Scalable and resilient Django with Kubernetes

harishnarayanan.org/writing/kubernetes-django/

If things work out as you've envisioned, there will be a time in your webapp's lifecycle when it's serving a large number of users. By the time things get to this point, it's ideal if you've architected your webapp to both *scale* gracefully to meet this load, and also be *resilient* to arbitrary failures of underlying compute resources.

This article is about how you can use Docker containers and Kubernetes to help your Django webapp achieve these architectural goals. While it meanders a bit through theory and philosophy, it does work up to a concrete example to help solidify concepts.

Caveats

Before we get too deep into the weeds, I'd like to note that the ideas expressed in this piece don't have anything particular to do with Django. I've simply chosen it as an example because it is a popular framework that I'm familiar with. It is straightforward to repurpose these principles for other software stacks.

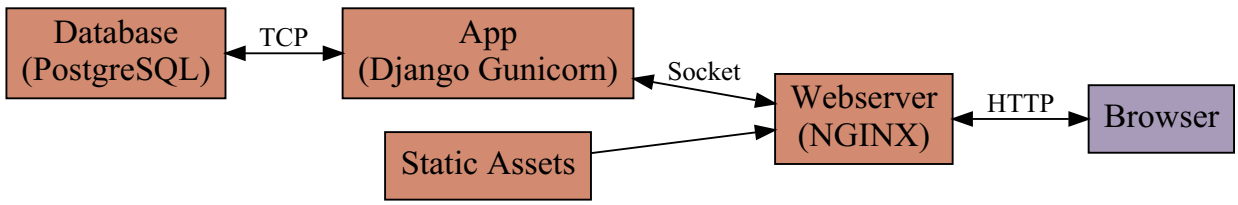
I'd also like to point out that this article juggles many moving pieces — some quite immature. If you can avoid this level of complexity at the current stage of your webapp's lifecycle, *you should*. Instead, focus your efforts on better understanding your users' problems and testing whether your app solves them. No one is going to know or complain that you're running your app on a single fragile server until enough people care to use your app regularly.

No one.

With that out of the way, let's get started!

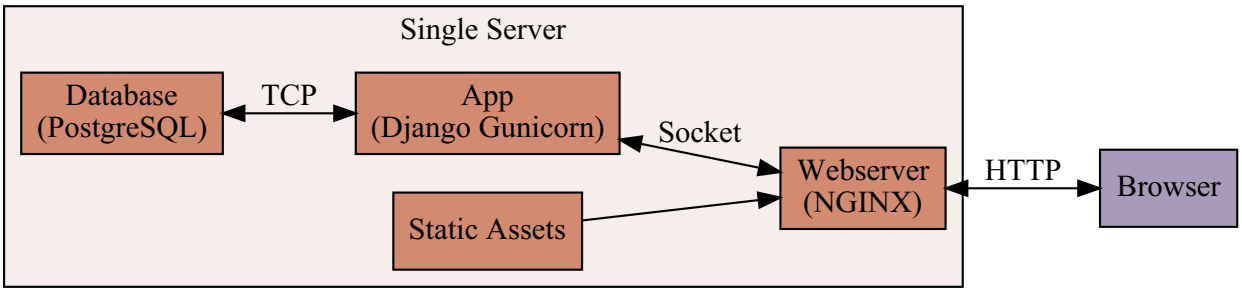
There are some problems with traditional VM-based deployments

Let's imagine that you're working on a Django webapp that's laid out in a fairly standard fashion: All your app's data resides in a PostgreSQL database. The app itself is written in Django-flavoured Python, and is served up using the Gunicorn application server. And in front of all this, you have the NGINX web server acting both as a reverse proxy and a static content server.



Layout of a non-trivial Django application.

When you're first starting out with your app and you only have a handful of users, it makes perfect sense to run all these pieces on a single server. So you run up to your favourite cloud provider, fire up a VPS running Debian or whatever, and install all these individual bits of software on the same machine.

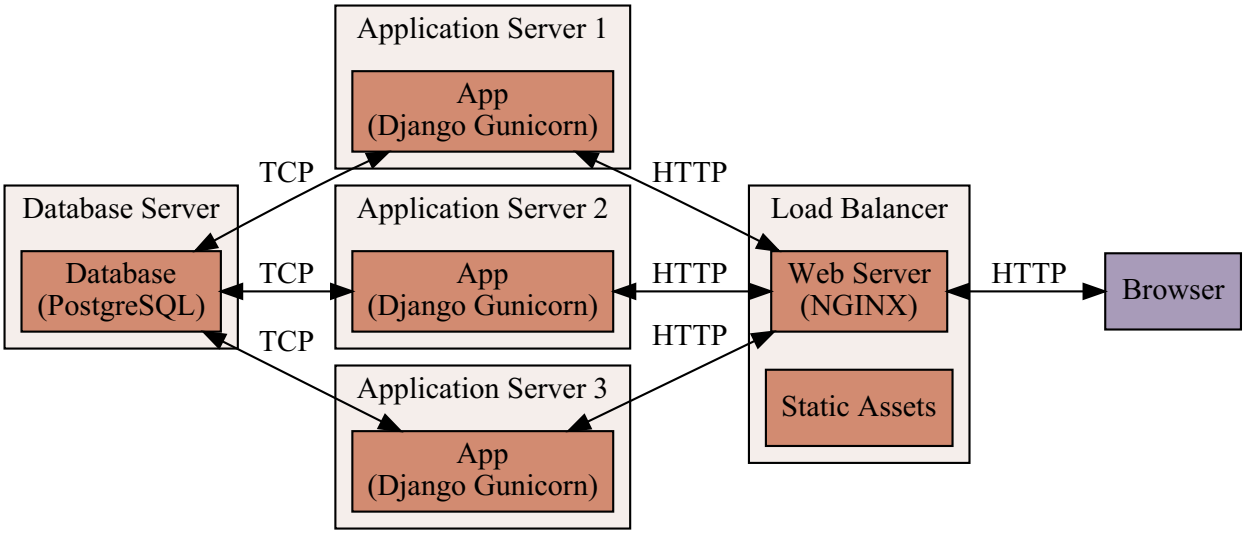


All pieces making up the app on a single machine.

Then, as your app starts to get more popular, you begin to work on scaling. At first, you follow the straightforward approach and simply provision larger and larger single machines to run your app on. This is called *vertical scaling* and works well until you reach a few thousand users.

And then your app gets *even more* popular.

Now you realise that if you were to split the components making up your app and put them on separate machines, you can scale the components independently. Meaning, for example, that you can run multiple instances of your Django app (called *horizontal scaling*) to handle your growing user base, while continuing to run your PostgreSQL server on only one (but potentially increasingly powerful) machine.



Running many instances of the app, talking to a single database.

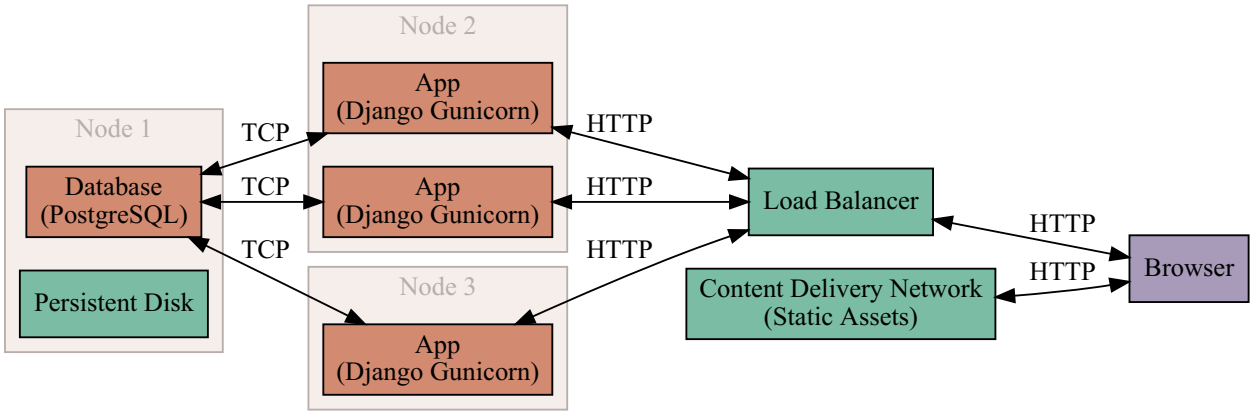
This is actually a pretty good deployment solution (and it's the basic idea underlying what we use today in practice at my day job, using Ansible to setup the servers), but it comes with its own set of inconveniences:

1. It's annoying to provision, setup and keep up-to-date one server for each component. This is not the level at which you want to be thinking about the system.
2. You generally have poor resource utilisation because each component doesn't effectively use all that the server it's running on has to offer. This is primarily because you're often setup for peak load, not median load.

3. If you try to resolve (2) by running multiple components (e.g. both your app and database) on the same machine, then there's nothing stopping one piece from clobbering the others. i.e. There is poor resource isolation within a given server.

So, what if we could shift our attention from managing servers to simply running the components of our app on a collection of computing resources? Furthermore, what if these components were well isolated from each other and efficiently used the resources they had at their disposal?

Then our deployment picture might look more like the following, where the primary pieces we care about (the application components) are shown in orange. The actual nodes (physical or virtual machines) that the components run on are de-emphasised visually because we don't care about the details. And we trust our underlying compute infrastructure to offer us some primitives such as persistent storage and load balancers (shown in green) that are common to any non-trivial webapp.



The application running on an abstract collection of resources.

This philosophical shift — from *managing servers* to simply *running components of our app* ideally — is precisely the promise of container technology like Docker and cluster orchestration frameworks like Kubernetes. And in the practical example that follows, we'll see how these tools allow us to easily recreate the ideal deployment scenario shown above.

So how exactly do Docker and Kubernetes help?

You can colloquially think of Docker containers as fat static binaries of your apps. They bundle your application code, the underlying libraries and all the necessary bits your app needs to run into a convenient package — one that can be run on a thin layer directly over the Linux kernel. What this means in practice is that you can take a container that you've built once and run it on different versions of Linux distributions, or entirely different Linux distributions. Everything should work seamlessly.

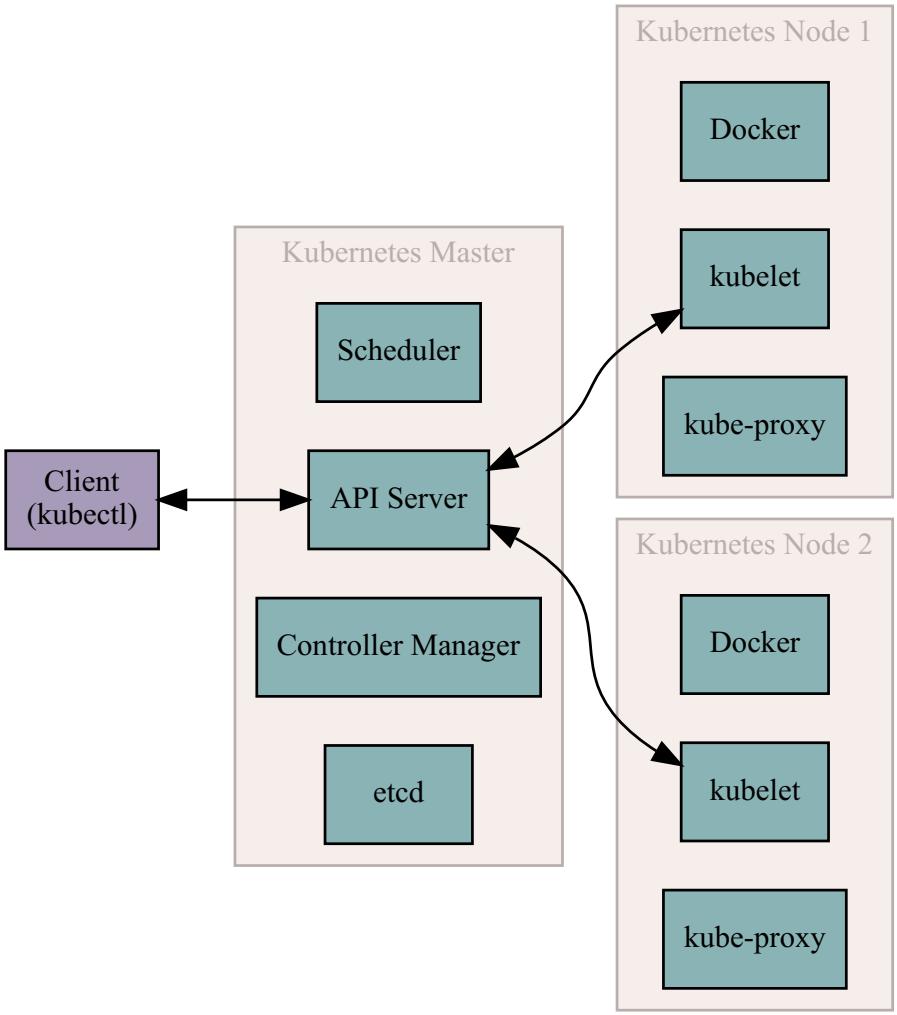
Thus, by forming an atomic unit that can be built, tested and run anywhere, containers raise your level of concern above the specifics of the operating system, allowing you to focus on your app. Containers also offer resource isolation, meaning that if two of them are running side by side, each can only see and do what they're supposed to.

This means our deployment journey can now be broken down into two coarse steps. The first is to take the different components of our app and package them into containers. The second is to run these on our computing resources — leveraging underlying computing primitives such as *load balancers*, and ensuring that the containers are properly networked.

This second step is where Kubernetes comes in.

Kubernetes is an open source system for managing clusters and deploying “containerised” applications. Kubernetes abstracts the underlying hardware (of your cloud provider or on-premises cluster), and presents a simple API that allows you to easily control it. You send this API some declarative state, e.g. “I’d like three copies of my Django app container running behind a load balancer, please,” and it ensures that the appropriate containers are scheduled on the nodes of your cluster. Furthermore, it monitors the situation and ensures that this state is maintained, allowing it to be robust to arbitrary changes in the system. This means, for example, that if a container is shut down prematurely because a node runs out of memory, Kubernetes will notice this and ensure another copy is restarted elsewhere.

Kubernetes works by having agents that sit on each node of your cluster. These allow for things like running Docker containers (the *docker daemon*), making sure the desired state is maintained (the *kubelet*), and that the containers can talk to each other (*kube-proxy*). These agents listen to and synchronise with a centralised API server to ensure that the system is in the desired state.



A simplified look at Kubernetes' architecture.

The Kubernetes API exposes a collection of cluster configuration resources that we can modify to express the state we want our cluster to be in. The API offers a standard REST interface, allowing us to interact with it in a multitude of ways. In the upcoming example, we’re going to be using a thin command line client called `kubectl` to communicate with the API server.

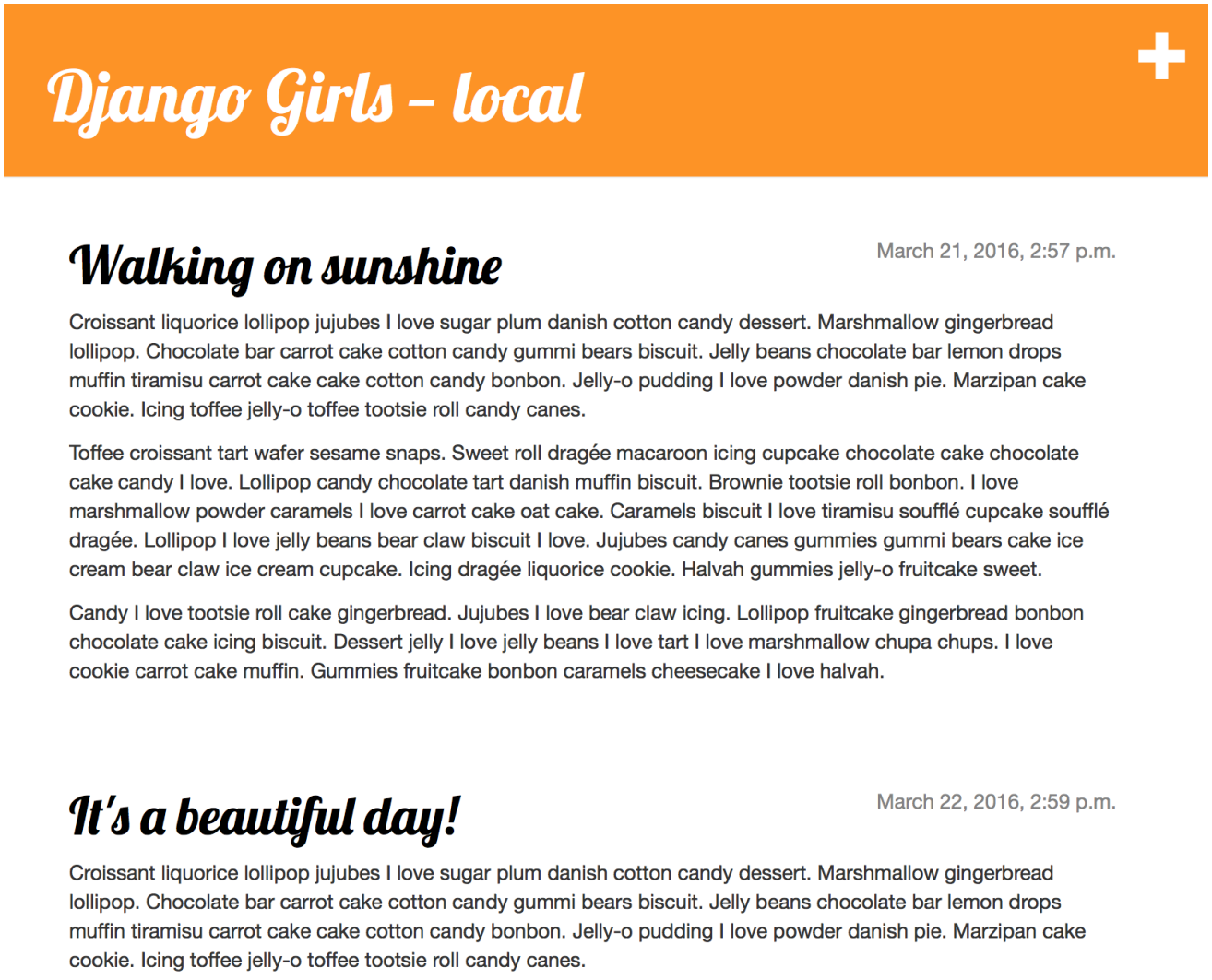
While the API offers numerous primitives to work with, here are a few that are important for our example today:

- **Pods** are a collection of closely coupled containers that are scheduled together on the same node, allowing them to share volumes and a local network. They are the smallest units that can be deployed within a Kubernetes cluster.
- **Labels** are arbitrary key/value pairs (e.g. `name: app` or `stage: production`) associated with Kubernetes resources. They allow for an easy way to select and organise sets of resources.
- **Replication Controllers** ensure that a specified number of pods (of a specific kind) are running at any given time. They group pods via labels.
- **Services** offer a logical grouping of a set of pods that perform the same function. By providing a persistent name, IP address or port for this set of pods, they offer service discovery and load balancing.

If this all seems a bit too abstract at the moment, do not fret. We're now going to jump into an example that demonstrates how these bits work in practice to help us deploy our Django app.

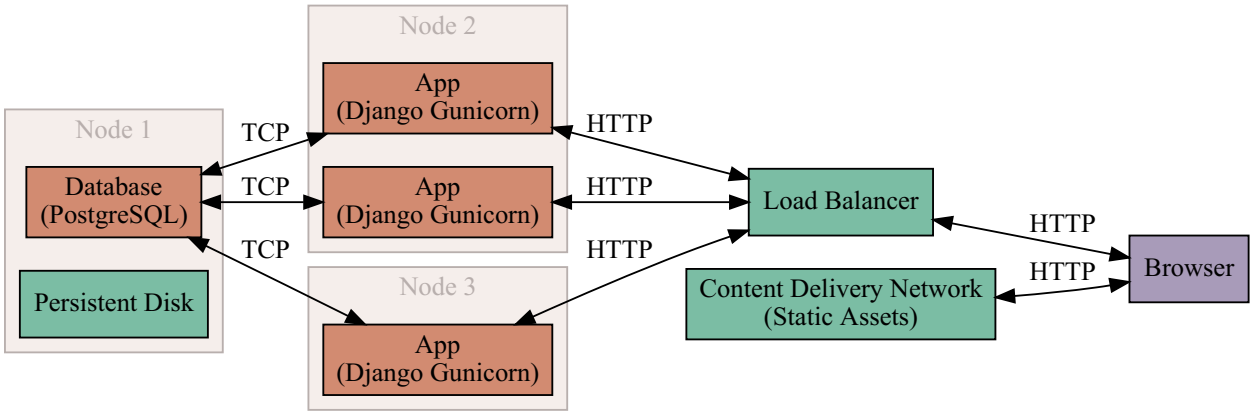
Practical example on Google Container Engine

The example application that we're going to be focusing on is a simple blog application.



Sample blog app following the Django Girls Tutorial.

While this is a very basic example, it contains all the necessary pieces we need to see the ideas we've discussed in practice. Over the course of this example, we're going to get access to a cluster controlled by Kubernetes, split our blog application into separate Docker containers, and deploy them using Kubernetes. The final result matches the idealised diagram introduced earlier.



The application running on an abstract collection of resources.

And once we have things up and running, we'll play around with the Kubernetes API to do different things, such as scaling your app, observe how it heals from failures, and learn how you can upgrade one version of your Django app to another with no downtime.

Preliminary steps

1. Fetch the source code for this example.

   ```
   git clone https://github.com/hnarayanan/kubernetes-django.git
   ```

2. Install Docker.

3. Take a look at and get a feel for the example Django application used in this repository. It is a simple blog that's built following the excellent Django Girls Tutorial.

4. Setup a cluster managed by Kubernetes. The effort required to do this can be substantial, so one easy way to get started is to sign up (for free) on Google Cloud Platform and use a managed version of Kubernetes called Google Container Engine (GKE).

    1. Create an account on Google Cloud Platform and update your billing information.
    2. Install the command line interface.
    3. Create a project (that we'll refer to henceforth as `$GCP_PROJECT`) using the web interface.
    4. Now, we're ready to set some basic configuration.

        ```
        gcloud config set project $GCP_PROJECT
        gcloud config set compute/zone europe-west1-d
        ```

    5. Then we create the cluster itself.

        ```
        gcloud container clusters create demo
        gcloud container clusters list
        ```

    6. Finally, we configure `kubectl` to talk to the cluster.

        ```
        gcloud container clusters get-credentials demo
        kubectl get nodes
        ```

Create and publish Docker containers

For this example, we'll be using Docker Hub to host and deliver our containers. And since we're not working with any sensitive information, we'll expose these containers to the public.

PostgreSQL

Build the container, remembering to use your own username on Docker Hub instead of `hnarayanan`:

```
cd containers/database
docker build -t hnarayanan/postgresql:9.5 .
```

You can check it out locally if you want:

```
docker run --name database -e POSTGRES_DB=app_db -e POSTGRES_PASSWORD=app_db_pw -e
POSTGRES_USER=app_db_user -d hnarayanan/postgresql:9.5

docker exec -i -t $PROCESS_ID bash
```

Push it to a repository:

```
docker login
docker push hnarayanan/postgresql:9.5
```

Django app running within Gunicorn

Build the container:

```
cd containers/app
docker build -t hnarayanan/djangogirls-app:1.2-orange .
```

Push it to a repository:

```
docker push hnarayanan/djangogirls-app:1.2-orange
```

We're going to see how to perform rolling updates later in this example. For this, let's create an alternative version of our app that simply has a different header colour, build a new container app and push that too to the container repository.

```
cd containers/app
emacs blog/templates/blog/base.html

# Add the following just before the closing </head> tag
    <style>
      .page-header {
        background-color: #ac4142;
      }
    </style>

docker build -t hnarayanan/djangogirls-app:1.2-maroon .
docker push hnarayanan/djangogirls-app:1.2-maroon
```

Deploy these containers to the Kubernetes cluster

PostgreSQL

Even though our application only requires a single PostgreSQL instance running, we still run it under a (pod) replication controller. This way, we have a service that monitors our database pod and ensures that one instance is running even if something weird happens, such as the underlying node fails.

```
cd  kubernetes/database
kubectl create -f replication-controller.yaml

kubectl get rc
kubectl get pods

kubectl describe pod <pod-id>
kubectl logs <pod-id>
```

Now we start a service to point to the pod.

```
cd  kubernetes/database
kubectl create -f service.yaml

kubectl get svc
kubectl describe svc database
```

Django app running within Gunicorn

We begin with three app pods (copies of the orange app container) talking to the single database.

```
cd kubernetes/app
kubectl create -f replication-controller-orange.yaml
kubectl get pods

kubectl describe pod <pod-id>
kubectl logs <pod-id>
```

Then we start a service to point to the pod. This is a load-balancer with an external IP so we can access the site.

```
cd kubernetes/app
kubectl create -f service.yaml
kubectl get svc
```

Before we access the website using the external IP presented by `kubectl get svc`, we need to do a few things:

1. Perform initial migrations:

   ```
   kubectl exec <some-app-orange-pod-id> -- python /app/manage.py migrate
   ```

2. Create an intial user for the blog:

   ```
   kubectl exec -it <some-app-orange-pod-id> -- python /app/manage.py
   createsuperuser
   ```

3. Have a CDN host static files since we don't want to use Gunicorn for serving these. This demo uses Google Cloud storage, but you're free to use whatever you want. Just make sure `STATIC_URL` in `containers/app/mysite/settings.py` reflects where the files are.

   ```
   gsutil mb gs://demo-assets
   gsutil defacl set public-read gs://demo-assets
   cd django-k8s/containers/app
   virtualenv
   source venv/bin/activate
   pip install Django==1.9.5
   export DATABASE_ENGINE='django.db.backends.sqlite3'
   ./manage.py collectstatic
   gsutil -m cp -r static
   ```

At this point you should be able to load up the website by visiting the external IP for the app service (obtained by running `kubectl get svc`) in your browser.

Go to `http://app-service-external-ip/admin/` to login using the credentials you setup earlier (while creating a super user), and return to the site to add some blog posts. Notice that as you refresh the site, the name of the app pod serving the site changes, while the content stays the same.

Play around to get a feeling for Kubernetes' API

Now, suppose your site isn't getting much traffic, you can gracefully *scale* down the number of running application pods to one. (Similarly you can increase the number of pods if your traffic starts to grow!)

```
kubectl scale rc app-orange --replicas=1
```

```
kubectl get pods
```

You can check *resiliency* by deleting one or more app pods and see it respawn.

```
kubectl delete pod <pod-id>
kubectl get pods
```

Notice Kubernetes will spin up the appropriate number of pods to match the last known state of the replication controller.

Finally, to show how we can migrate from one version of the site to the next, we'll move from the existing orange version of the application to another version that's maroon.

First we scale down the orange version to just one copy:

```
kubectl scale rc app-orange --replicas=1
kubectl get pods
```

Then we spin up some copies of the new maroon version:

```
cd kubernetes/app
kubectl create -f replication-controller-maroon.yaml
kubectl get pods
```

Notice that because the app service is pointing simply to the label `name: app`, both the one orange and the three maroon apps respond to http requests to the external IP.

When you're happy that the maroon version is working, you can spin down all remaining orange versions, and delete its replication controller.

```
kubectl scale rc app-orange --replicas=0
kubectl delete rc app-orange
```

Cleaning up

After you're done playing around with this example, remember to cleanly discard the compute resources we spun up for it.

```
gcloud container clusters delete demo
gsutil -m rm -r gs:
```

In conclusion

This article covered a lot of ground. We first motivated the need for containers and cluster orchestration frameworks in general. We then saw how Docker and Kubernetes in particular help us deploy a Django application that can scale gracefully to meet loads, while simultaneously being resilient to arbitrary failures of underlying compute resources.

While this is a good introduction to concepts, there are a few details I glossed over which you will want to consider carefully before deciding if Kubernetes is right for you.

The first is that the setup of a Kubernetes cluster (when not using a hosted version like Google Container Engine, as in our example) is non-trivial. And while Kubernetes attempts to abstract away the underlying hardware, the actual experience you have using it is quite dependent on the actual infrastructure you're running on. So do play around

with it in your environment to gauge if the complexity is worth it for you.

The second is that our example deployment needs more work using additional Kubernetes primitives before it becomes useful in practice. These include using:

- *Persistent Volumes* (and *Persistent Volume Claims*) to ensure that the PostgreSQL data is persistent beyond the life of its pod.
- *Secrets* to handle the database password and other sensitive information.
- *Horizontal Pod Autoscaling* to automatically adjust the number of running pods based on observed CPU utilisation.
- *Daemon Sets* to help aggregate logging across nodes.

Keep an eye on the issues list for the example project to find out more about progress on these fronts. And you're free to help out too. You can also add additional pieces to the puzzle (such as Redis or Elasticsearch). Pull-requests are more than welcome if you work any of these out!

I'll leave you with the one thought that really excites me about all this. There is fascinating philosophical shift going on right now where we're turning our attention from *managing servers* to simply *running components of our app*. And this level of abstraction feels just right.

Selected references and further reading

1. Linux Containers: Parallels, LXC, OpenVZ, Docker and More
2. Borg, Omega, and Kubernetes
3. Building Scalable and Resilient Web Applications on Google Cloud Platform
4. Understanding Kubernetes from the ground up — Kubelet, API Server, Scheduler
5. Packaging Django into containers
6. Running Postgres Inside Kubernetes With Google Container Engine
7. Deploying Django with Kubernetes — Talk, Example Code
8. Deploying a containerised Rails app to Google Container Engine with Kubernetes — Part 1, Part 2, Part 3