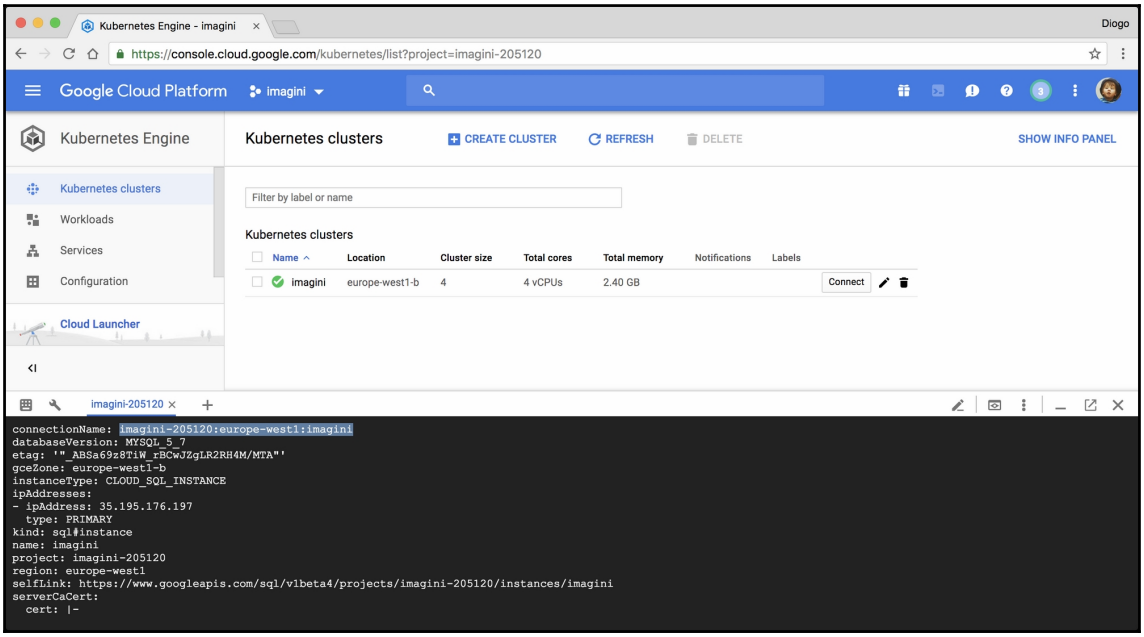
Then, we'll use the `gcloud` command to create the proxy user account that will be used by the SQL proxy to access our Cloud SQL instance:



Now, type the following command:

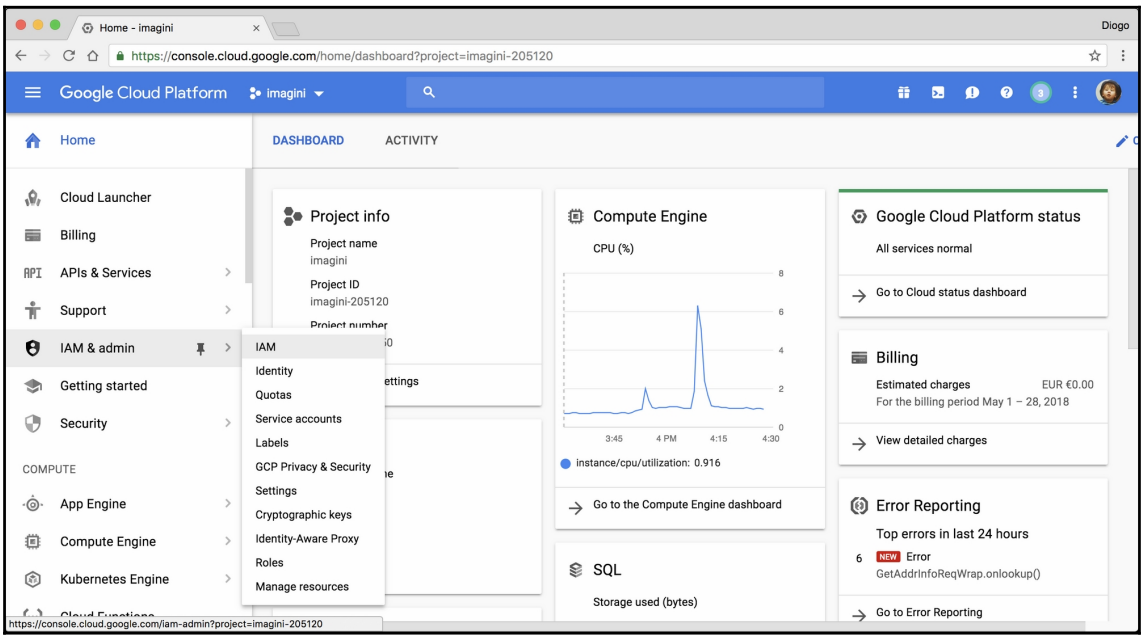
```
gcloud sql instances describe imagini
```

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.

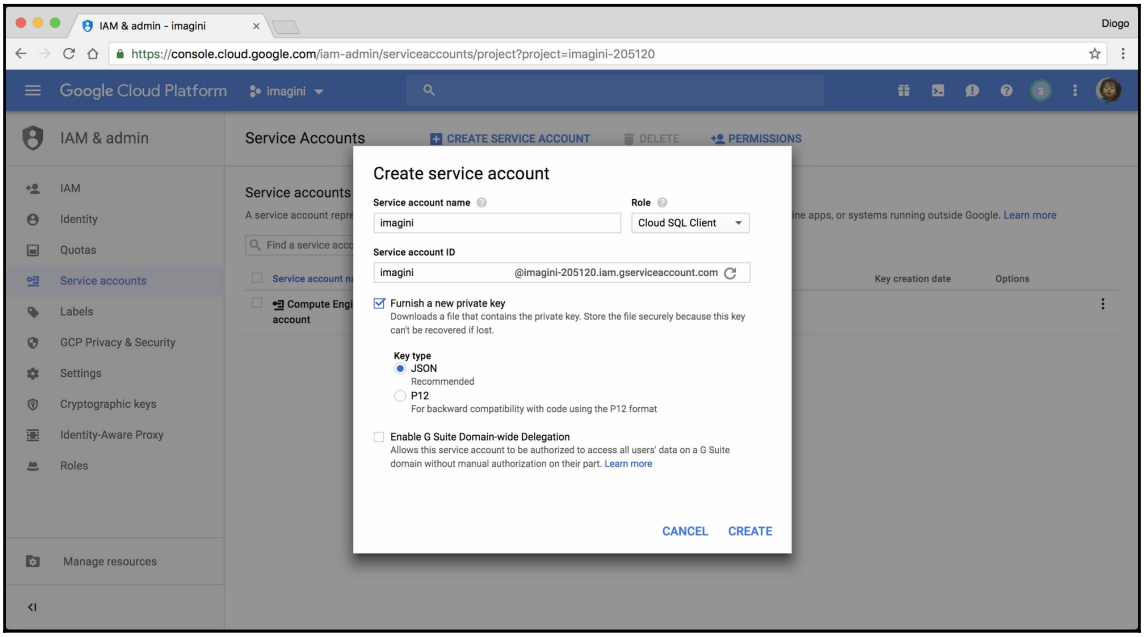


Copy the name down and save it for later. We'll need it when we create our deployment configuration.

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

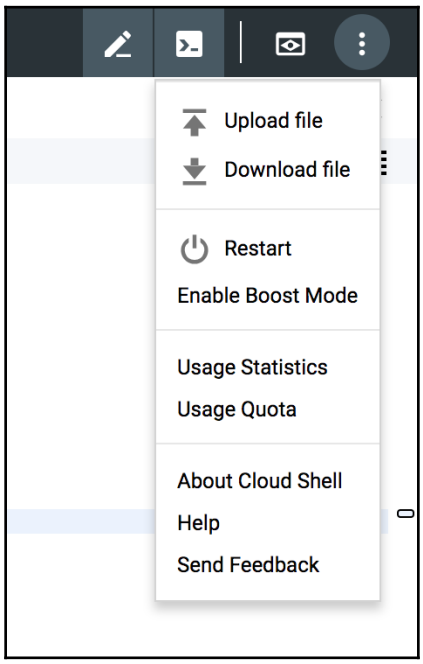


Create a service account with the **Cloud SQL Client** role, and check **Furnish a new private key**:

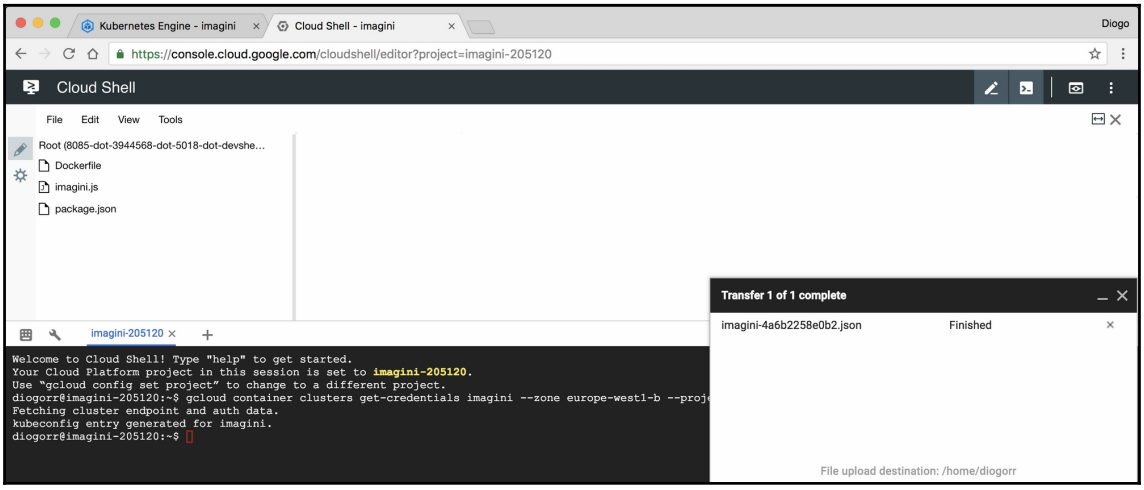


After hitting **Create**, the account will be created and a JSON file will be given to you to save in your local disk. Head back to the cluster, open the console, and then click to open the editor again.

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



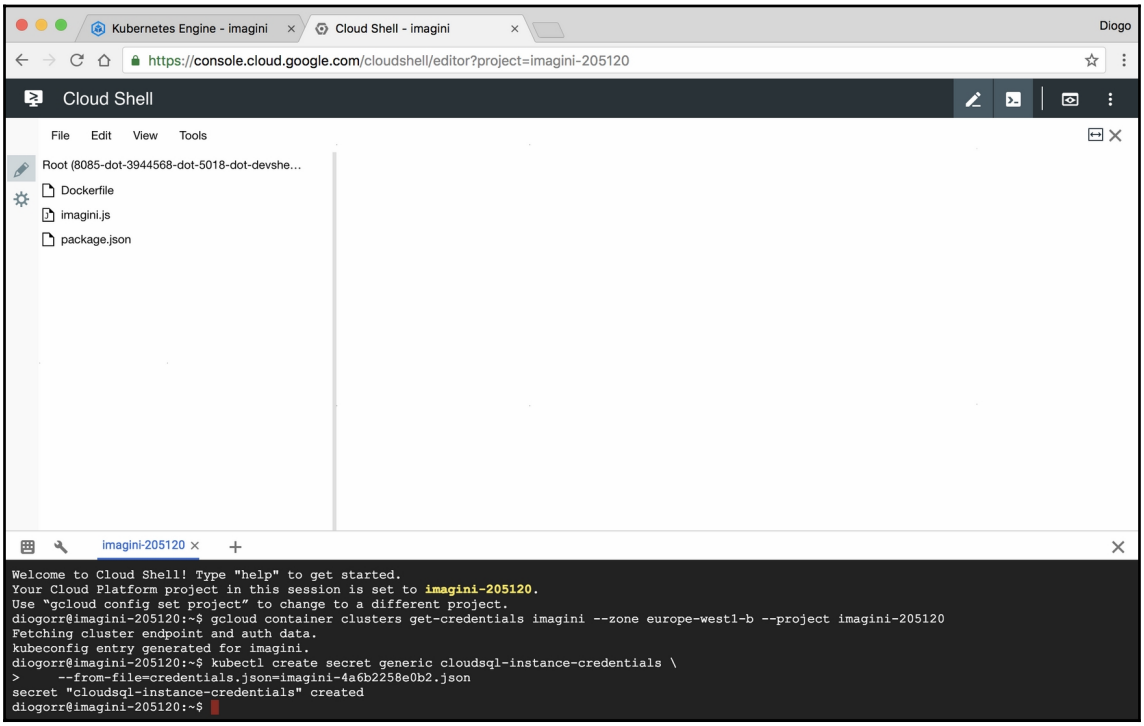
Pick the JSON file you just saved and upload it using that option:



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 cloudsql-instance-credentials --from-
file=credentials.json=imagini-4a6b2258e0b2.json
```

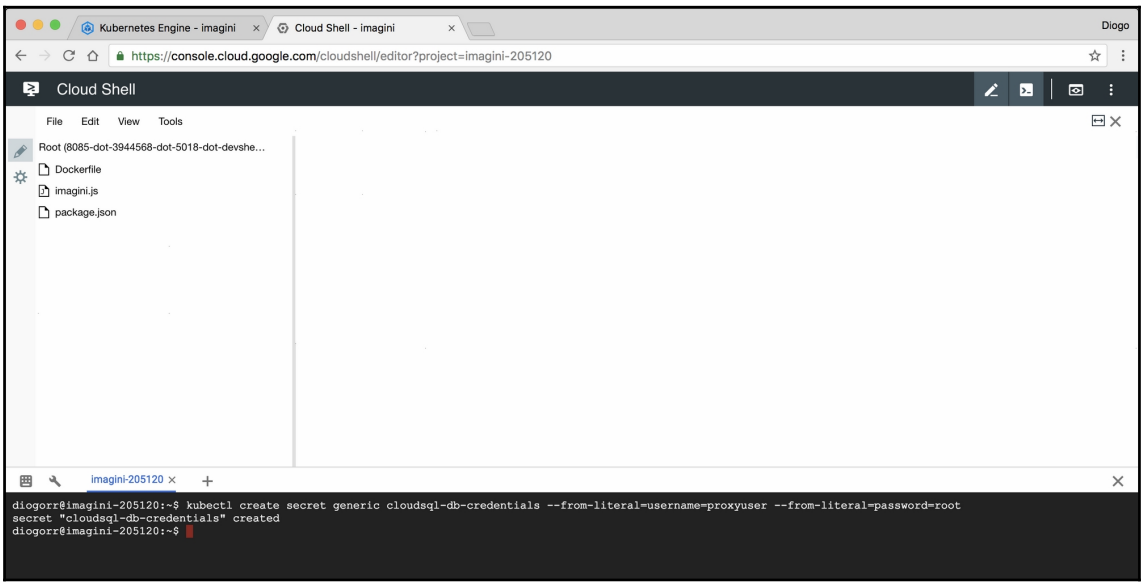
The following screenshot shows the result of the preceding command executed in the console. You should see something like **secret "cloudsql-instance-credentials" created**.



Now, let's create the database credentials secret:

```
kubectl create secret generic cloudsql-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**.



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
```