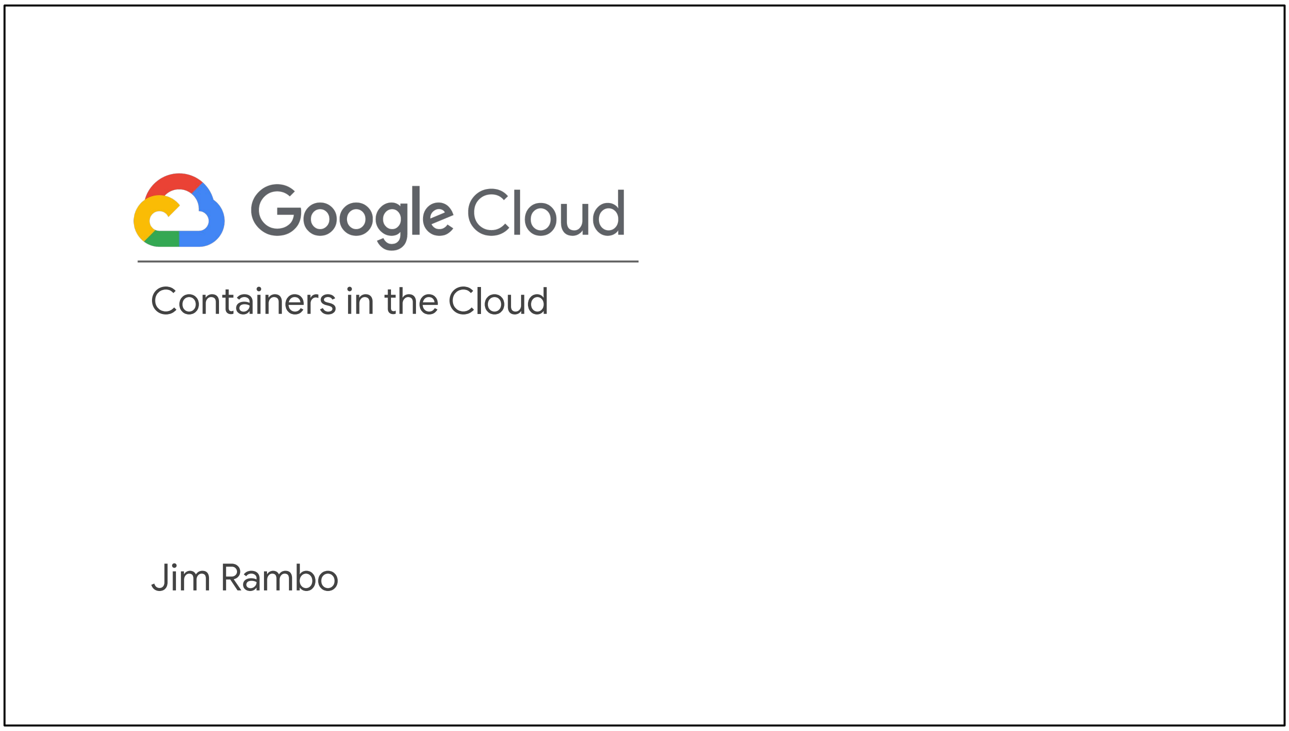
Video Name: T-GCPFCI-B_5_l1_Containers in the Cloud

Content Type: Video - Lecture Presenter

Presenter: Jim Rambo

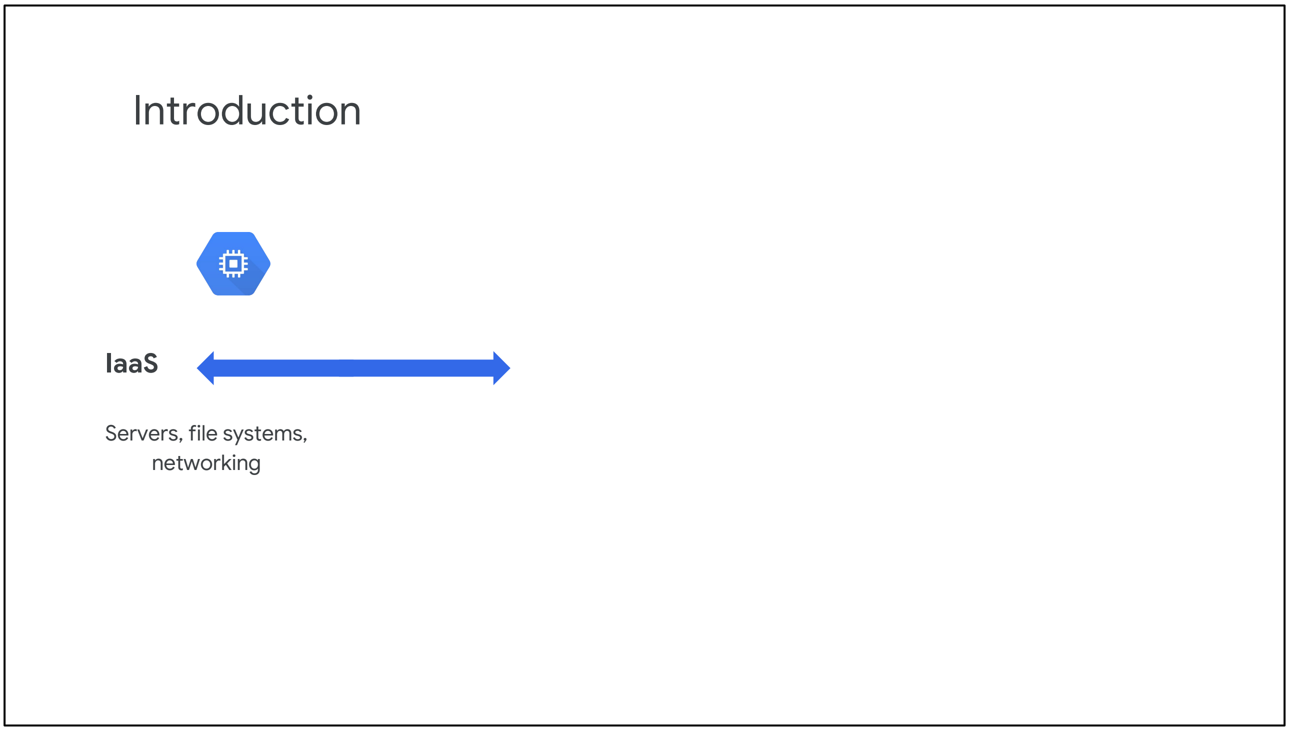


Google Cloud

Containers in the Cloud

Jim Rambo

Welcome to this module on containers and Google Kubernetes Engine.

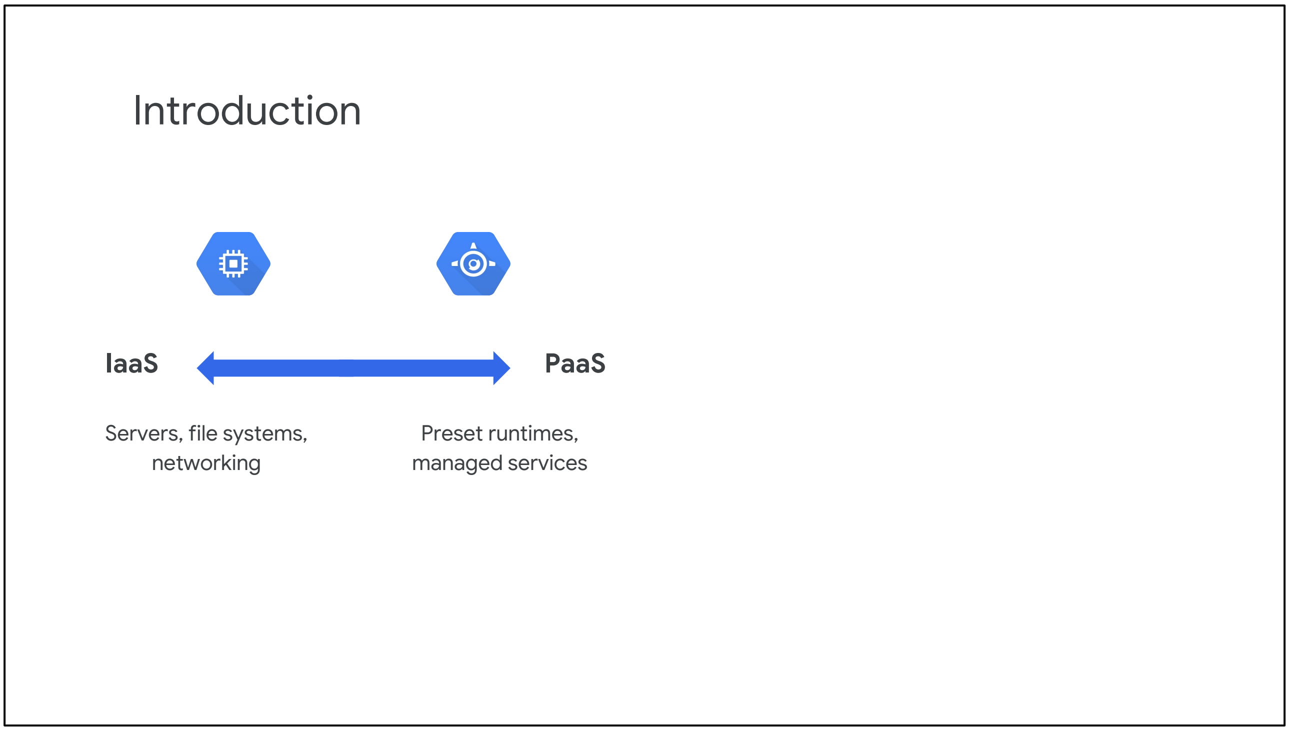


Introduction

IaaS

Servers, file systems, networking

We've already discussed **Compute Engine**, which is GCP's **Infrastructure as a Service** offering, with access to servers, file systems, and networking.



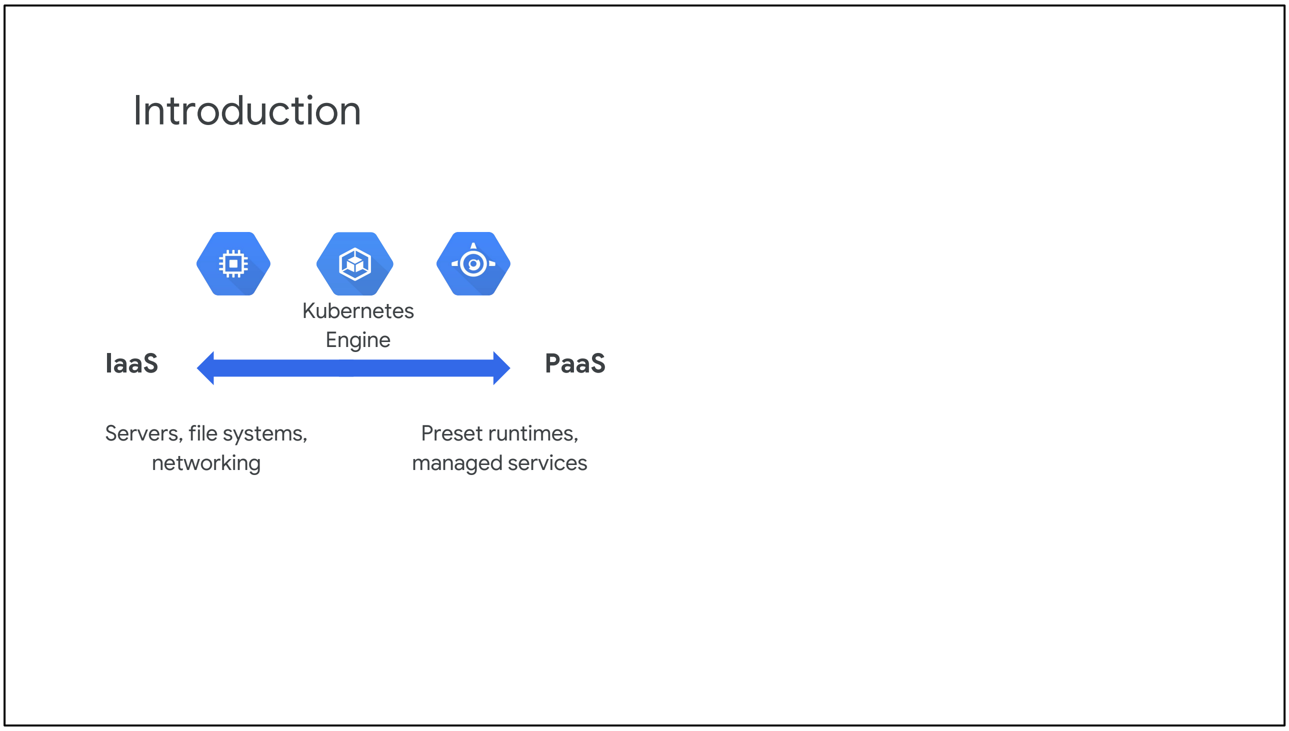
Introduction

IaaS ⟷ **PaaS**

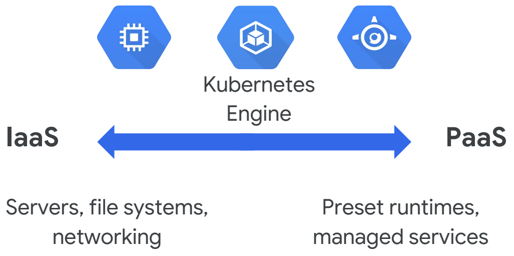
Servers, file systems, networking

Preset runtimes, managed services

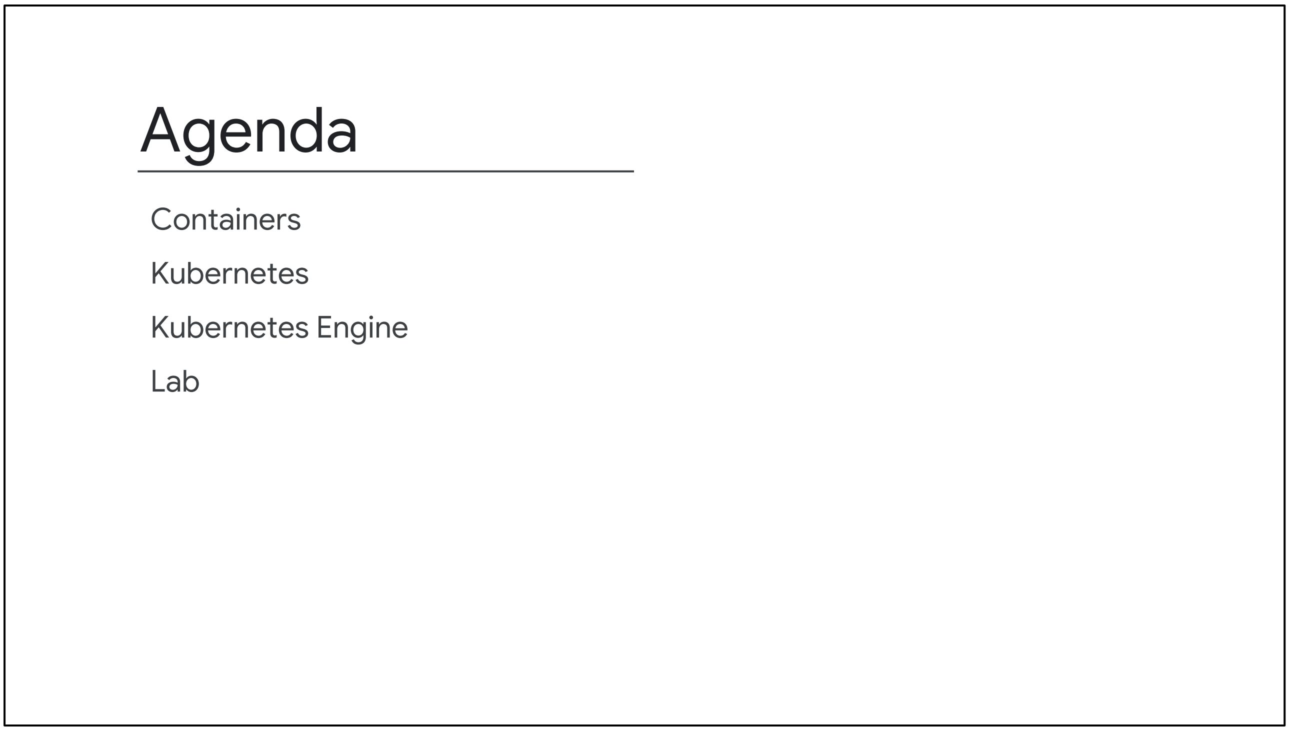
And **App Engine** which is GCP's **PaaS** offering.



Introduction


Kubernetes Engine
IaaS
PaaS
Servers, file systems, networking
Preset runtimes, managed services

Now I'm going to introduce you to containers and Kubernetes Engine which is a hybrid which conceptually sits between the two and benefits from both.



Agenda

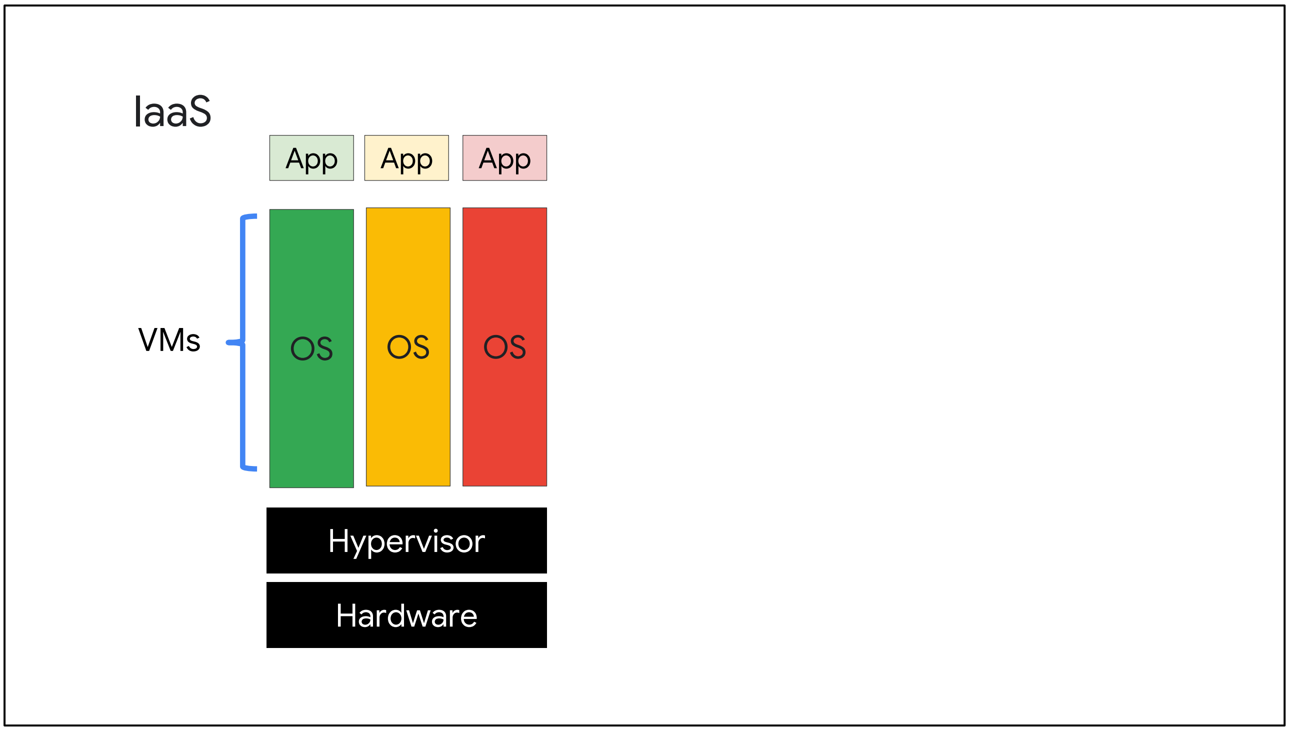
Containers

Kubernetes

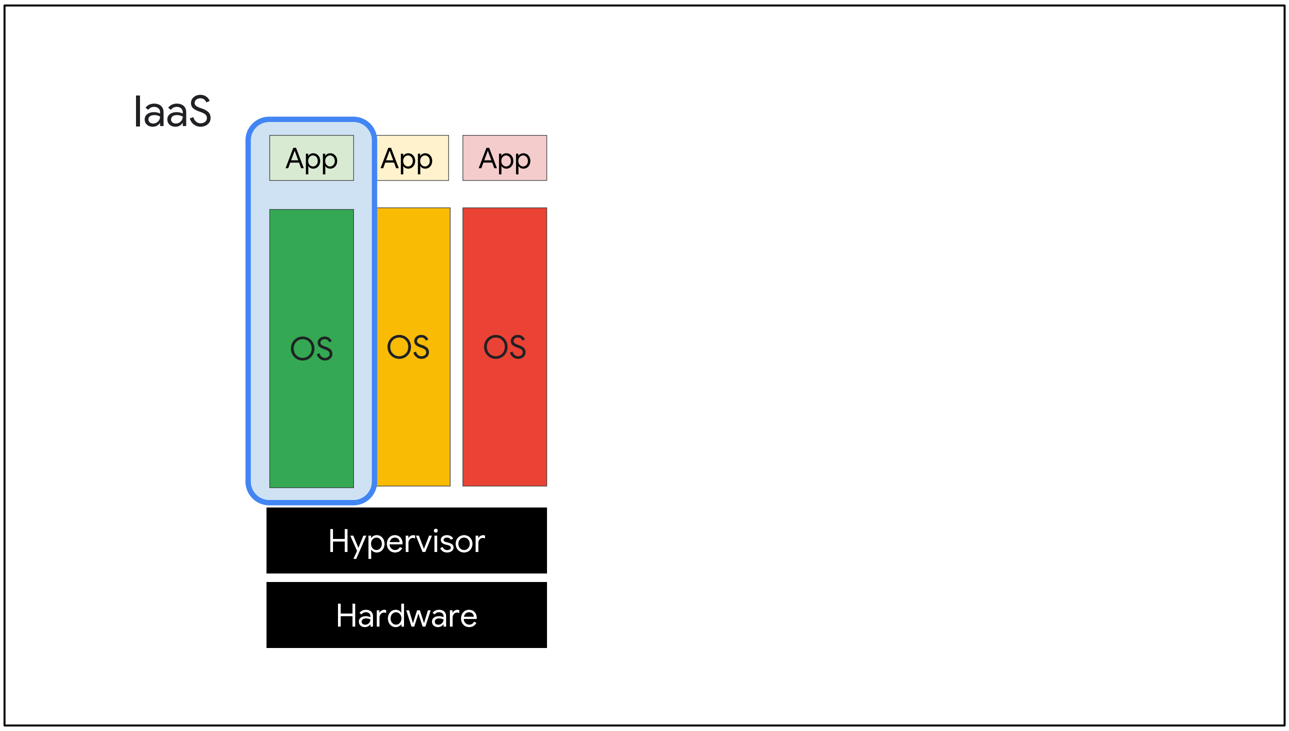
Kubernetes Engine

Lab

I'll describe why you want to use containers and how to manage them in **Kubernetes Engine**.

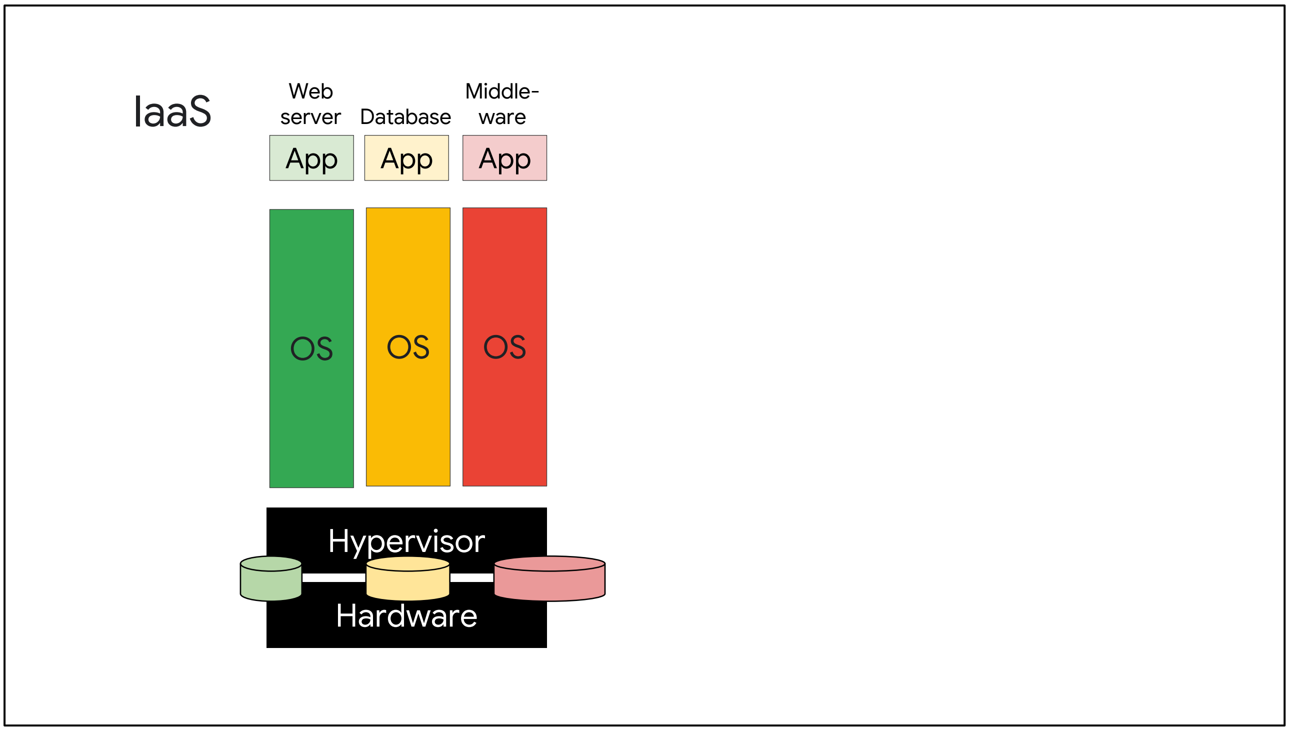


Let's begin, by remembering that **Infrastructure as a Service** allows you to share compute resources with other developers by **virtualizing the hardware** using **virtual machines**. Each developer can deploy their own operating system, access the hardware, and build their applications in a self-contained environment with access to RAM, file systems, networking interfaces, and so on.

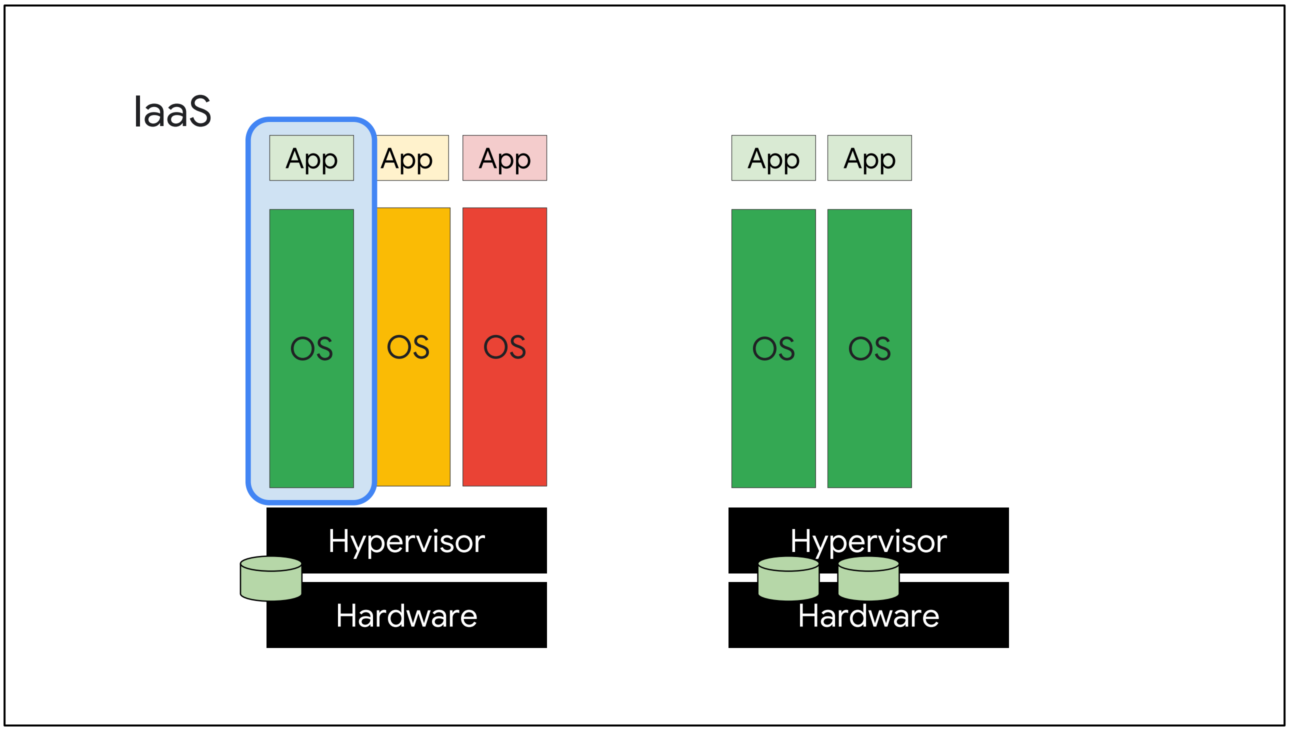


But flexibility comes with a cost. The smallest unit of compute is an app with its **VM**. The guest OS may be large, even gigabytes in size, and takes minutes to boot.

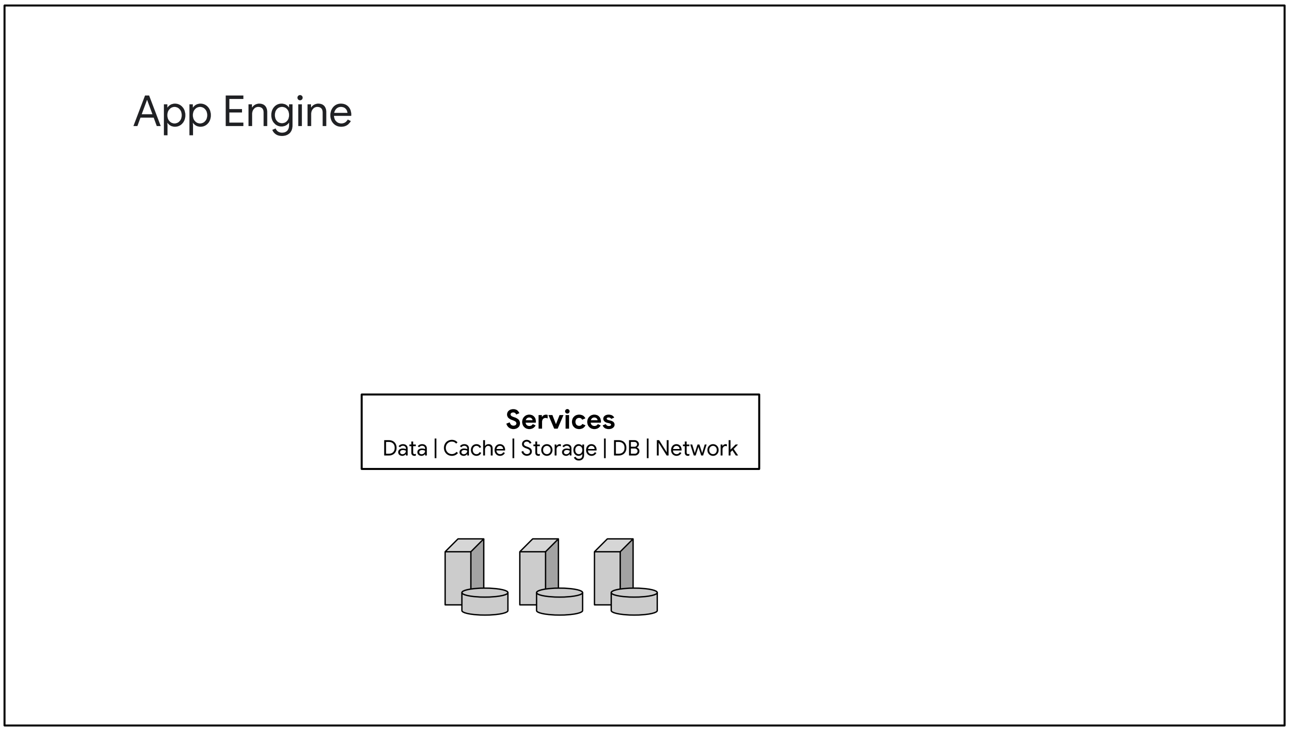
1:00



But you have your tools of choice on a configurable system. So you can install your favorite runtime, web server, database, or middleware, configure the underlying system resources, such as disk space, disk I/O, or networking and build as you like.

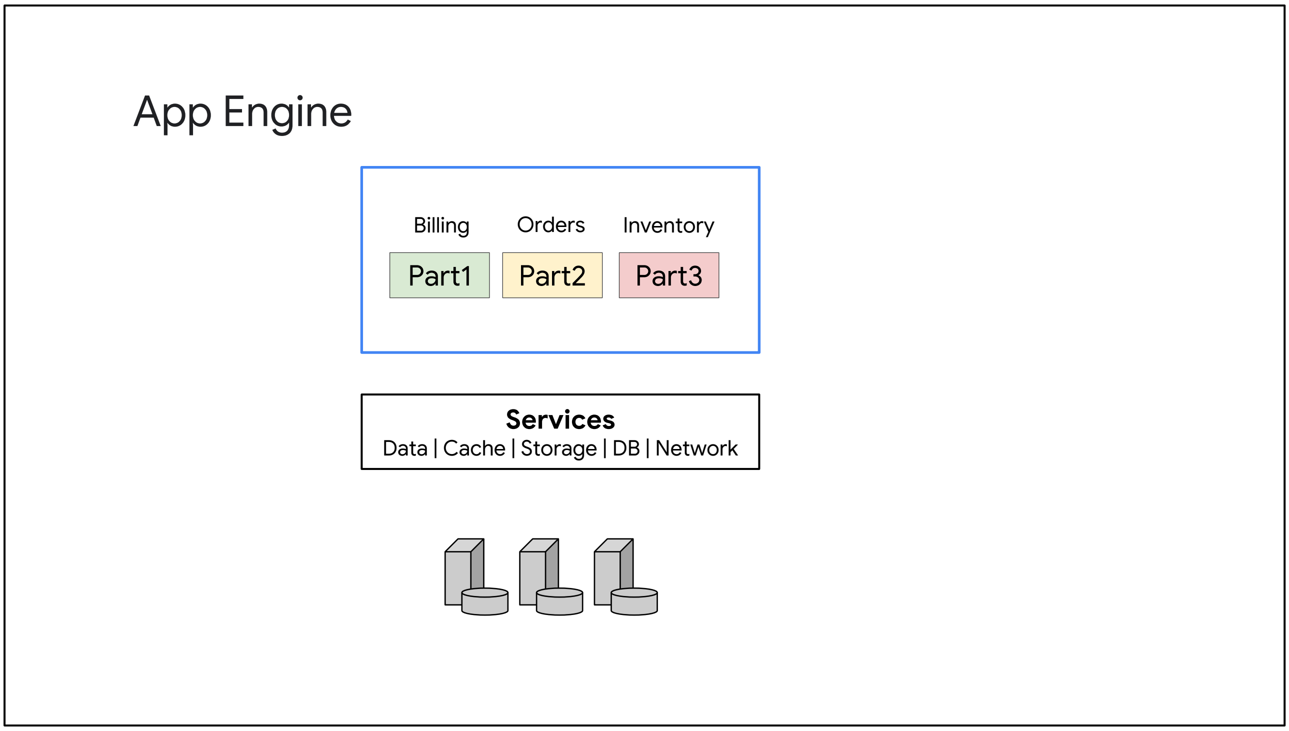


However, as demand for your application increases, you have to copy an entire VM and boot the guest OS for each instance of your app, which can be slow and costly.



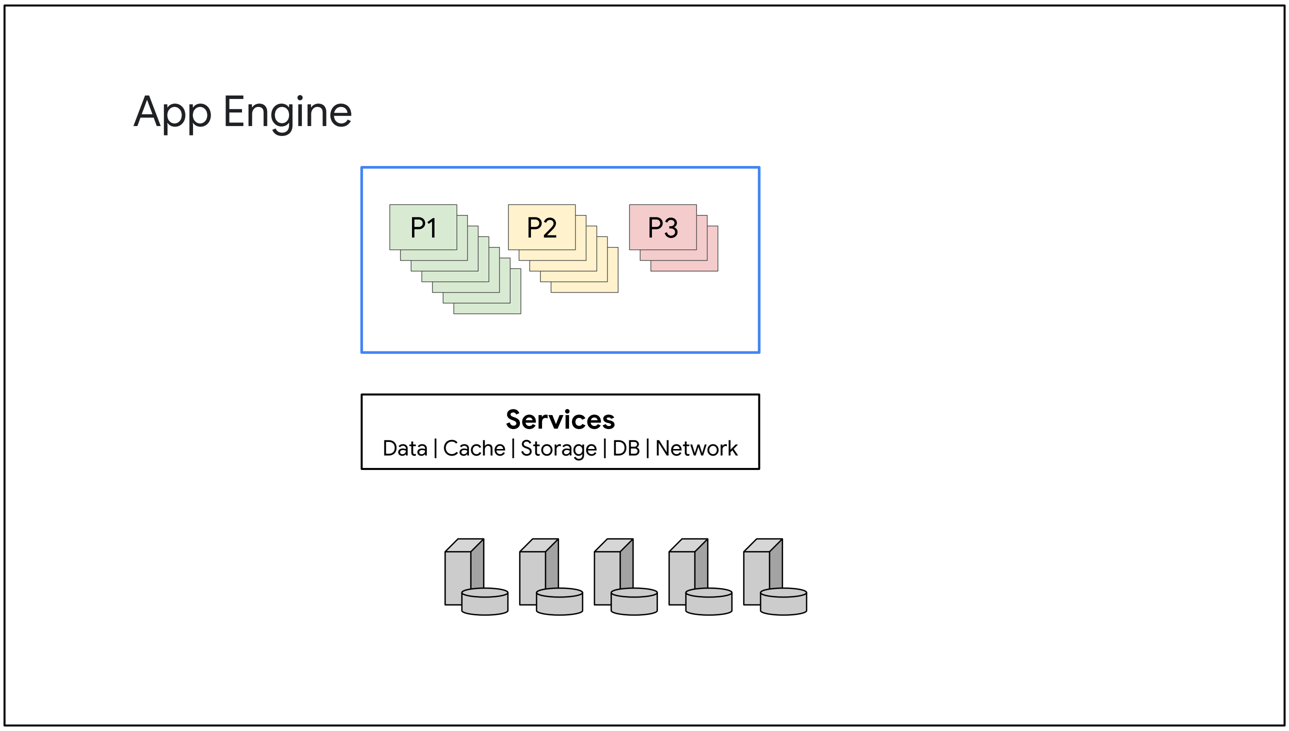
App Engine

With **App Engine** you get access to programming services.

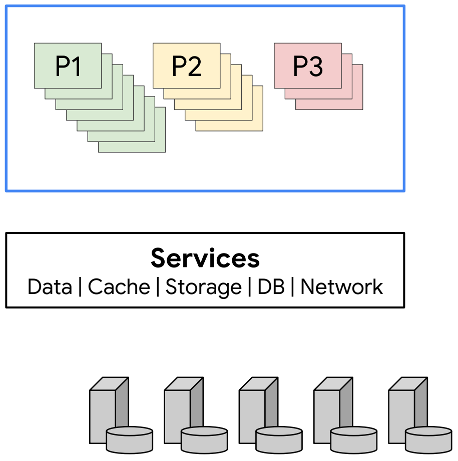


So all you do is write your code in self-contained **workloads** that use these services and include any dependent libraries.

2:00

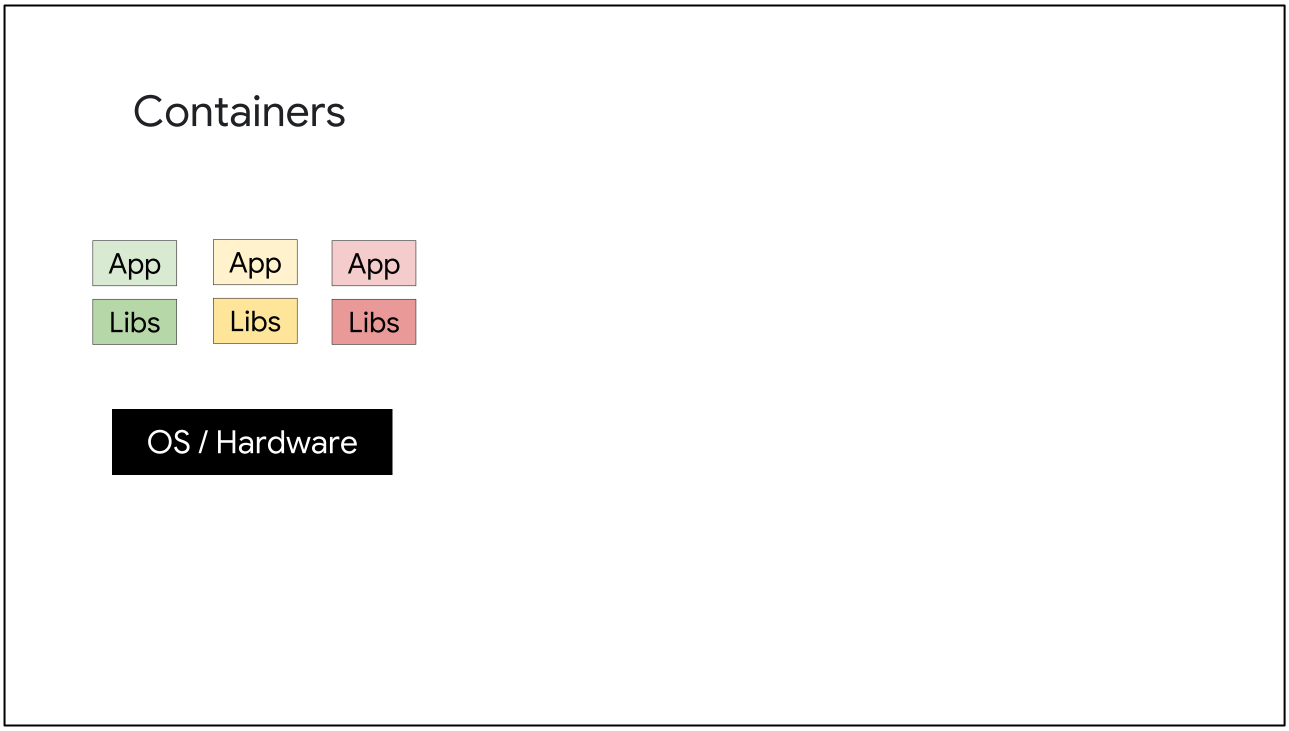


App Engine



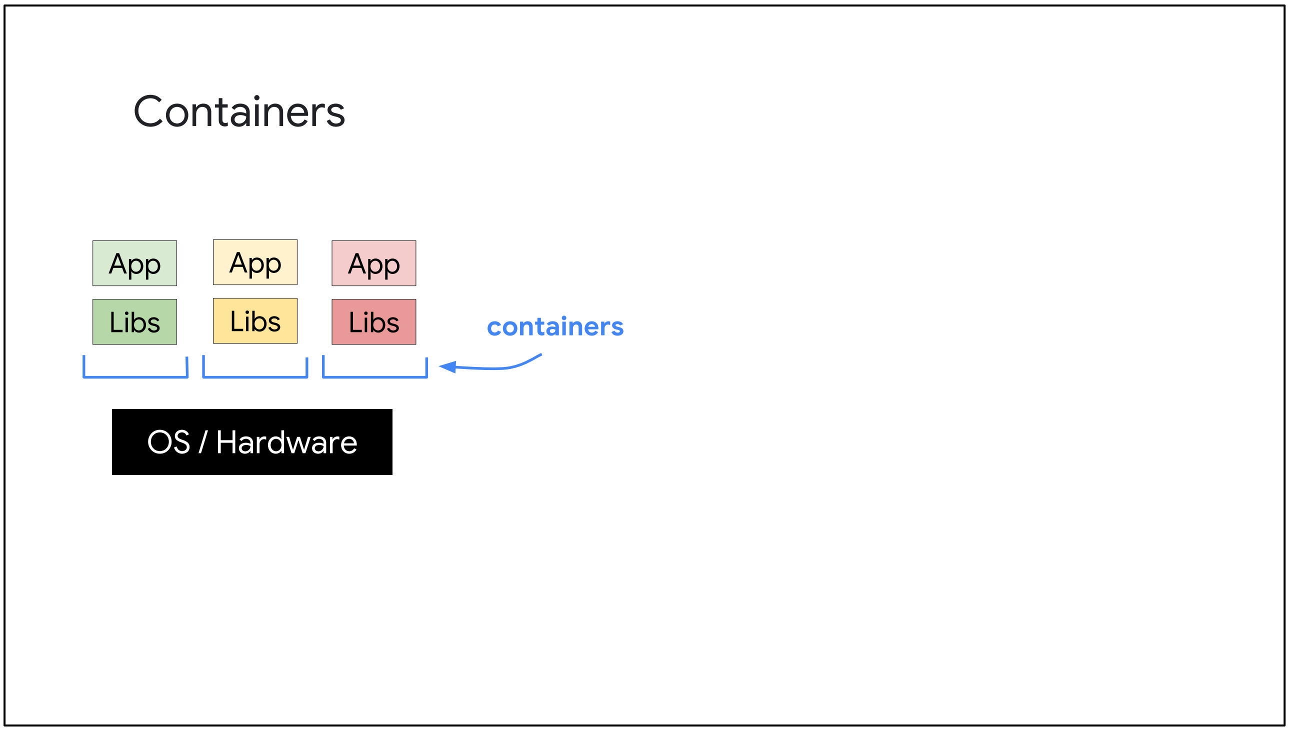
As demand for your app increases, the platform scales your app seamlessly and independently by workload and infrastructure.

This scales rapidly but you won't be able to fine-tune the underlying architecture to save cost.

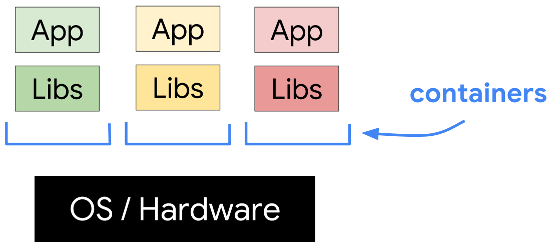

Containers

That's where containers come in.

The idea of a **container** is to give you the independent scalability of workloads in PaaS and an abstraction layer of the OS and hardware in IaaS.

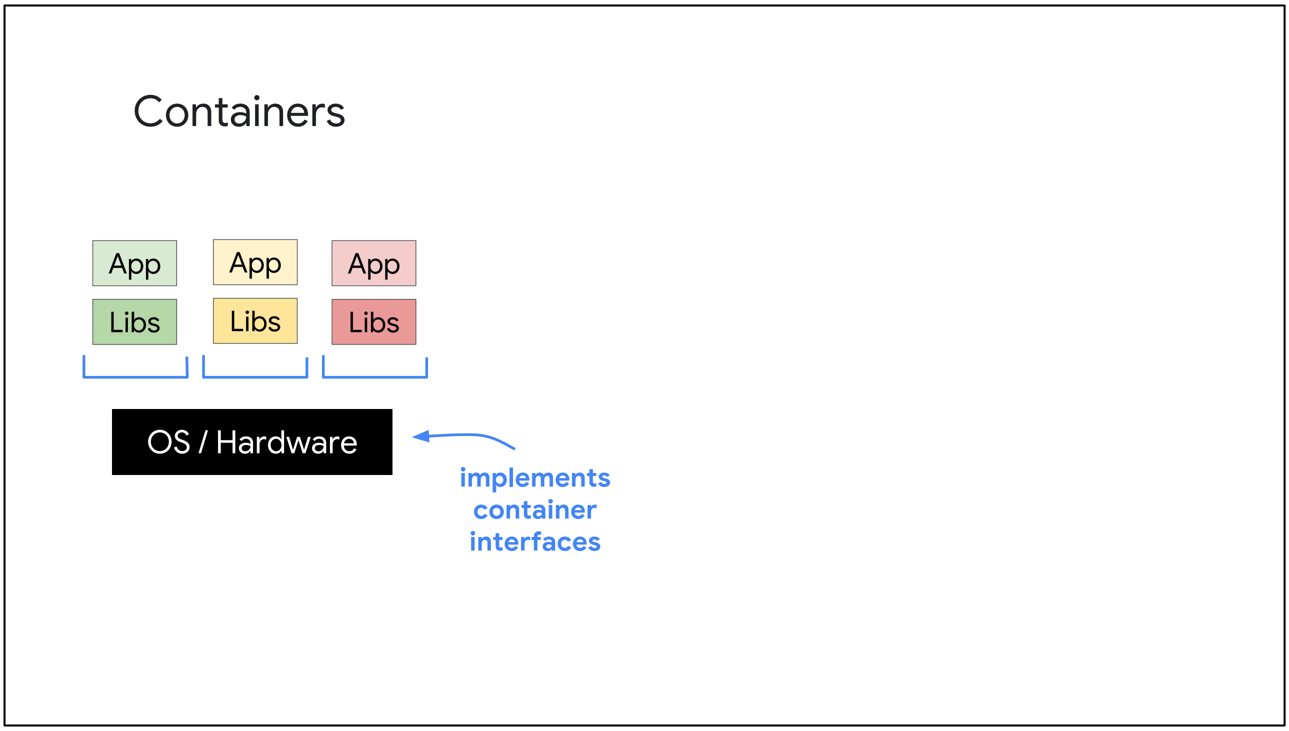


Containers



What you get is an **invisible box** around your code and its dependencies, with limited access to its own **partition** of the file system and hardware.

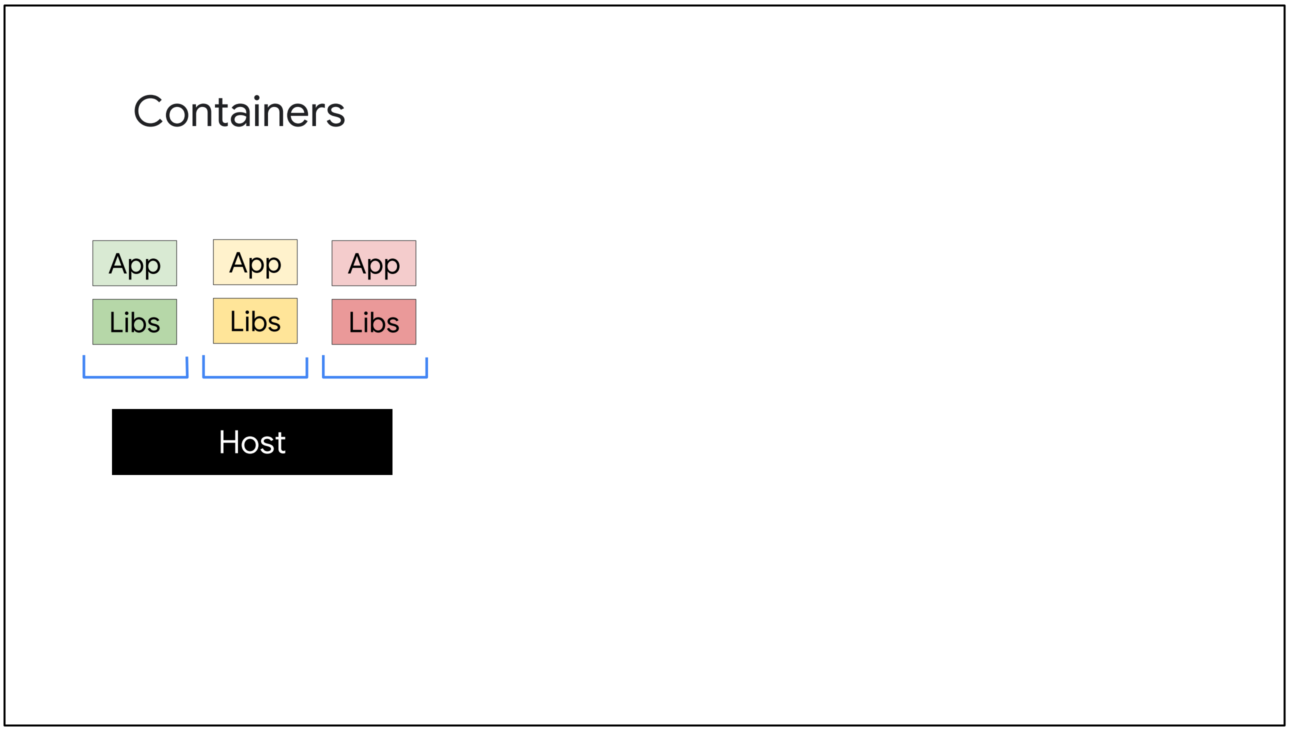
It only requires a few system calls to create and it starts as quickly as a process.



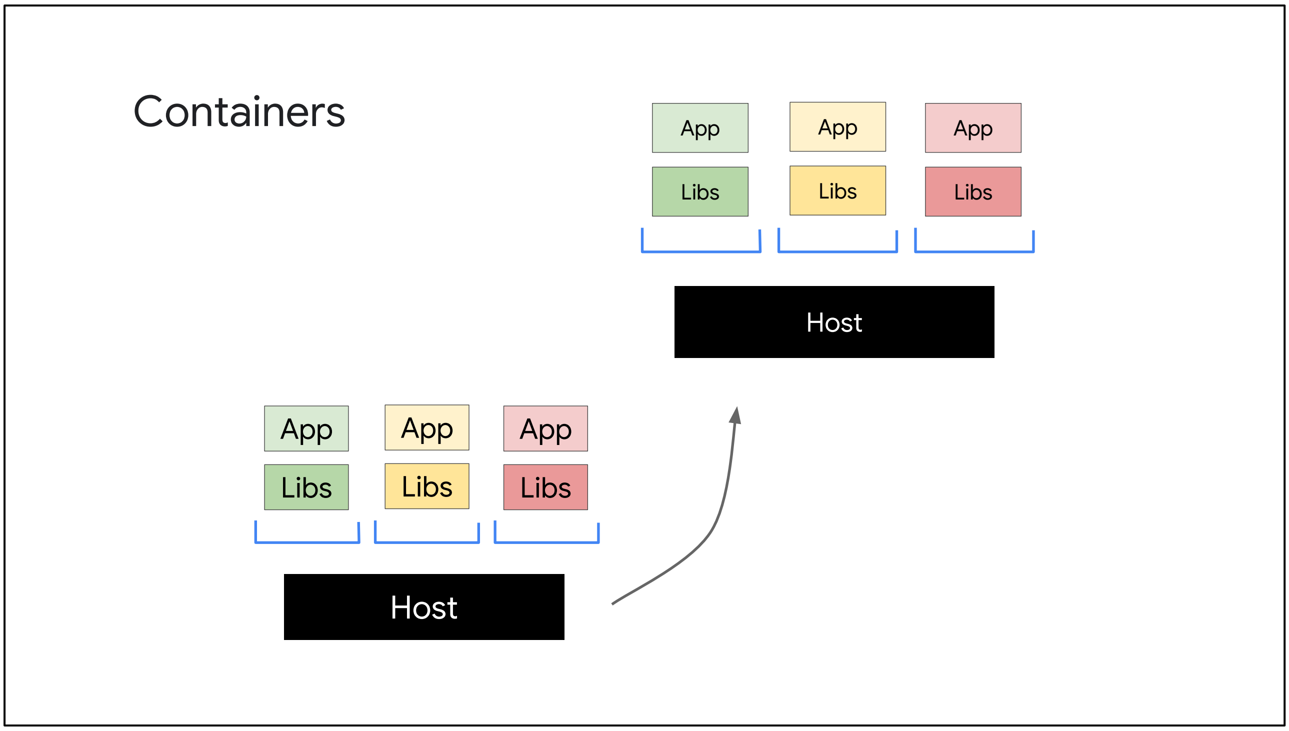
All you need on each host is an OS kernel that supports containers and a container runtime.

In essence, you are **virtualizing the OS**. It scales like PaaS, but gives you nearly the same flexibility as IaaS.

2:15

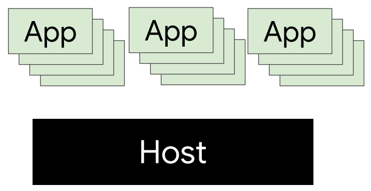


With this abstraction, your code is ultra portable and you can treat the OS and hardware as a black box.



So you can go from development, to staging, to production, or from your laptop to the cloud, without changing or rebuilding anything.

Containers

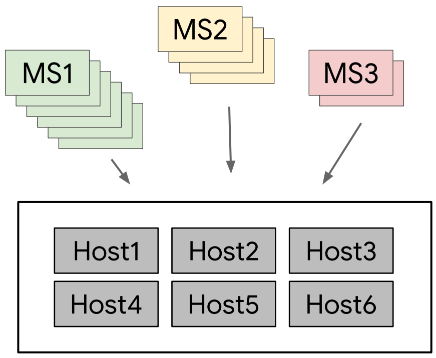


If you want to scale, for example, a web server, you can do so in seconds and deploy dozens or hundreds of them (depending on the size or your workload) on a single host.

Now that's a simple example of scaling one container running the whole application on a single host.

2:45

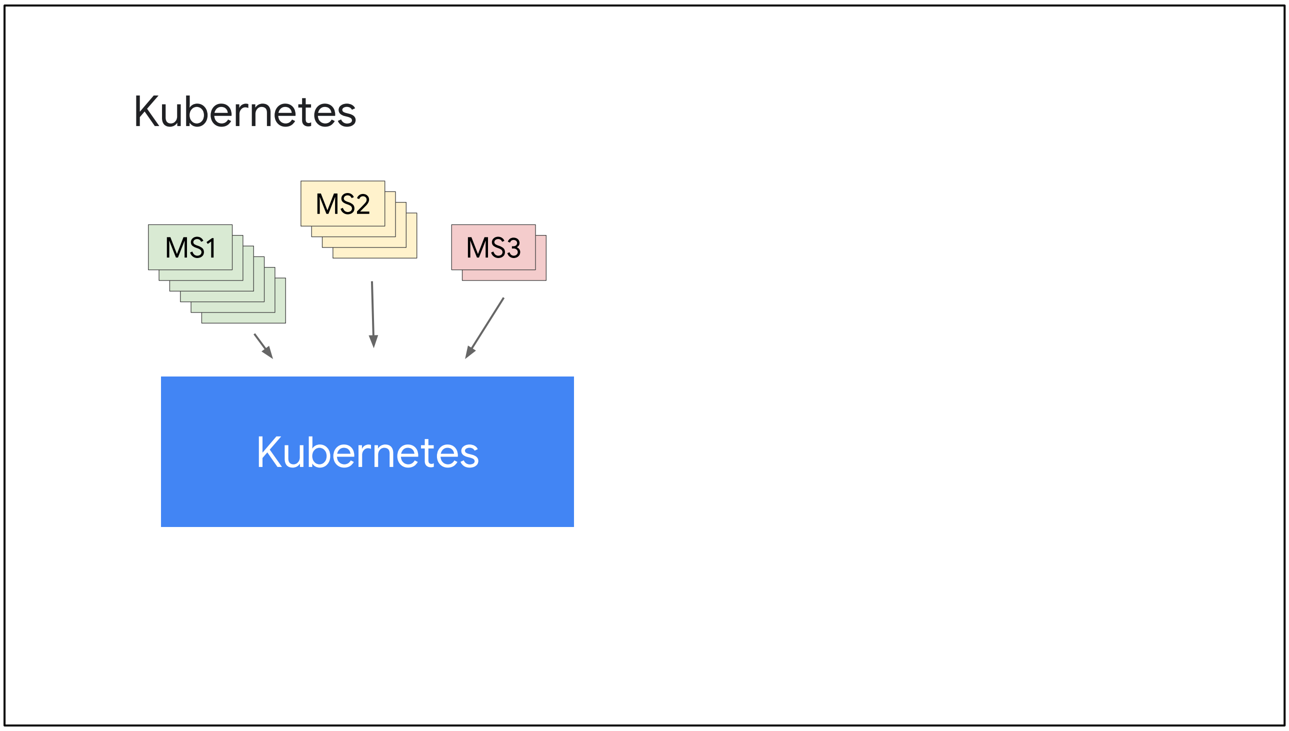
Containers



You'll likely want to build your applications using lots of containers, each performing their own function like **microservices**.

If you build them this way, and connect them with network connections, you can make them modular, deploy easily, and scale independently across a group of **hosts**.

And the hosts can scale up and down and start and stop containers as demand for your app changes or as hosts fail.



Kubernetes

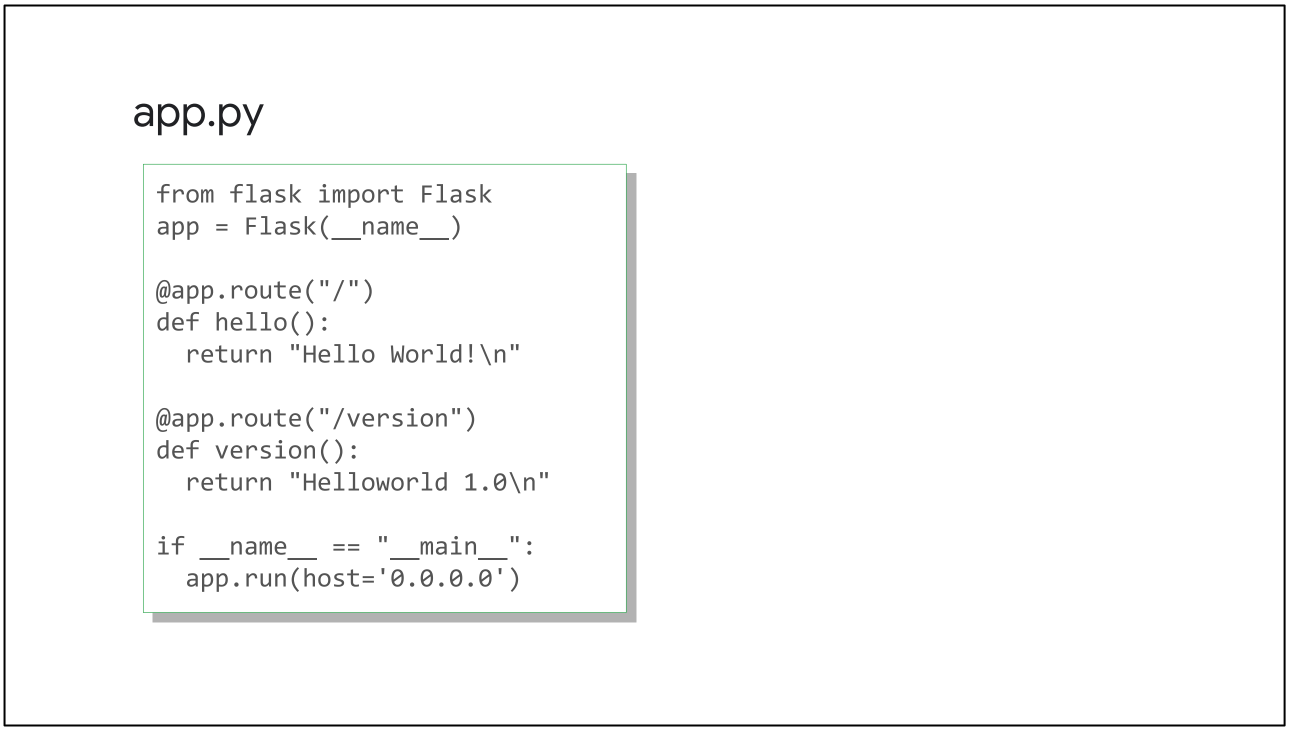
A tool that helps you do this well is **Kubernetes**.

Kubernetes makes it easy to orchestrate many containers on many hosts, scale them as microservices, and deploy rollouts and rollbacks.

First, I'll show you how you build and run containers.

I'll use an open-source tool called **Docker** that defines a format for bundling your application, its dependencies, and machine-specific settings into a container; you could use a different tool like **Google Container Builder**. It's up to you.

3:30



app.py

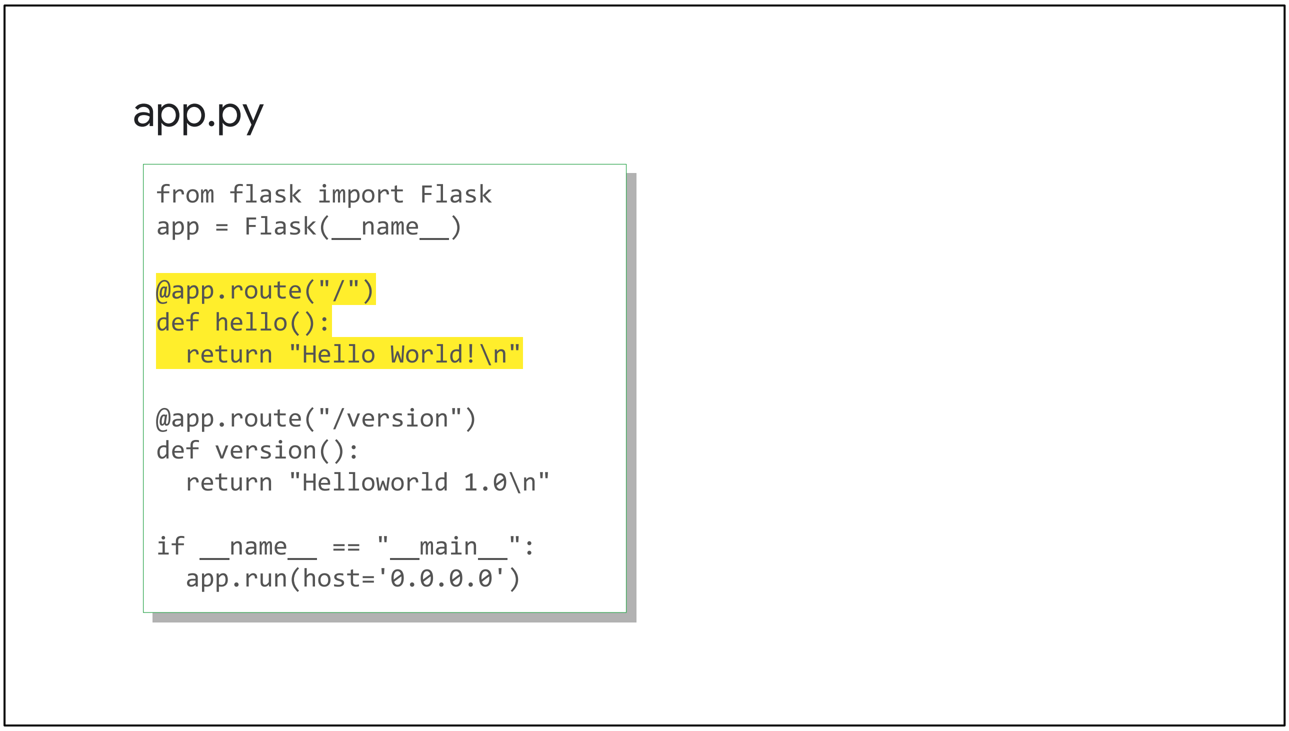
```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!\n"

@app.route("/version")
def version():
  return "Helloworld 1.0\n"

if __name__ == "__main__":
  app.run(host='0.0.0.0')
```

Here is an example of some code you may have written.

app.py

```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!\n"

@app.route("/version")
def version():
  return "Helloworld 1.0\n"

if __name__ == "__main__":
  app.run(host='0.0.0.0')
```

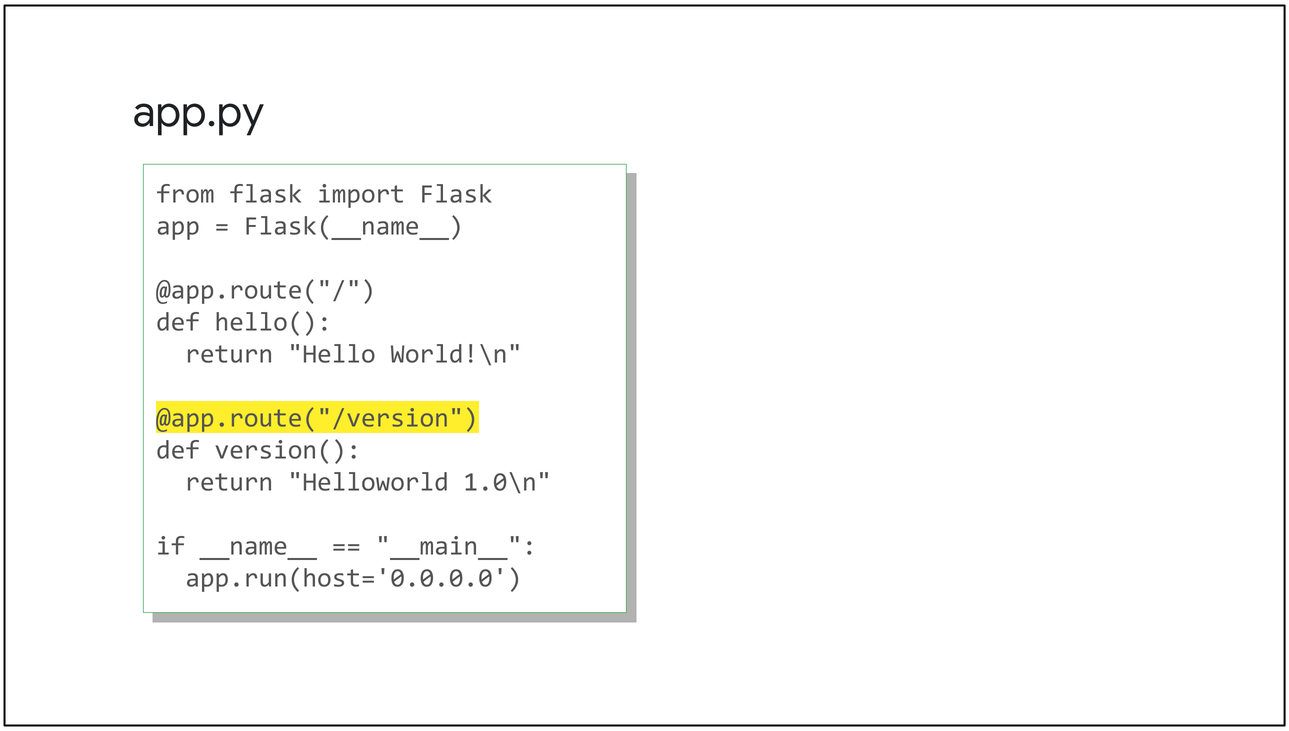
It's a Python app.



app.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!\n"

@app.route("/version")
def version():
  return "Helloworld 1.0\n"

if __name__ == "__main__":
  app.run(host='0.0.0.0')
```

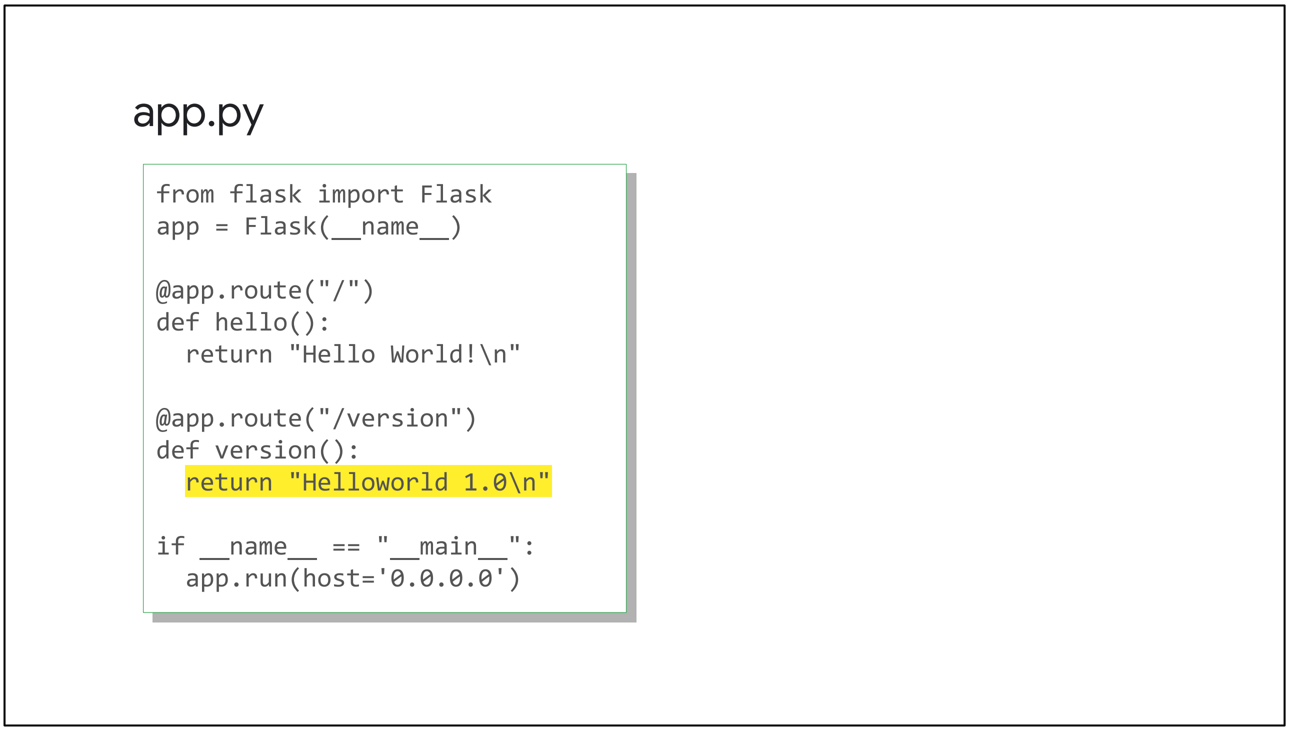
It says "Hello World"



app.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!\n"

@app.route("/version")
def version():
  return "Helloworld 1.0\n"

if __name__ == "__main__":
  app.run(host='0.0.0.0')
```

Or if you hit this endpoint,


app.py

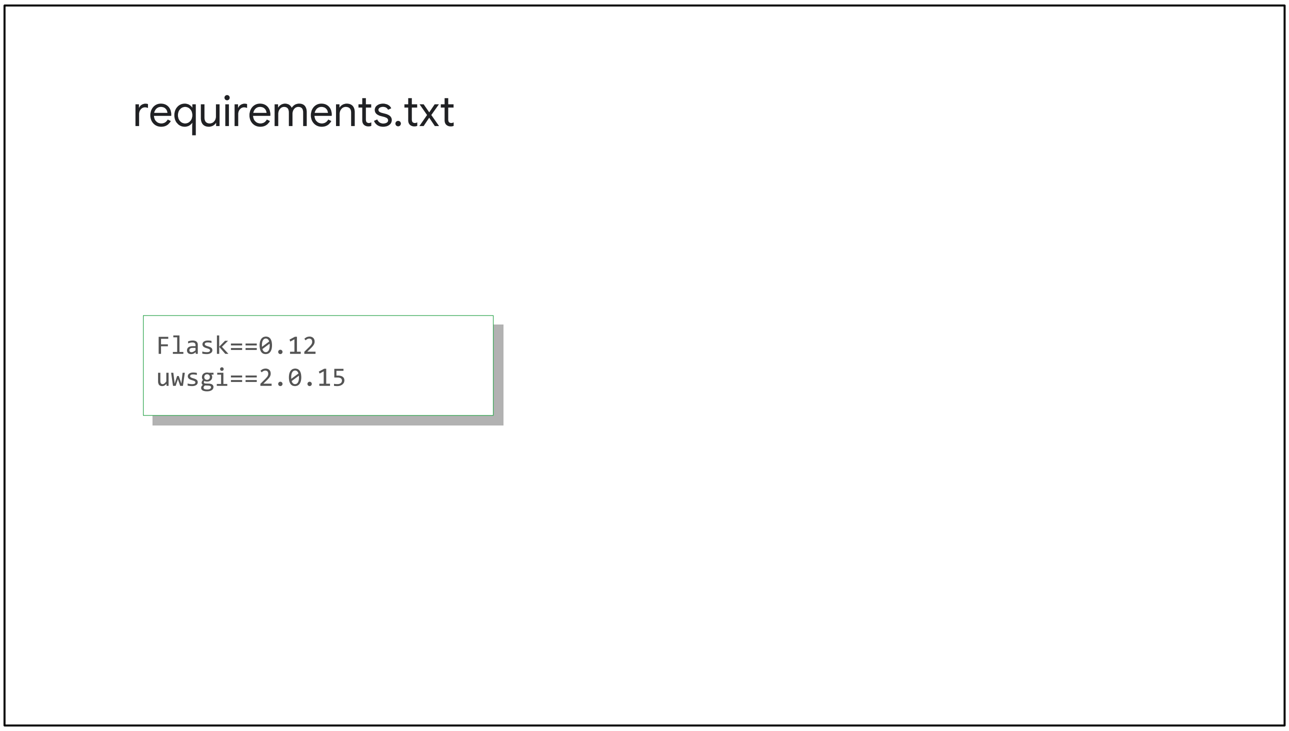
```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!\n"

@app.route("/version")
def version():
  return "Helloworld 1.0\n"

if __name__ == "__main__":
  app.run(host='0.0.0.0')
```

it gives you the version.

So how do you get this app into Kubernetes?

You have to think about your version of Python, what dependency you have on Flask,

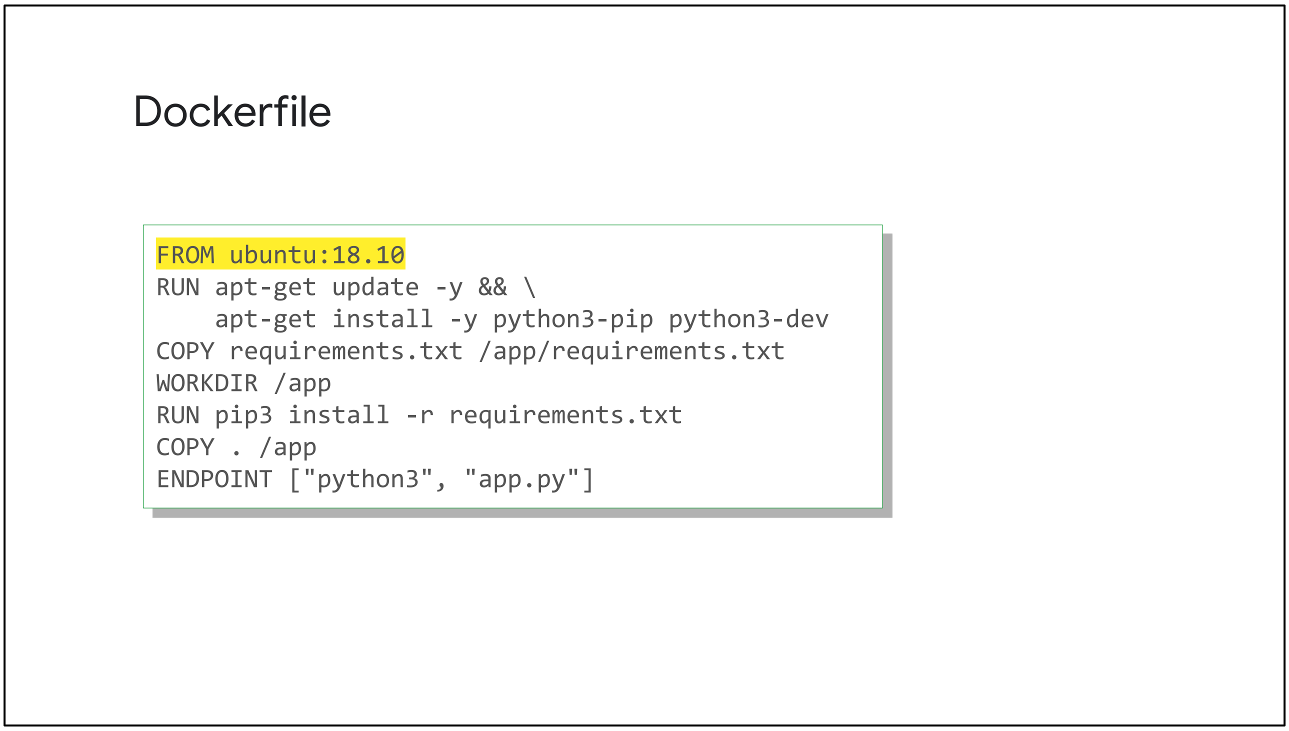
3:45



requirements.txt

```
Flask==0.12
uwsgi==2.0.15
```

how to use the requirements.txt file, how to install Python, and so on.

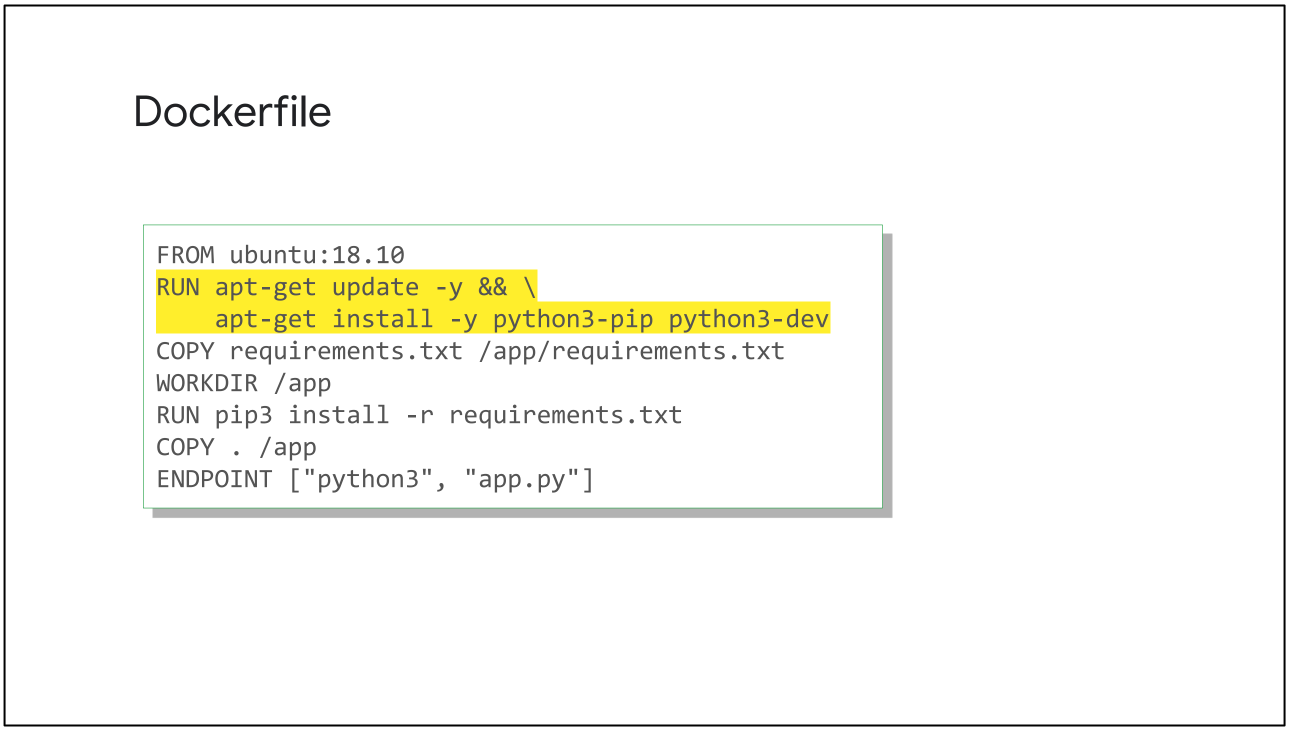


Dockerfile

```
FROM ubuntu:18.10
RUN apt-get update -y && \
    apt-get install -y python3-pip python3-dev
COPY requirements.txt /app/requirements.txt
WORKDIR /app
RUN pip3 install -r requirements.txt
COPY . /app
ENDPOINT ["python3", "app.py"]
```

So you use a **Dockerfile** to specify how your code gets packaged into a container.

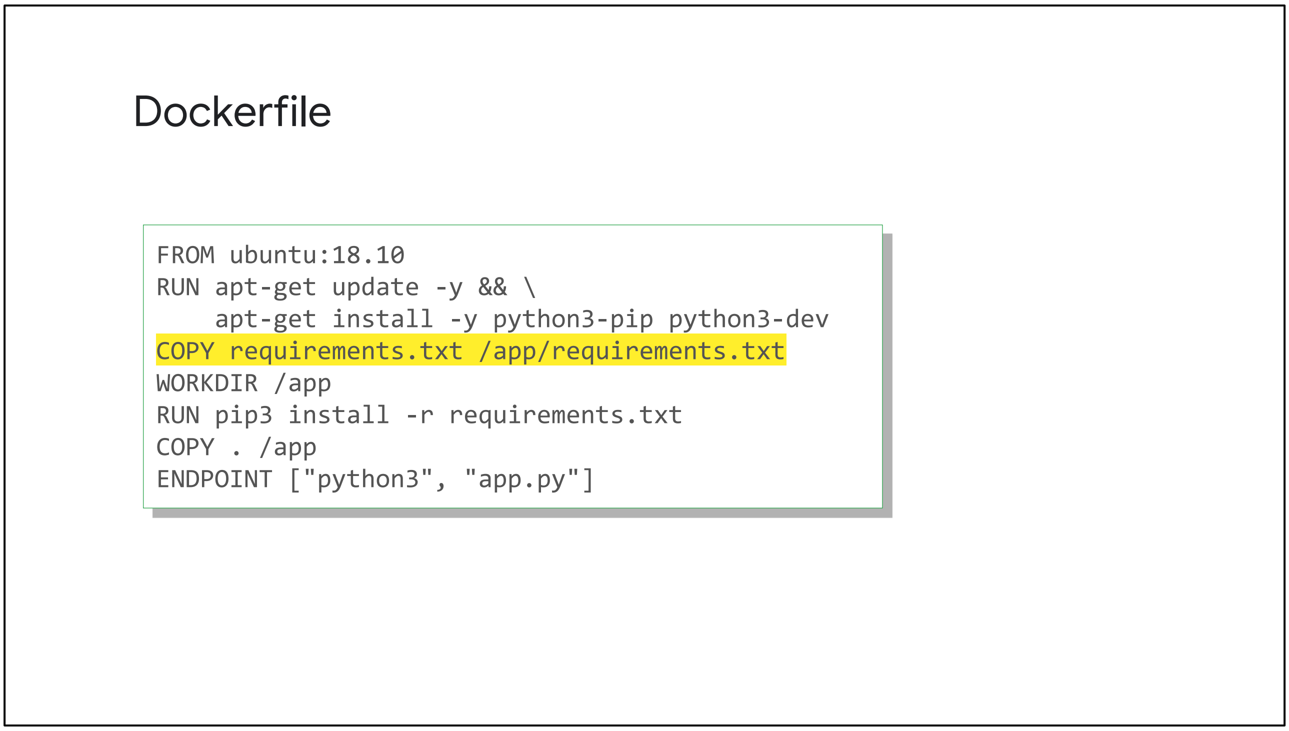
For example, if you're a developer, and you're used to using Ubuntu with all your tools, you start there.



Dockerfile

```
FROM ubuntu:18.10
RUN apt-get update -y && \
    apt-get install -y python3-pip python3-dev
COPY requirements.txt /app/requirements.txt
WORKDIR /app
RUN pip3 install -r requirements.txt
COPY . /app
ENDPOINT ["python3", "app.py"]
```

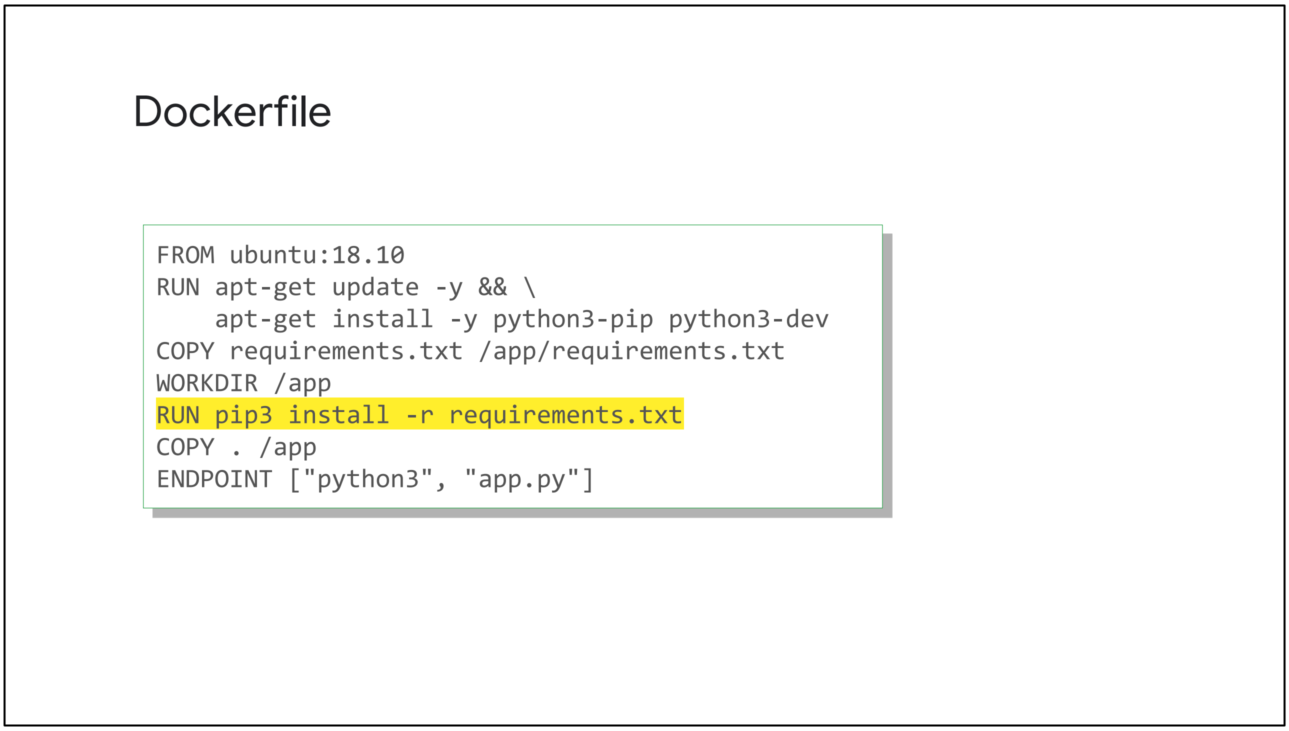
You can install Python the same way you would on your dev environment.



Dockerfile

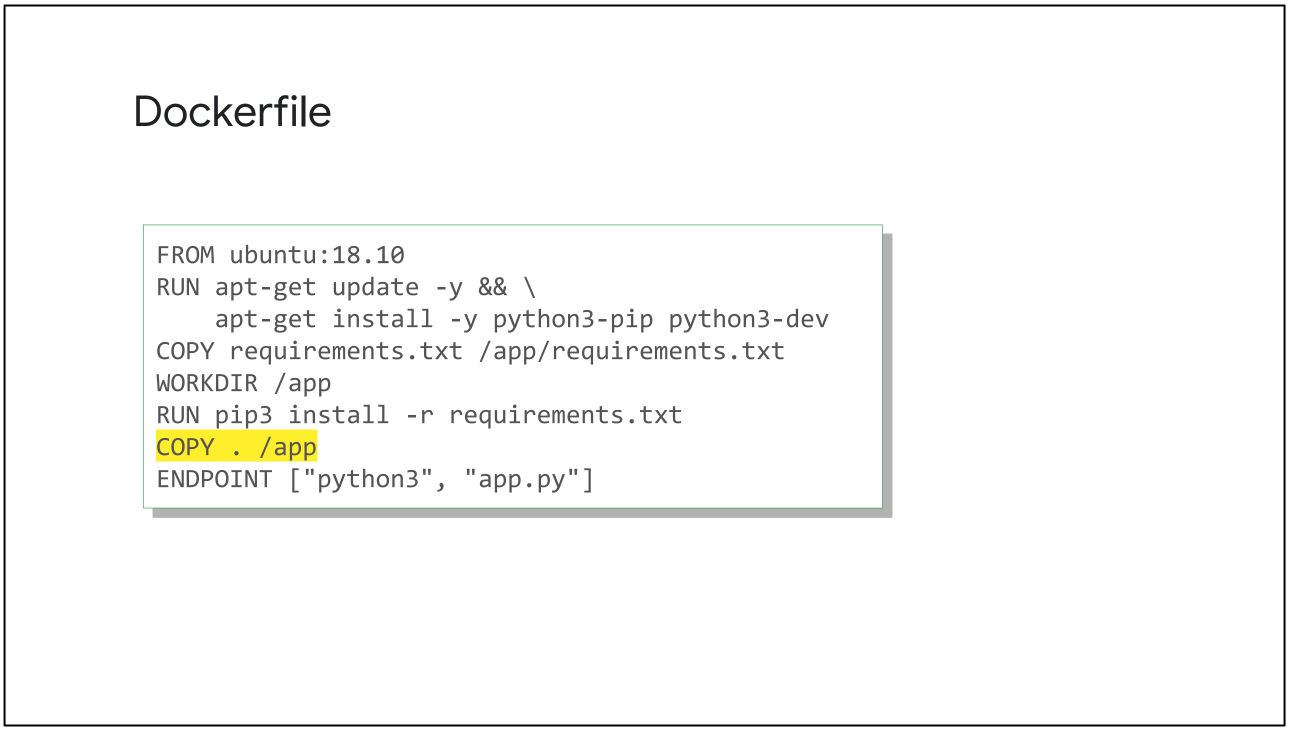
```
FROM ubuntu:18.10
RUN apt-get update -y && \
    apt-get install -y python3-pip python3-dev
COPY requirements.txt /app/requirements.txt
WORKDIR /app
RUN pip3 install -r requirements.txt
COPY . /app
ENDPOINT ["python3", "app.py"]
```

You can take that requirements file from Python that you know.


Dockerfile

```
FROM ubuntu:18.10
RUN apt-get update -y && \
    apt-get install -y python3-pip python3-dev
COPY requirements.txt /app/requirements.txt
WORKDIR /app
RUN pip3 install -r requirements.txt
COPY . /app
ENDPOINT ["python3", "app.py"]
```

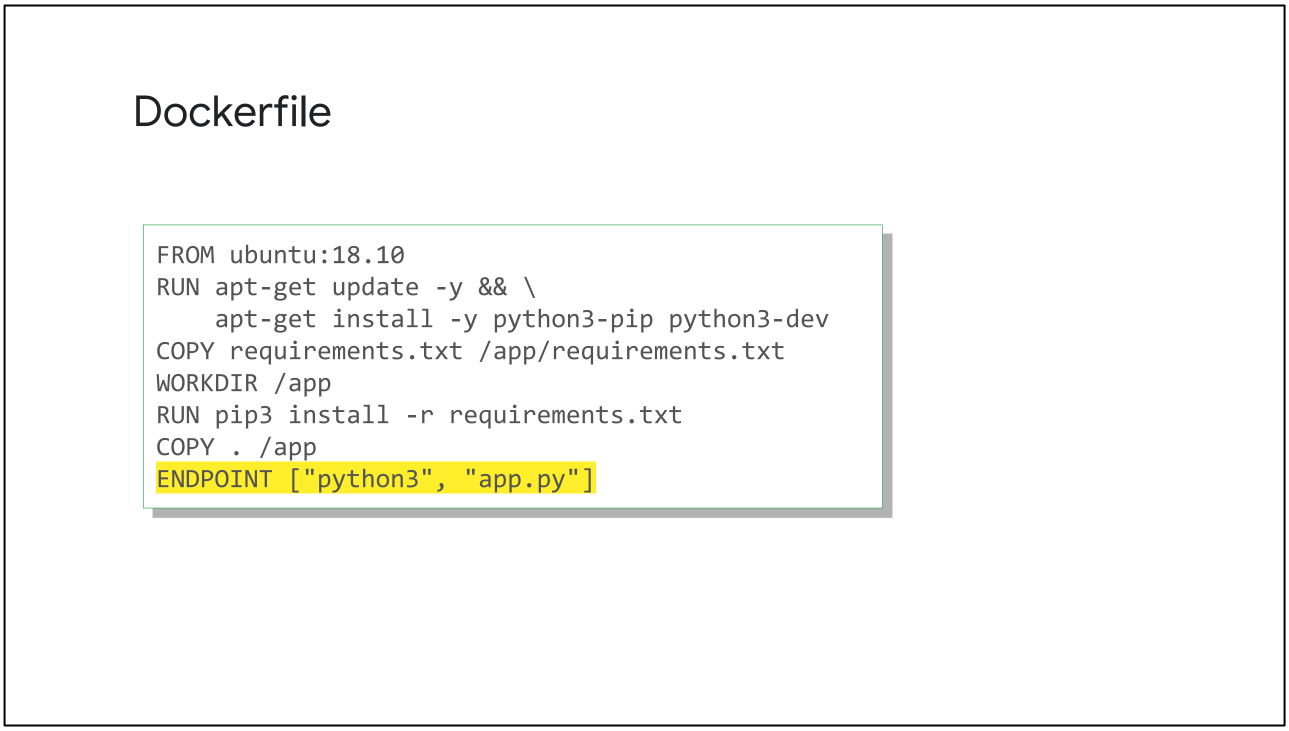
And you can use tools inside **Docker** or **Container Builder** to install your dependencies the way you want.



Dockerfile

```
FROM ubuntu:18.10
RUN apt-get update -y && \
    apt-get install -y python3-pip python3-dev
COPY requirements.txt /app/requirements.txt
WORKDIR /app
RUN pip3 install -r requirements.txt
COPY . /app
ENDPOINT ["python3", "app.py"]
```

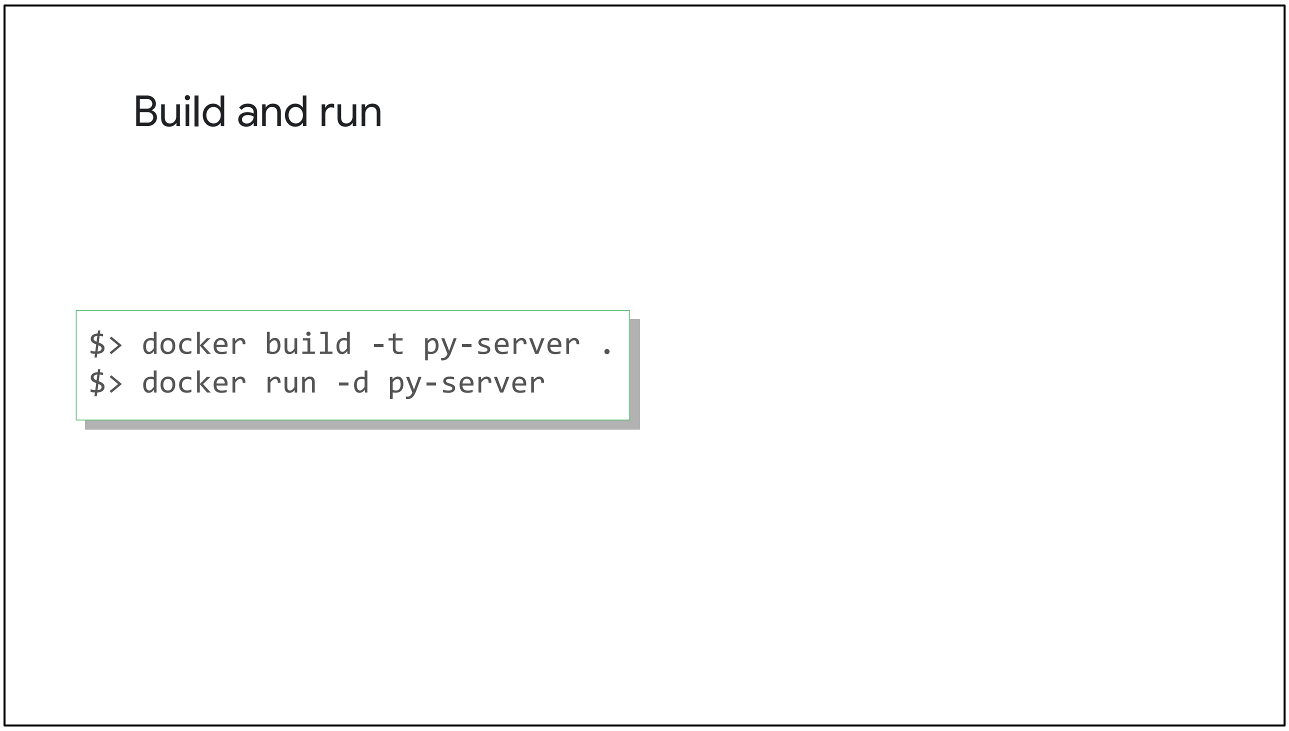
Eventually, it produces an app,



Dockerfile

```
FROM ubuntu:18.10
RUN apt-get update -y && \
    apt-get install -y python3-pip python3-dev
COPY requirements.txt /app/requirements.txt
WORKDIR /app
RUN pip3 install -r requirements.txt
COPY . /app
ENDPOINT ["python3", "app.py"]
```

and here's how you run it.



Build and run

```
$> docker build -t py-server .
$> docker run -d py-server
```

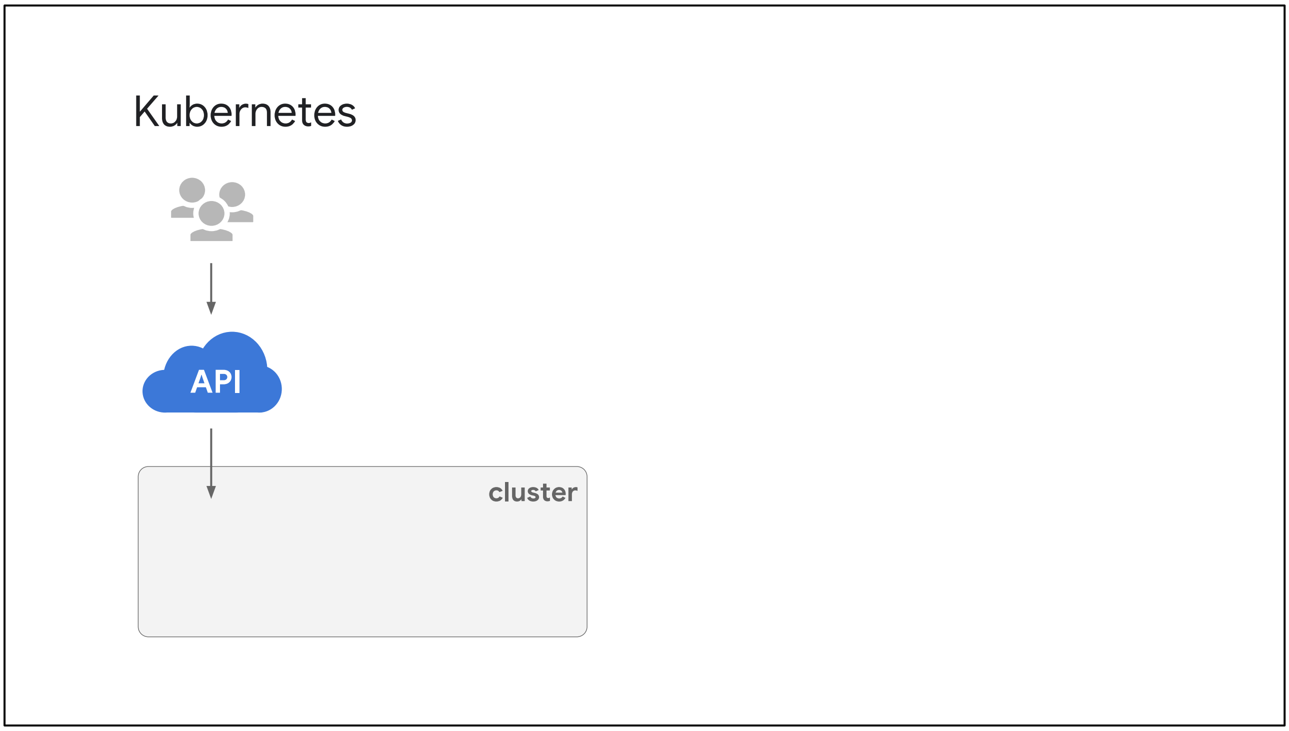
Then you use the "**docker build**" command to build the container.

This builds the container and stores it locally as a runnable **image**. You can save and upload the image to a container registry service and share or download it from there.

Then you use the "**docker run**" command to run the image.

As it turns out, packaging applications is only about 5% of the issue. The rest has to do with: application configuration, service discovery, managing updates, and monitoring. These are the components of a reliable, scalable, distributed system.

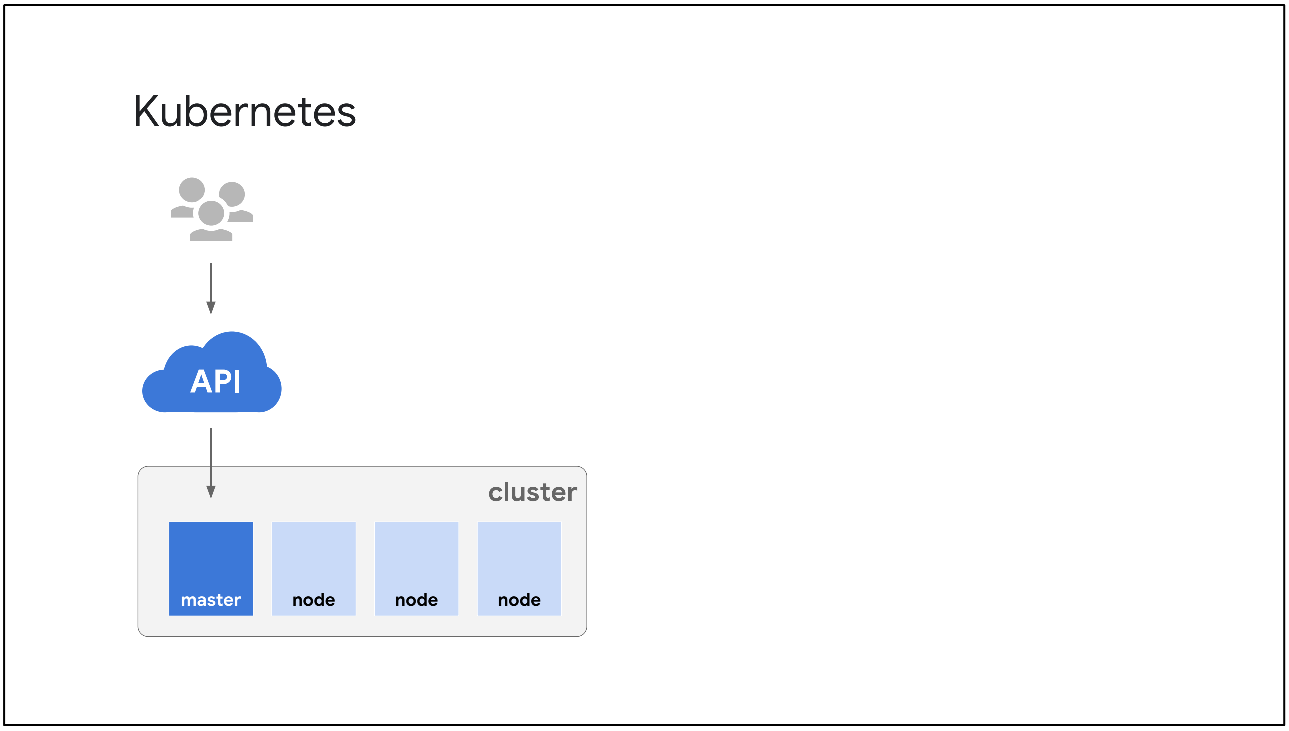
4:45



Now, I'll show you where **Kubernetes** comes in.

Kubernetes is an open-source **orchestrator** that abstracts containers at a higher level so you can better manage and scale your applications.

At the highest level, Kubernetes is a set of APIs that you can use to deploy containers on a set of **nodes** called a **cluster**.

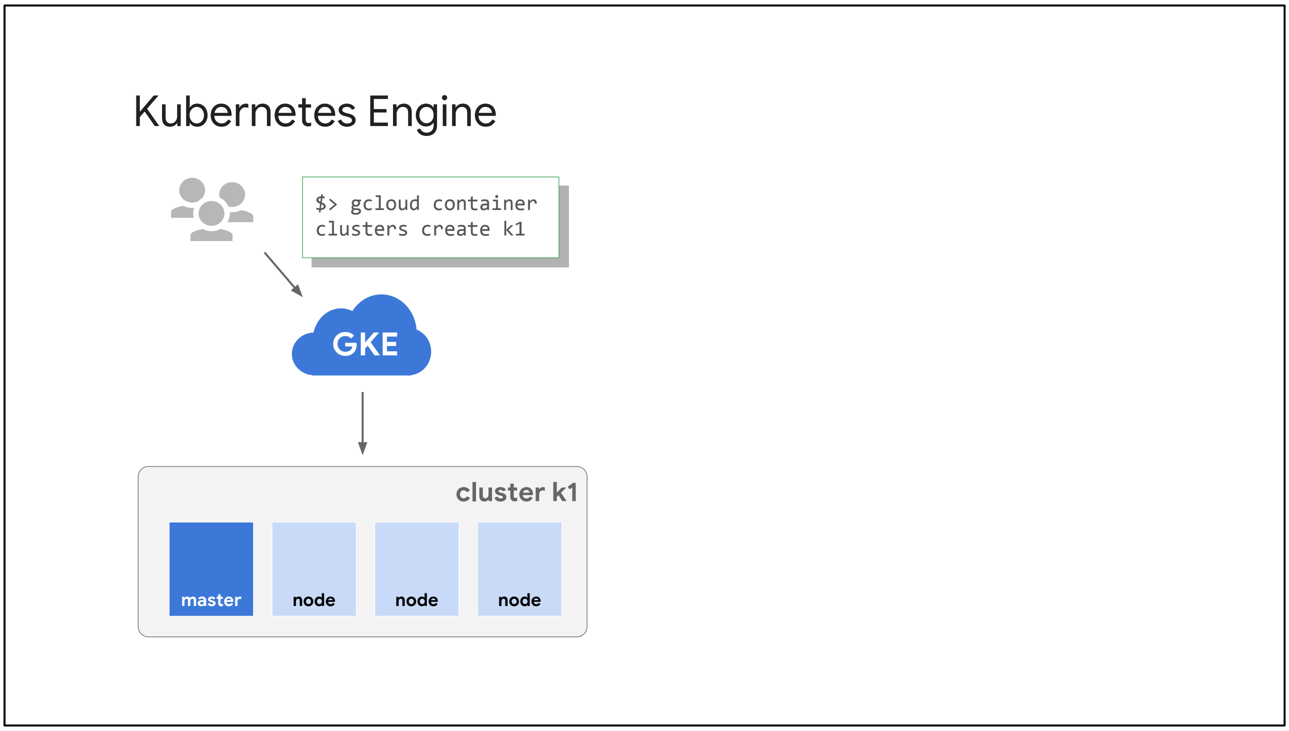


The system is divided into a set of **master** components that run as the control plane and a set of **nodes** that run containers. In Kubernetes, a node represents a computing instance, like a machine. In Google Cloud, nodes are virtual machines running in Compute Engine.

You can describe a set of applications and how they should interact with each other and Kubernetes figures how to make that happen

Now that you've built a container, you'll want to deploy one into a **cluster**.

5:30



Kubernetes Engine

```
$> gcloud container
clusters create k1
```

Kubernetes can be configured with many options and add-ons, but can be time consuming to bootstrap from the ground up. Instead, you can bootstrap Kubernetes using **Kubernetes Engine** or (GKE).

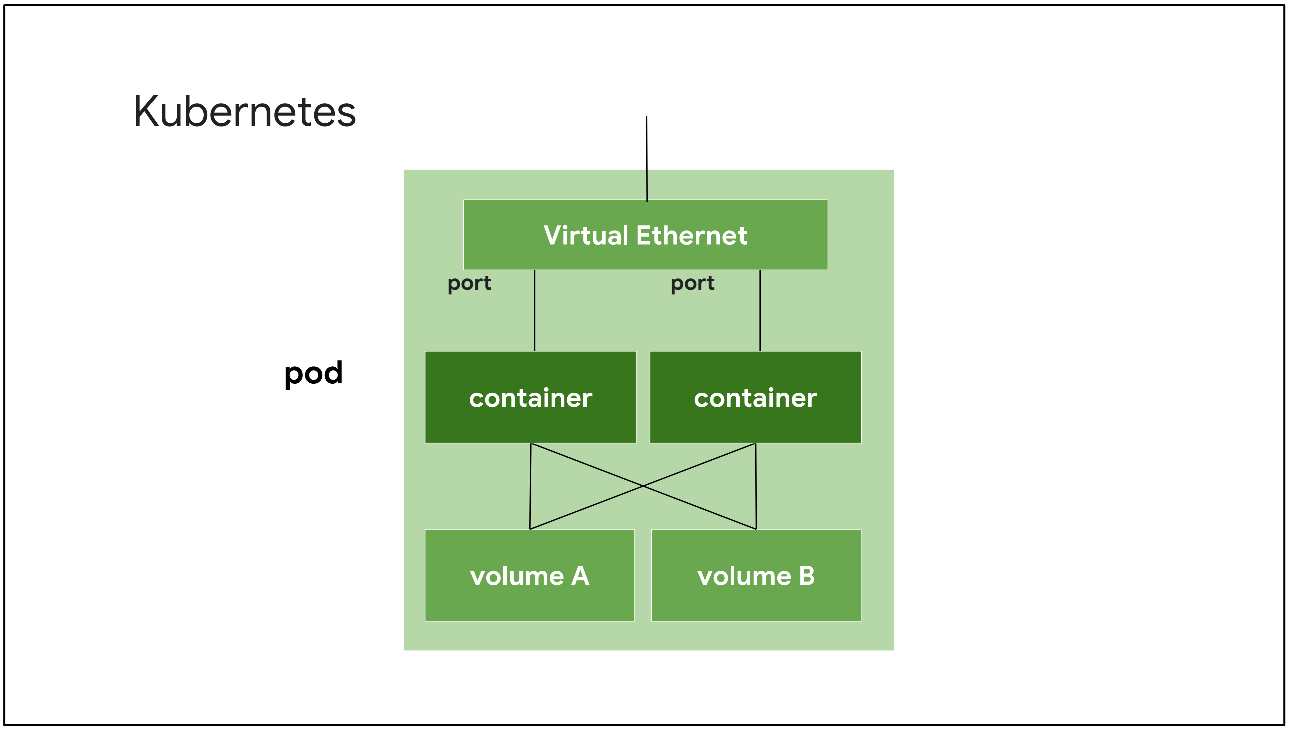
GKE is a hosted Kubernetes by Google. GKE clusters can be customized and they support different machine types, number of nodes, and network settings.

To start up Kubernetes on a **cluster** in GKE, all you do is run this command:

At this point, you should have a cluster called 'k1' configured and ready to go.

You can check its status in admin console.

6:00



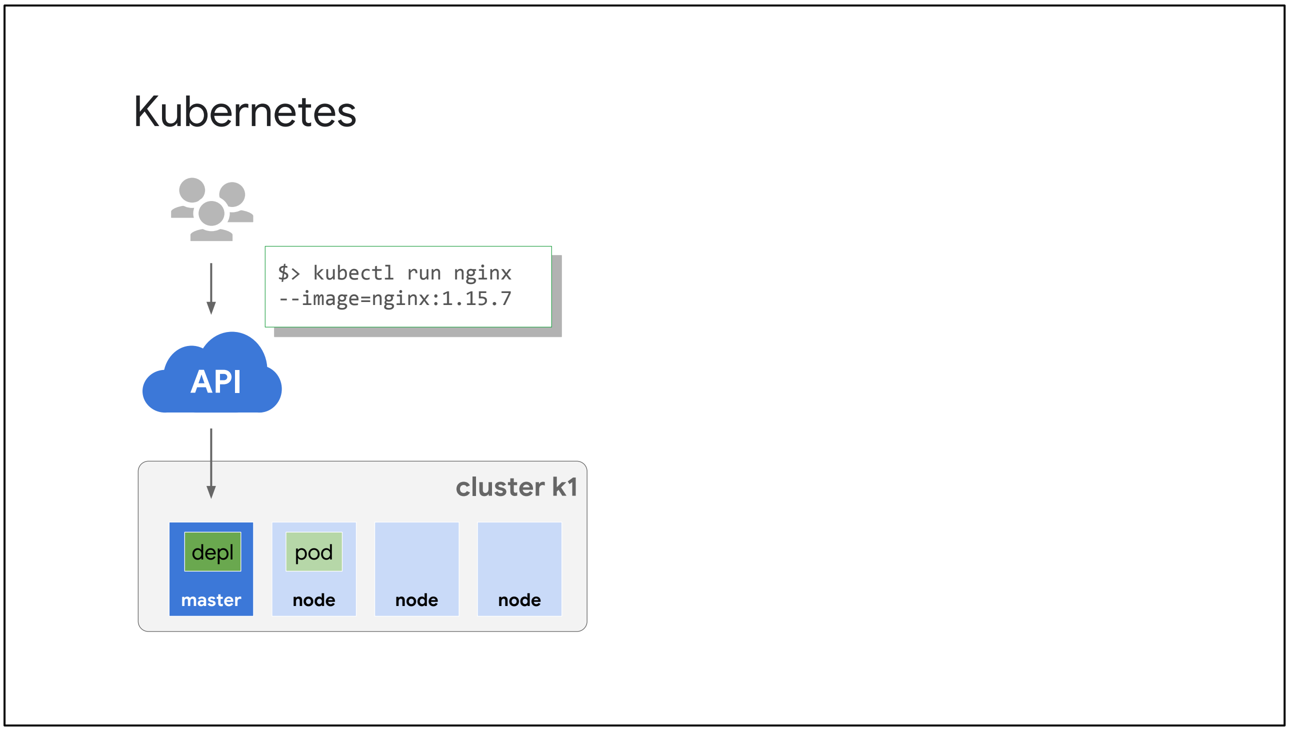
Then you deploy **containers** on nodes using a wrapper around one or more containers called a Pod.

A **Pod** is the smallest unit in Kubernetes that you create or deploy. A Pod represents a running process on your cluster as either a component of your application or an entire app.

Generally, you only have one container per pod, but if you have multiple containers with a hard dependency, you can package them into a single pod and share networking and storage. The Pod provides a unique network IP and set of ports for your containers, and options that govern how containers should run.

Containers inside a Pod can communicate with one another using localhost and ports that remain fixed as they're started and stopped on different nodes.

6:30

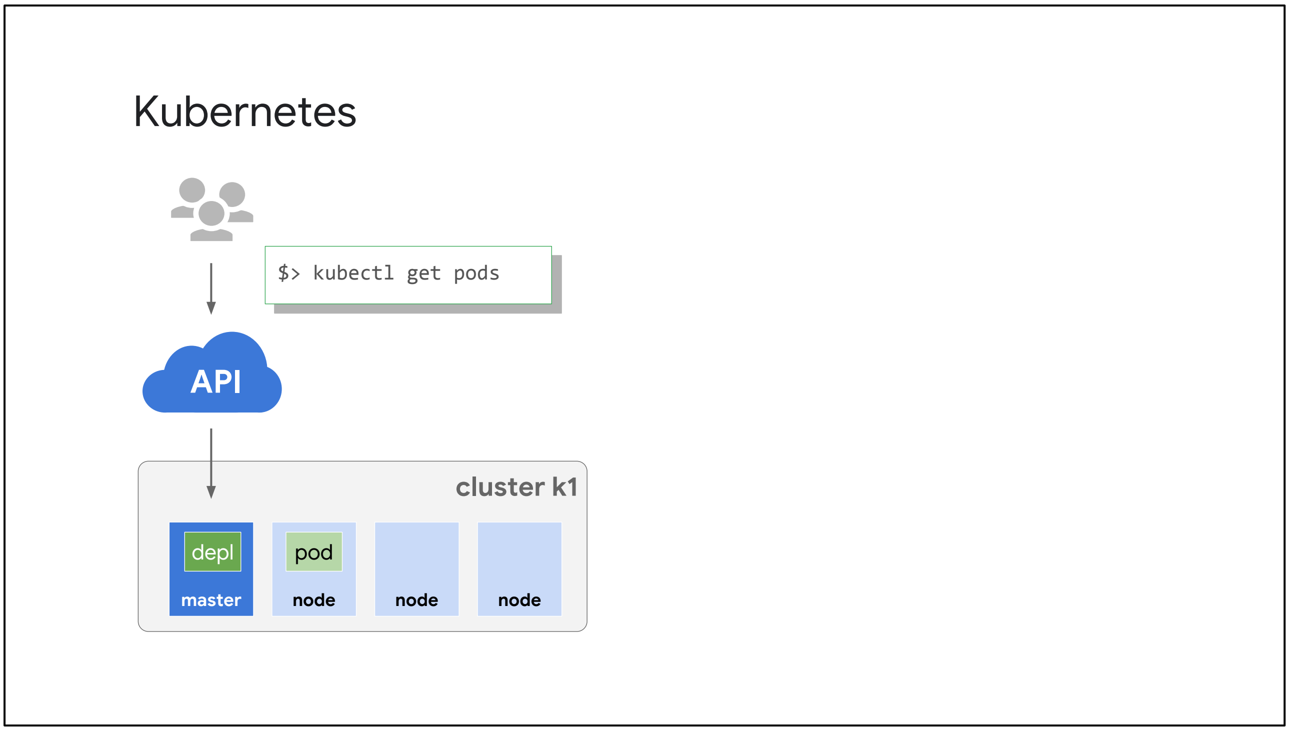


Kubernetes

```
$> kubectl run nginx
--image=nginx:1.15.7
```

One way to run a container in a Pod in Kubernetes is to use the **kubectl run** command. We'll learn a better way later in this module, but this gets you started quickly.

This starts a **Deployment** with a container running in a **Pod** and the container inside the Pod is an image of the nginx server.

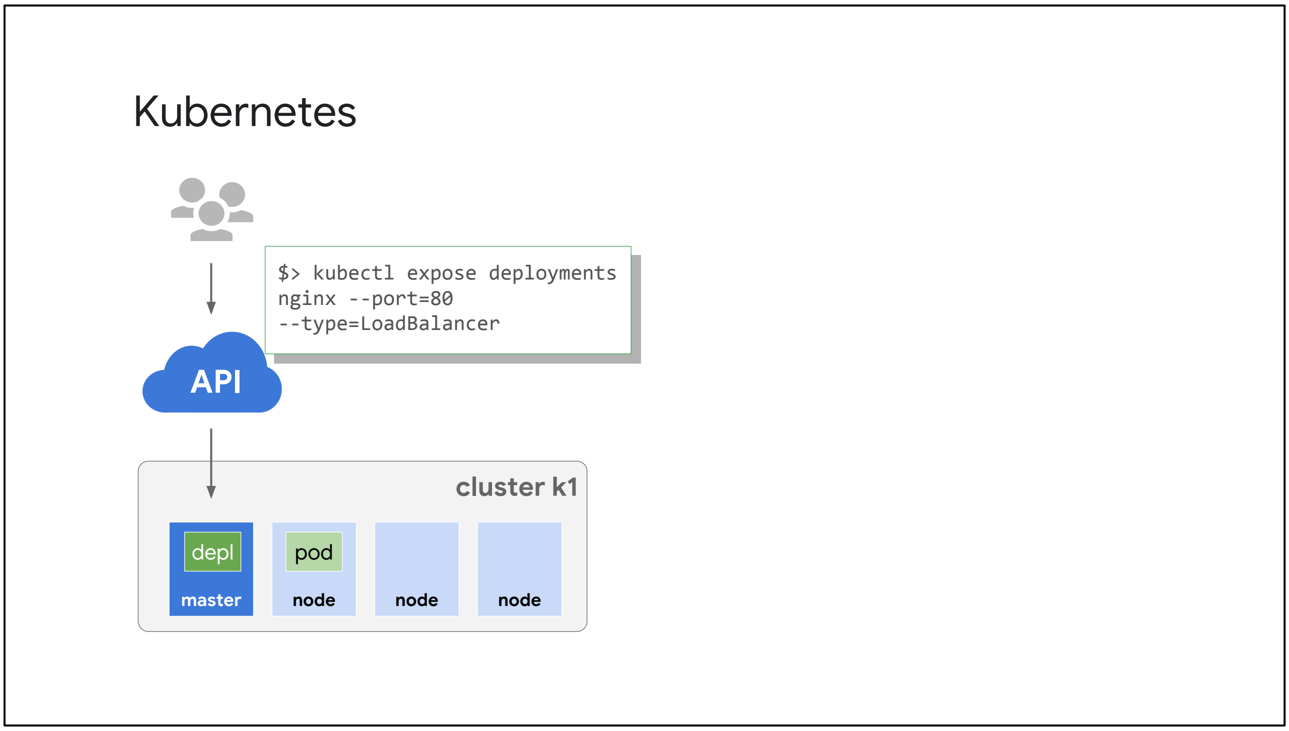


A **Deployment** represents a group of replicas of the same Pod and keeps your Pods running even when nodes they run on fail. It could represent a component of an application or an entire app. In this case, it's the nginx web server.

To see the running nginx Pods, run the command:

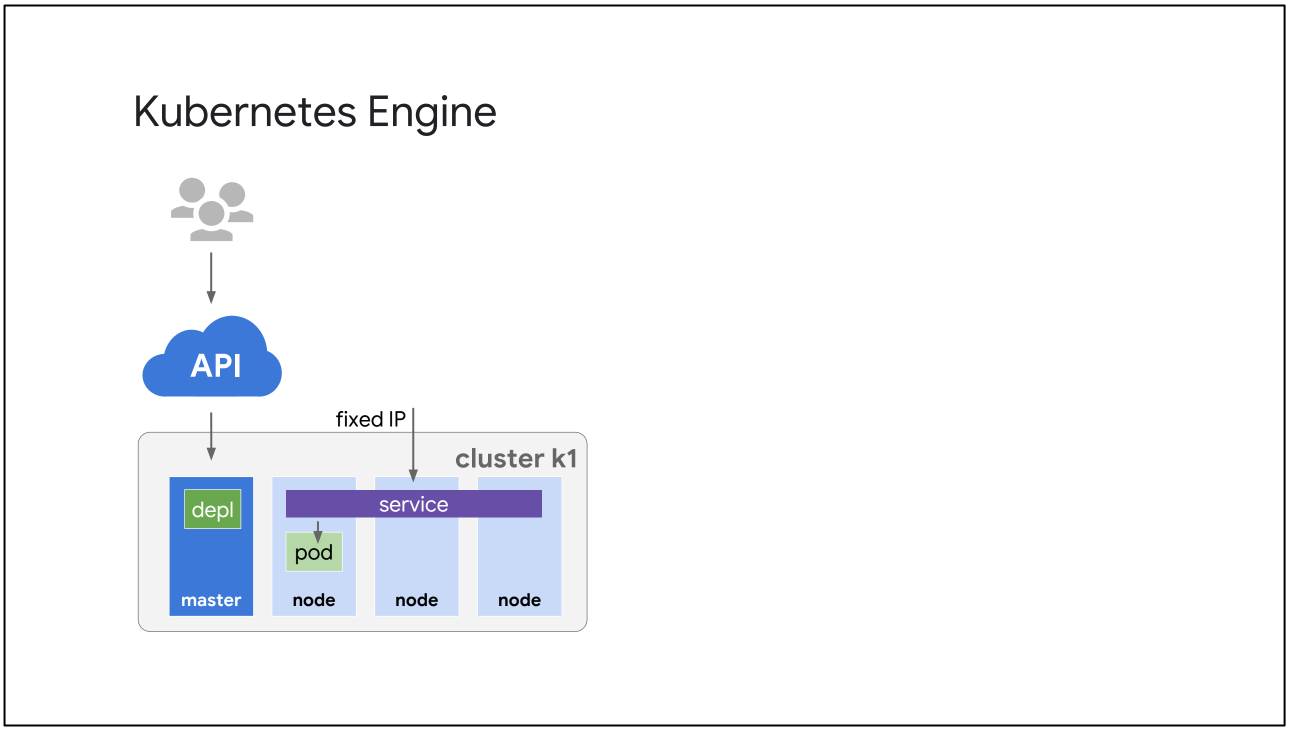
```
$ kubectl get pods
```

7:00


Kubernetes

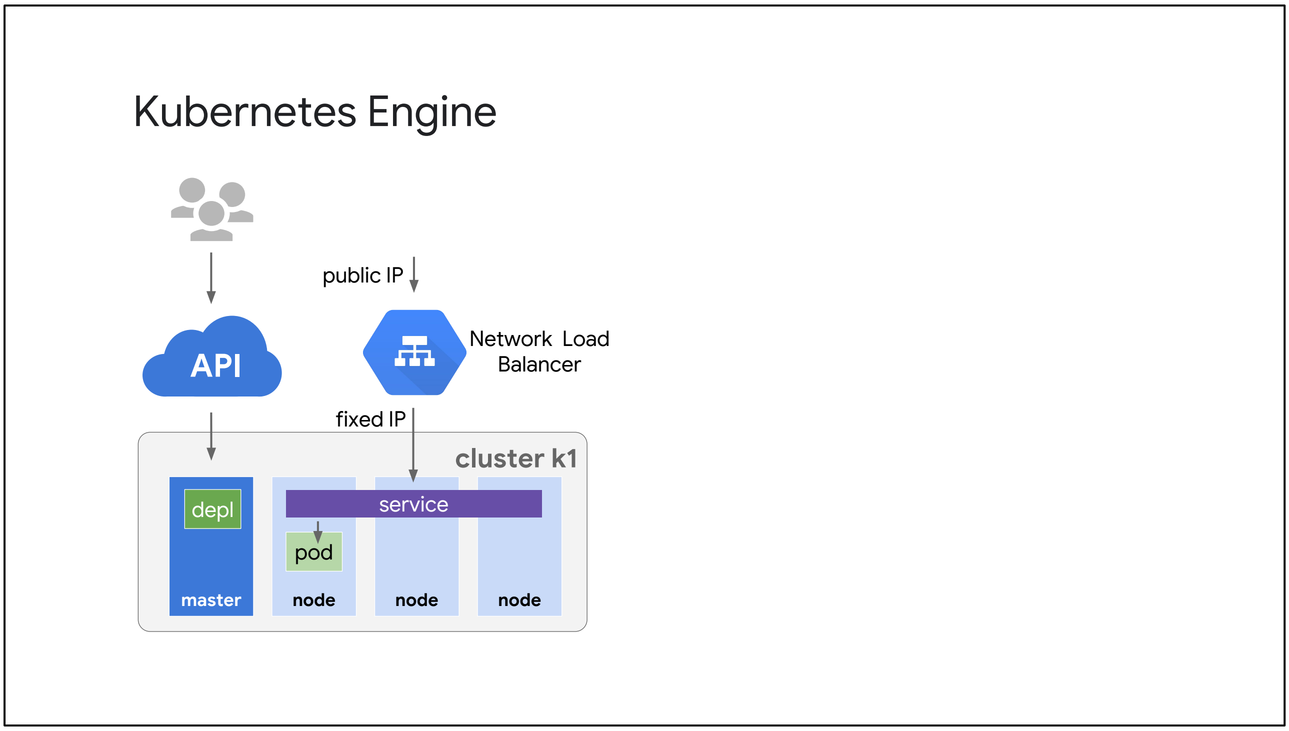
```
$> kubectl expose deployments
nginx --port=80
--type=LoadBalancer
```

By default, Pods in a Deployment are only accessible inside your GKE cluster. To make them publicly available, you can connect a load balancer to your Deployment by running the **kubectl expose** command:



Kubernetes Engine

Kubernetes creates a **Service** with a fixed IP for your Pods,



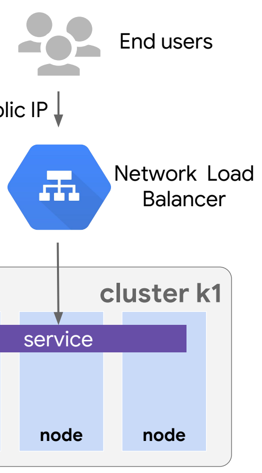
Kubernetes Engine

and a controller says "I need to attach an external **load balance**r with a public IP address to that **Service** so others outside the cluster can access it".

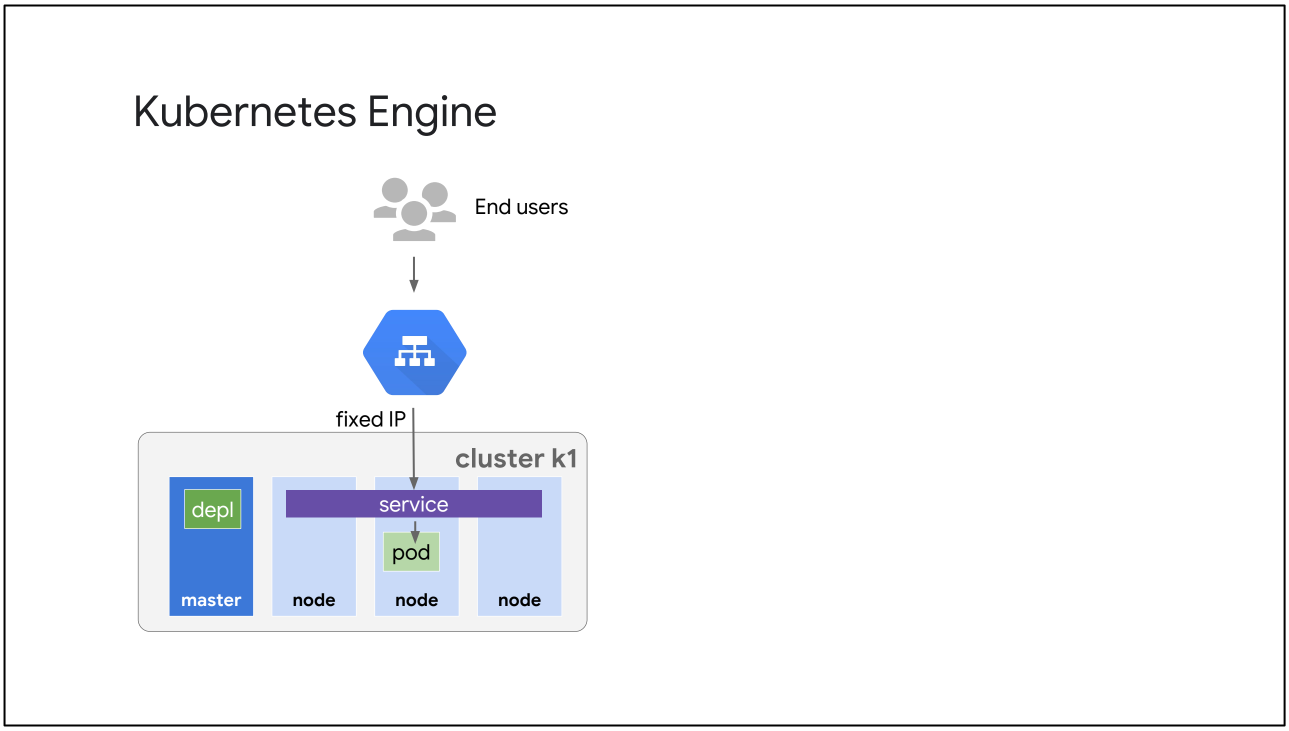
In **GKE**, the load balancer is created as a **Network Load Balancer**.

7:30

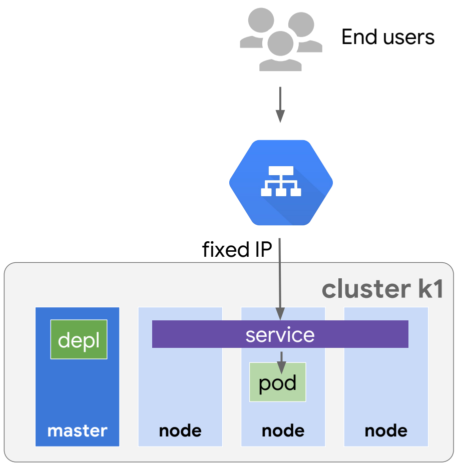
Kubernetes Engine



Any client that hits that IP address will be routed to a Pod behind the Service, in this case there is only one--your simple nginx Pod.



Kubernetes Engine



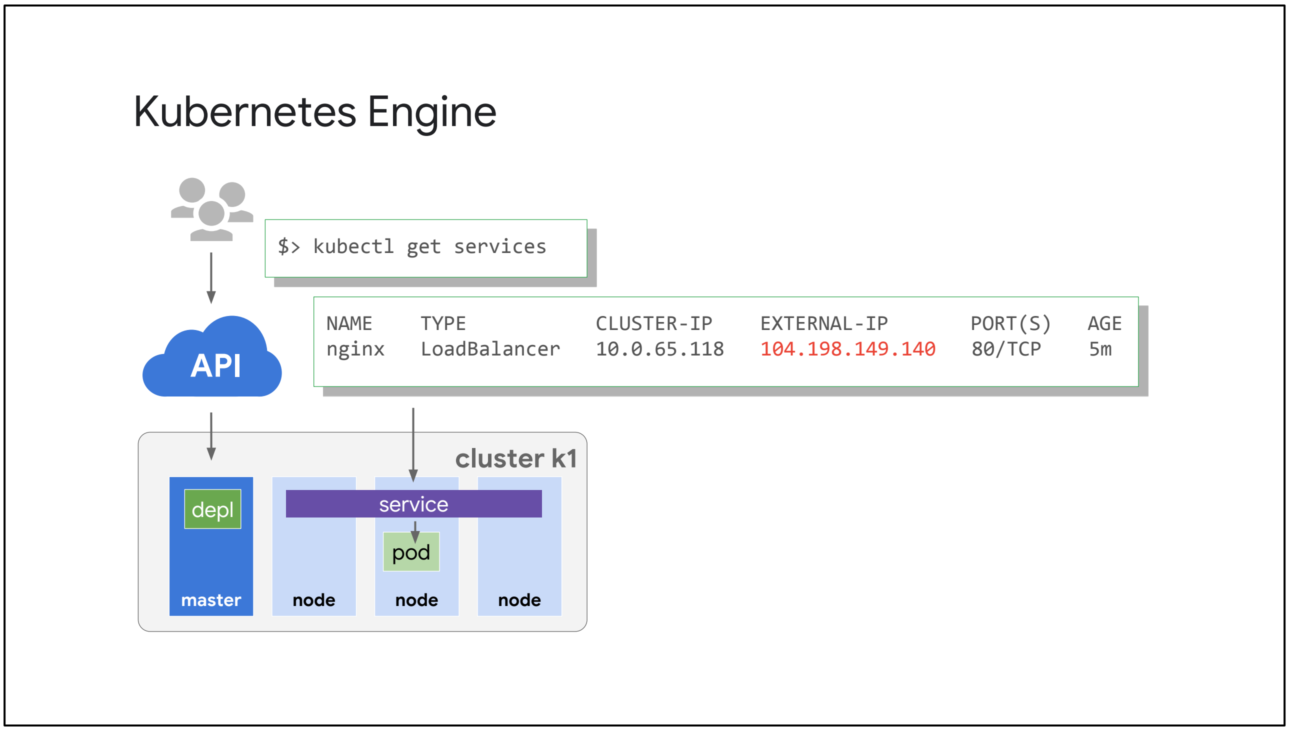
A **Service** is an abstraction which defines a logical set of Pods and a policy by which to access them.

As Deployments create and destroy Pods, Pods get their own IP address. But those addresses don't remain stable over time.

A Service groups a set of Pods and provides a stable endpoint (or fixed IP) for them.

For example, if you create two sets of Pods called frontend and backend, and put them behind their own Services, backend Pods may change, but frontend Pods are not aware of this. They simply refer to the backend Service.

8:15

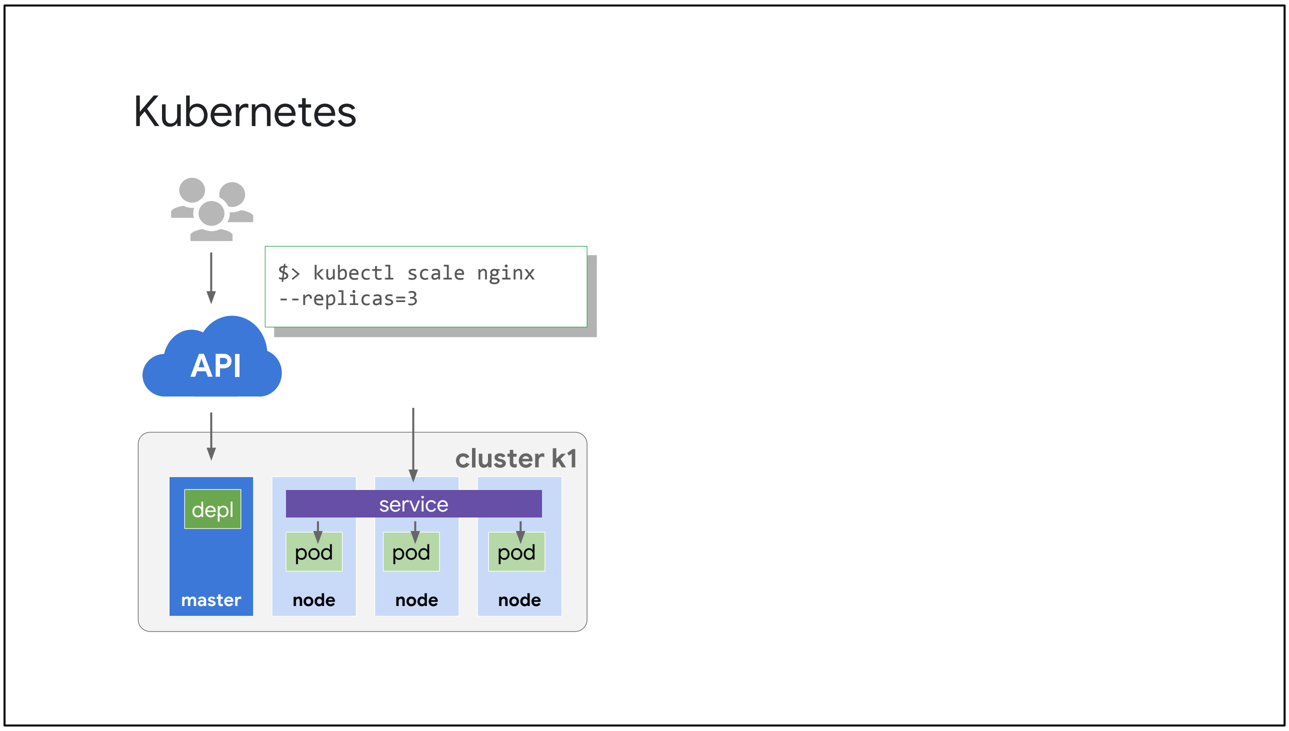


Kubernetes Engine

```
$> kubectl get services
```

```
NAME    TYPE          CLUSTER-IP    EXTERNAL-IP     PORT(S)  AGE
nginx   LoadBalancer  10.0.65.118   104.198.149.140 80/TCP   5m
```

You can run the **kubectl get services** command to get the public IP to hit the nginx container remotely.

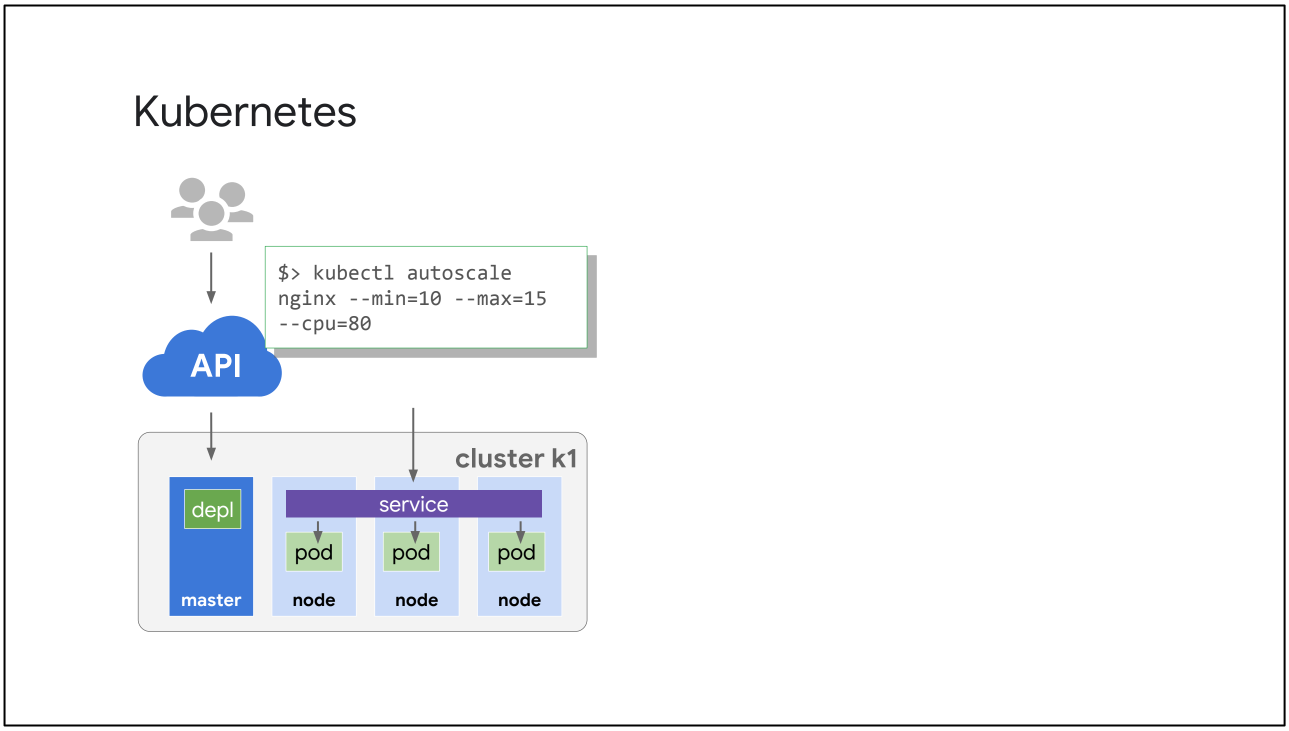

Kubernetes

```
$> kubectl scale nginx
--replicas=3
```

To scale a Deployment, run the **kubectl scale** command.

In this case, three Pods are created in your Deployment and they're placed behind the Service and share one fixed IP.

8:30

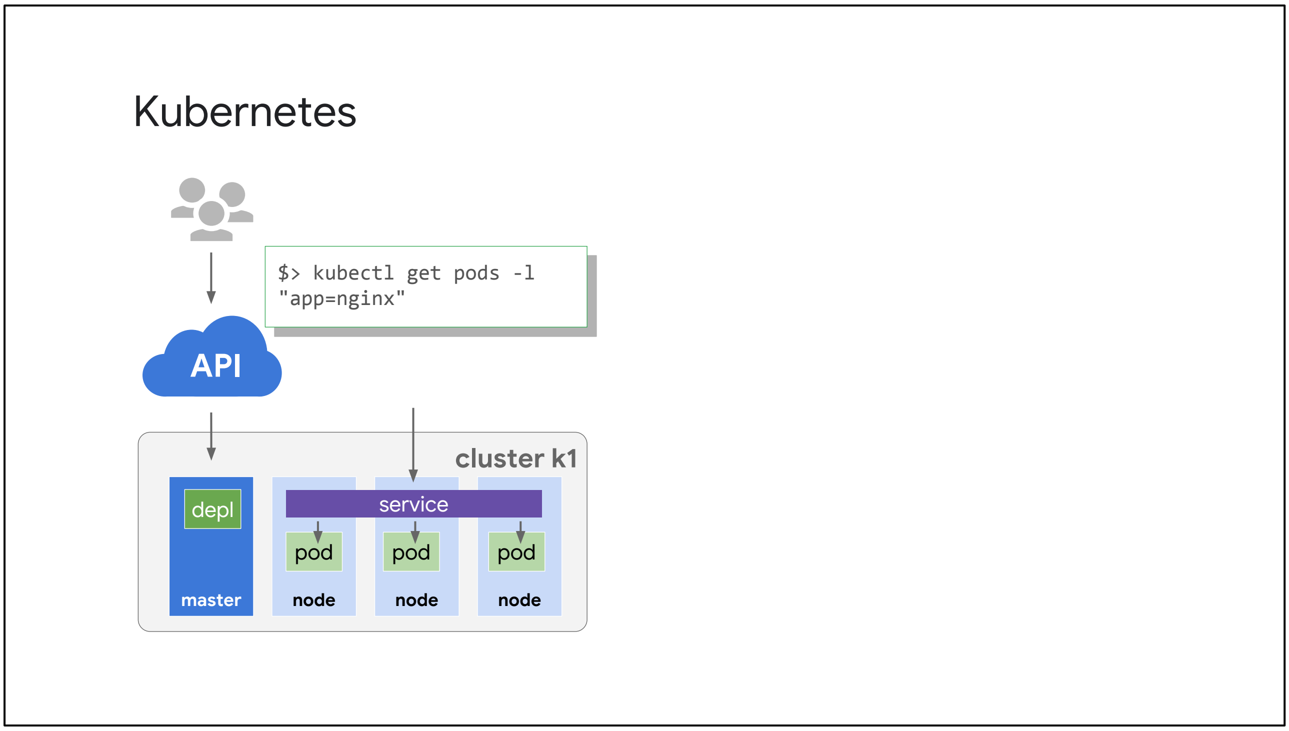


Kubernetes

```
$> kubectl autoscale
nginx --min=10 --max=15
--cpu=80
```

You could also use **autoscaling** with all kinds of parameters.

Here's an example of how to autoscale the Deployment to between 10 and 15 Pods when CPU utilization reaches 80 percent.



Kubernetes

```
$> kubectl get pods -l
"app=nginx"
```

So far, I've shown you how to run **imperative** commands like **expose** and **scale**. This works well to learn and test Kubernetes step-by-step.

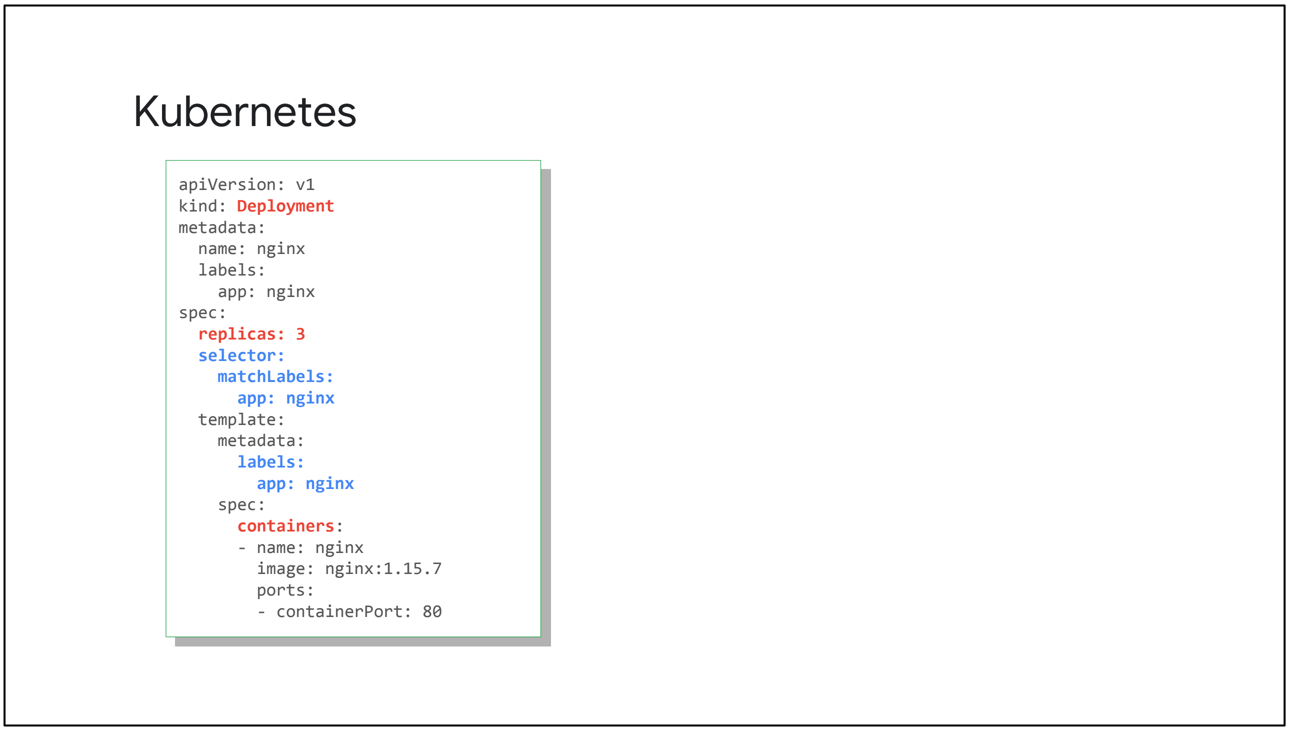
But the real strength of Kubernetes comes when you work in a **declarative** way.

Instead of issuing commands, you provide a configuration file that tells Kubernetes what you want your desired state to look like, and Kubernetes figures out how to do it.

Let me show you how to scale your Deployment using an existing Deployment config file.

To get the file, you can run a kubectl get pods command like the following.

9:00



Kubernetes

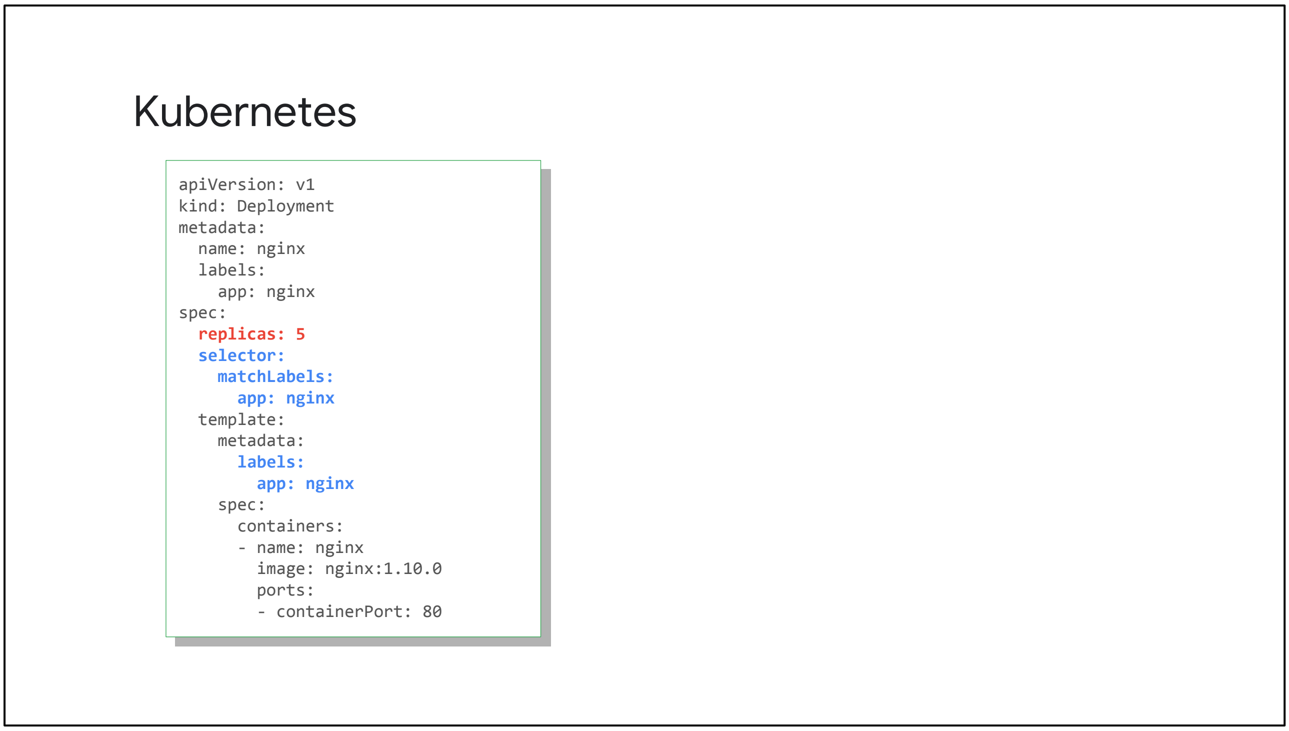
```
apiVersion: v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.7
        ports:
        - containerPort: 80
```

And you'll get a Deployment configuration file like the following.

In this case, it declares you want three replicas of your nginx Pod.

It defines a **selector** field so your Deployment knows how to group specific Pods as replicas, and you add a **label** to the Pod template so they get selected.

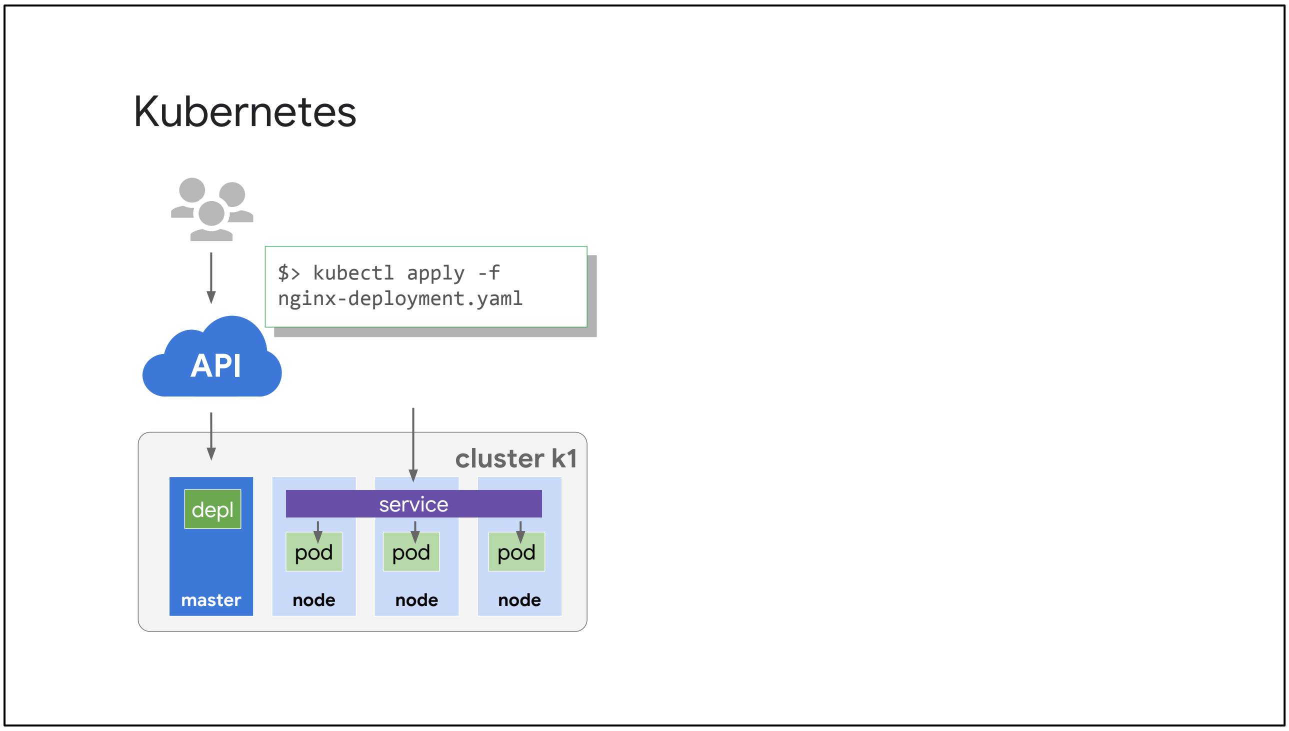
9:15



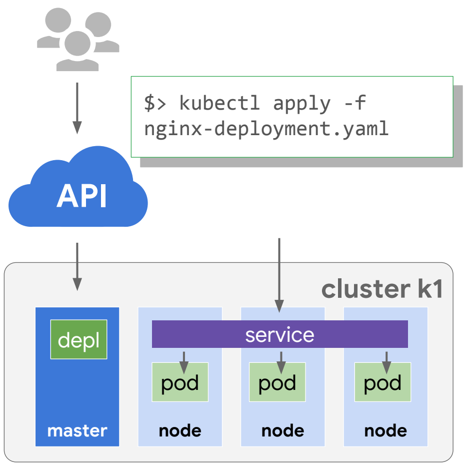
Kubernetes

```
apiVersion: v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.10.0
        ports:
        - containerPort: 80
```

To run five replicas instead of three, all you do is update the Deployment config file.

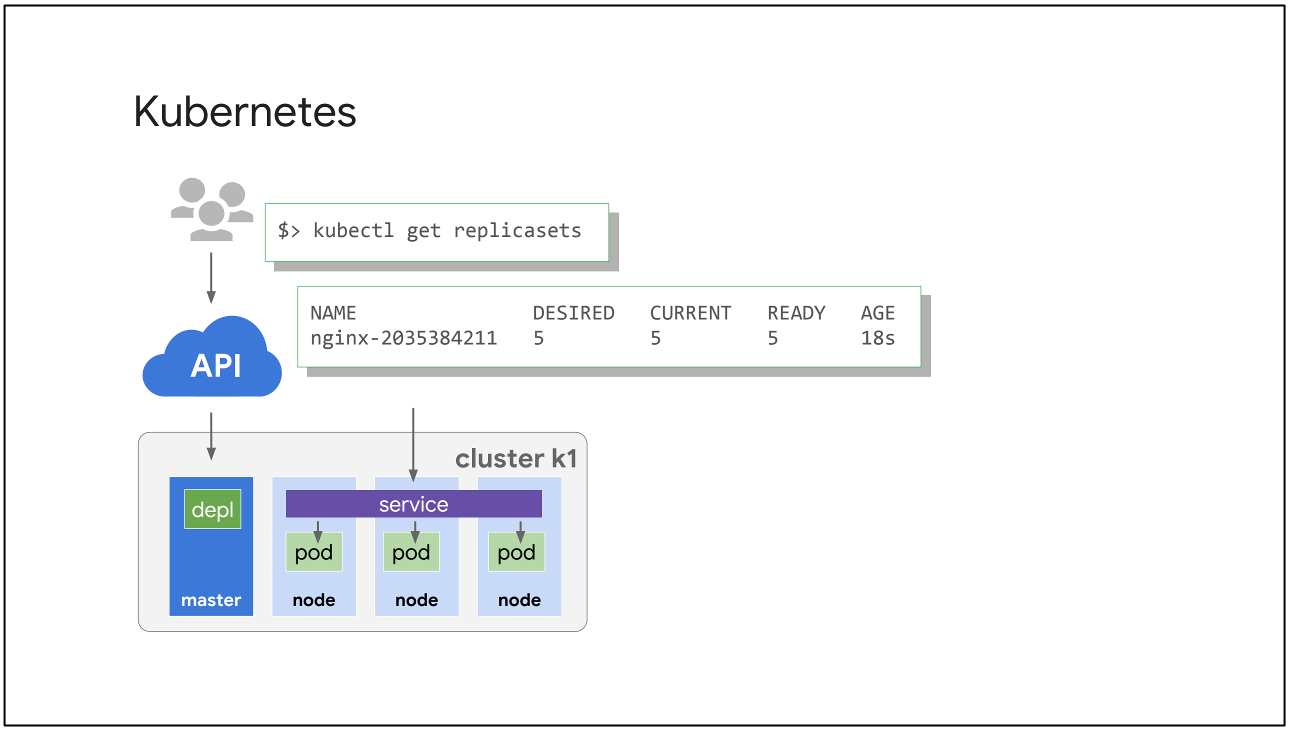


Kubernetes



```
$> kubectl apply -f
nginx-deployment.yaml
```

And run the **kubectl apply** command to use the config file.

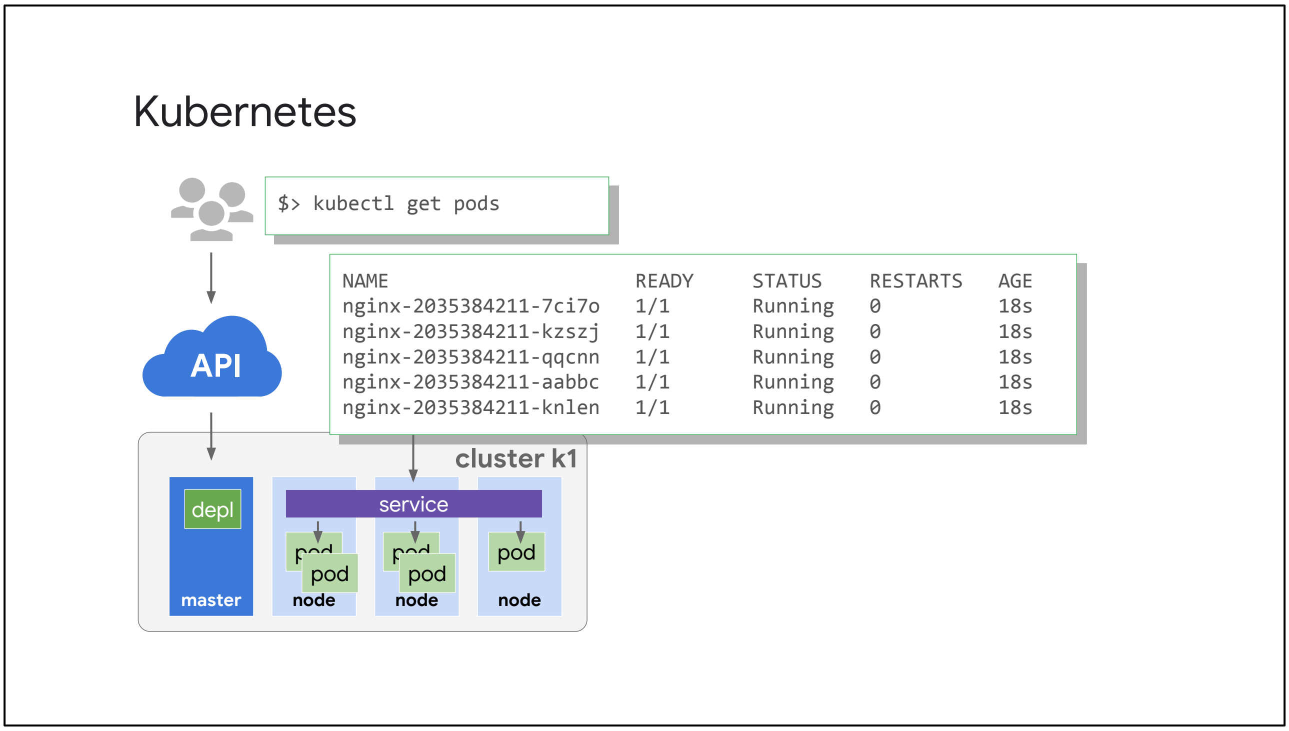


Kubernetes

```
$> kubectl get replicasets
```

```
NAME              DESIRED    CURRENT    READY    AGE
nginx-2035384211  5          5          5        18s
```

Now look at your replicas to see their updated state.



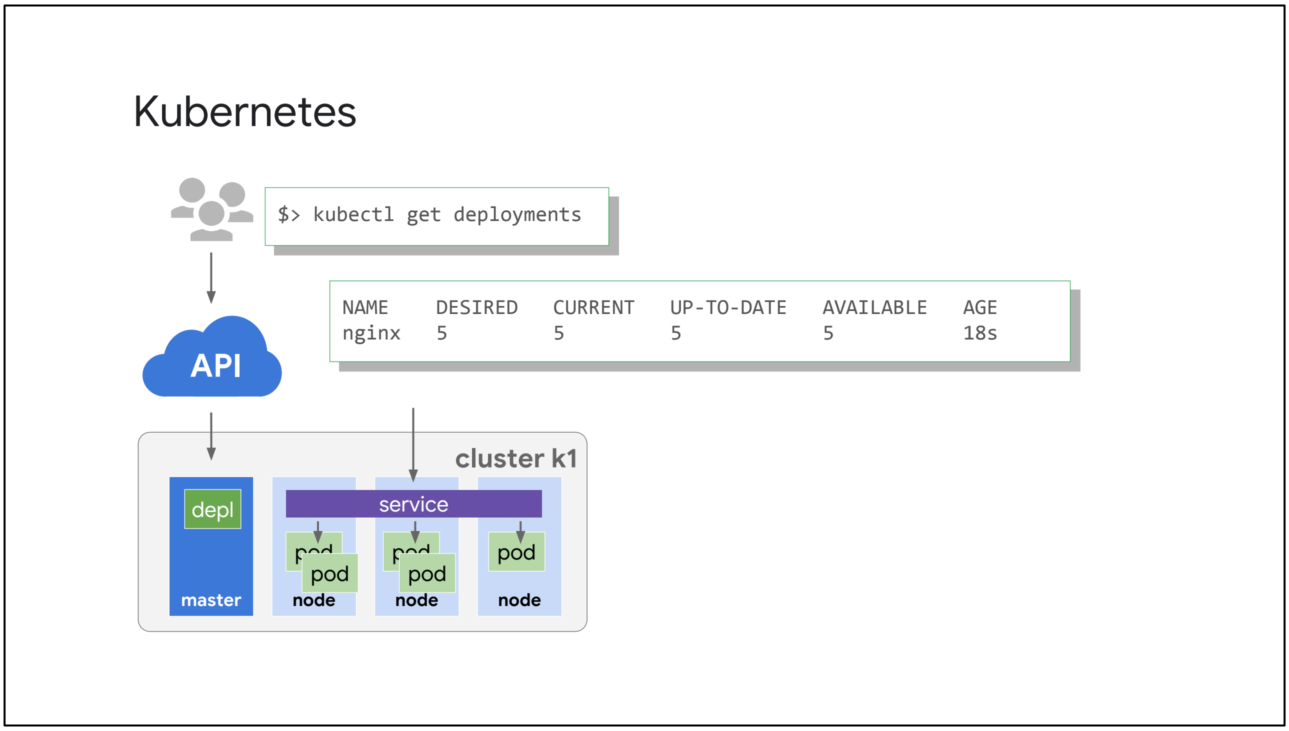
Kubernetes

```
$> kubectl get pods
```

```
NAME                         READY     STATUS     RESTARTS     AGE
nginx-2035384211-7ci7o       1/1       Running    0            18s
nginx-2035384211-kzszj       1/1       Running    0            18s
nginx-2035384211-qqcnn       1/1       Running    0            18s
nginx-2035384211-aabbc       1/1       Running    0            18s
nginx-2035384211-knlen       1/1       Running    0            18s
```

Then use the **kubectl get pods** command to watch the pods come on line.

In this case, all five are **READY** and **RUNNING**.



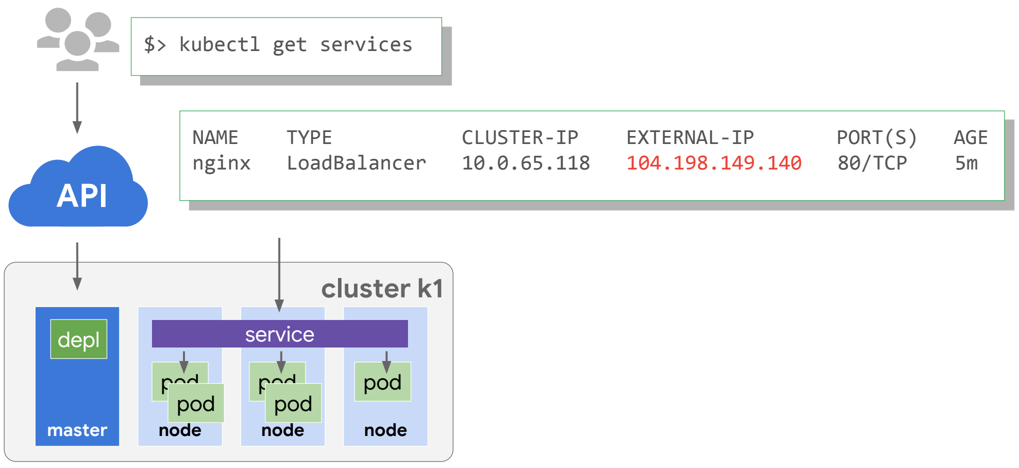
Kubernetes

```
$> kubectl get deployments
```

```
NAME     DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
nginx    5          5          5             5            18s
```

And check the Deployment to make sure the proper number of replicas are running using either $ kubectl **get deployments** or $ kubectl **describe deployments**. In this case, all five Pod replicas are **AVAILABLE**.

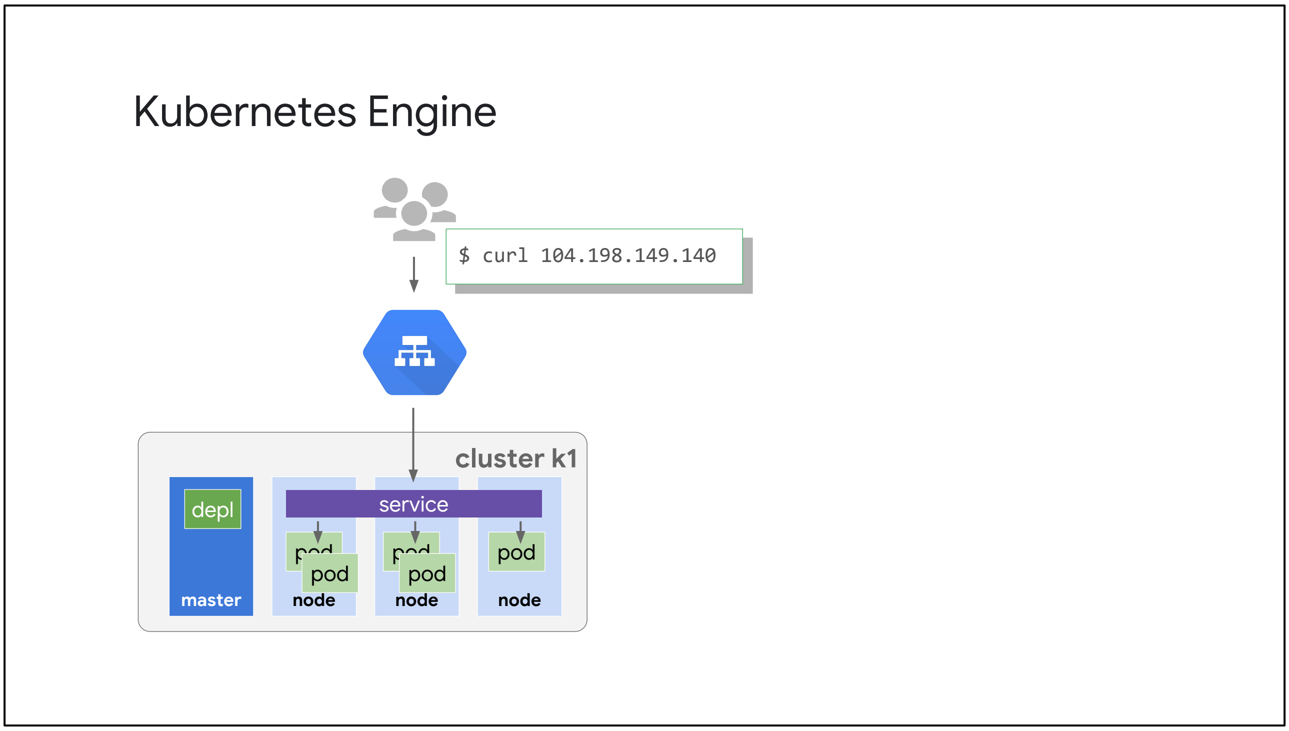
Kubernetes



```
$> kubectl get services
```

```
NAME    TYPE          CLUSTER-IP   EXTERNAL-IP      PORT(S)  AGE
nginx   LoadBalancer  10.0.65.118  104.198.149.140  80/TCP   5m
```

And you can still hit your endpoint like before using $ kubectl **get services** to get the external IP of the Service,



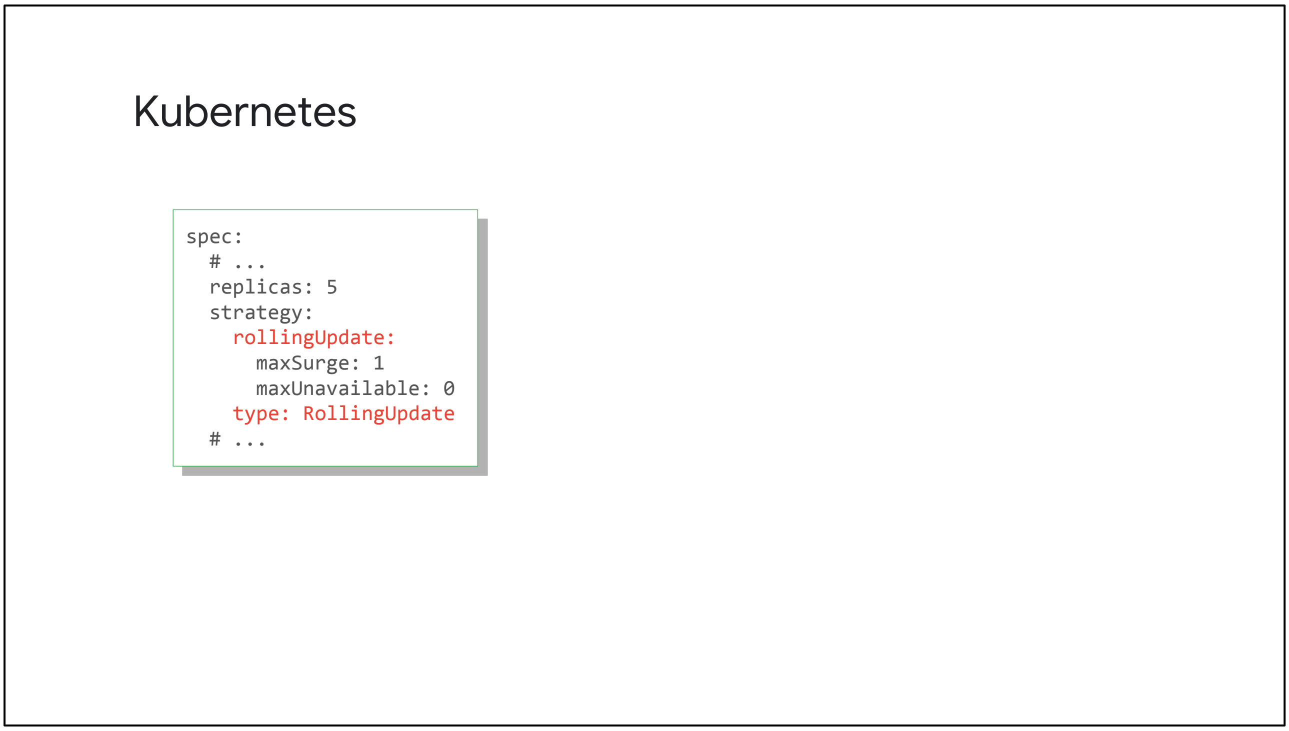
Kubernetes Engine

```
$ curl 104.198.149.140
```

and hit the public IP from a client.

At this point, you have five copies of your nginx Pod running in GKE, and you have a single Service that's proxying the traffic to all five Pods. This allows you to share the load and scale your Service in Kubernetes.

10:15



Kubernetes

```
spec:
  # ...
  replicas: 5
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  # ...
```

The last question is what happens when you want to update a new version of your app?

You want to update your container to get new code out in front of users, but it would be risky to roll out all those changes at once.

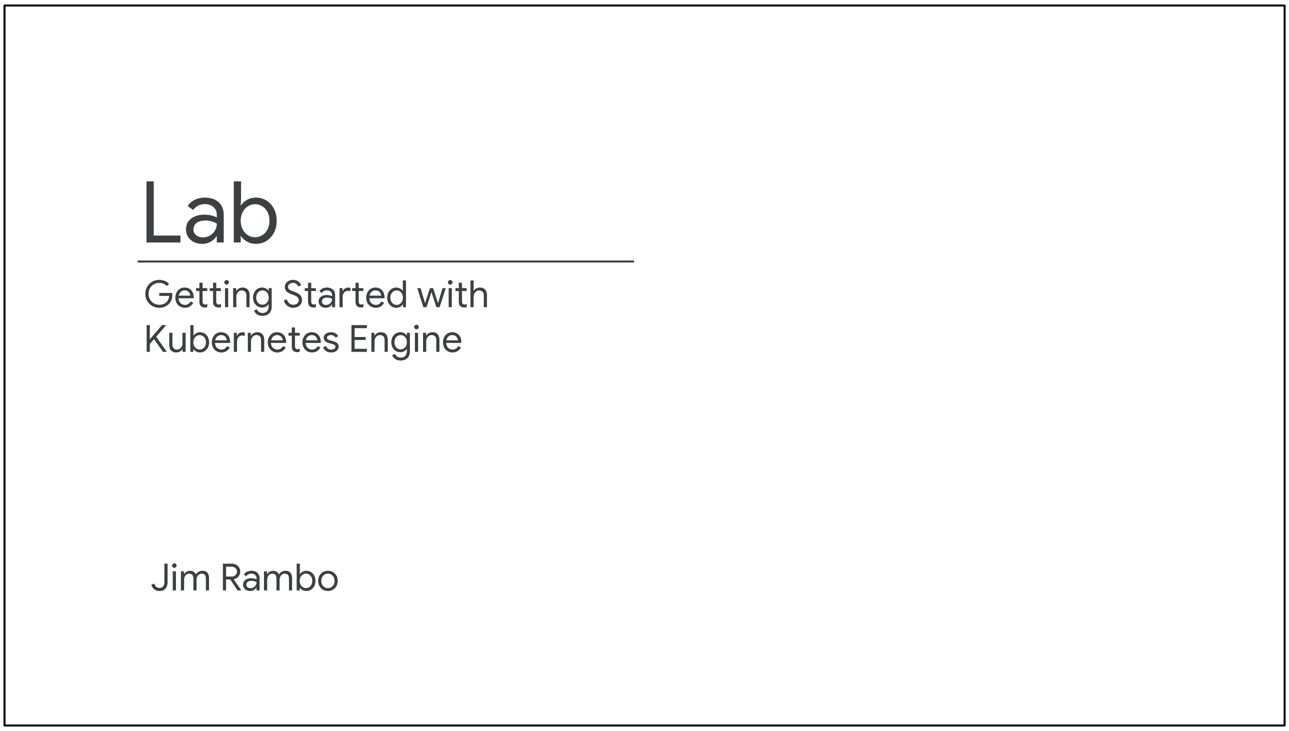
So you use **kubectl rollout** or change your deployment configuration file and apply the change using **kubectl apply**.

New Pods will be created according to your update strategy. Here is an example configuration that will create new version Pods one by one, and wait for a new Pod to be available before destroying one of the old Pods.

Video Name: T-GCPFCI-B_5_l2_LabIntroKubernetesEngine

Content Type: Video - Lecture Presenter

Presenter: Jim Rambo



Lab

Getting Started with Kubernetes Engine

Jim Rambo

There are a lot of features in Kubernetes and GKE we haven't even touched on such as: configuring health checks, setting session affinity, managing different rollout strategies, and deploying Pods across regions for high-availability. But for now, that's enough.

In this module, you've learned how to:

- Build and run containerized applications
- Orchestrate and scale them on a cluster
- And deploy them using rollouts.

Now you'll see how to do this in a demo and practice it in a lab exercise.

11:00

Video Name:
T-GCPFCI-B_M5_L4_GettingStartedWithKubernetesEngine

Content Type: Video - Lecture Presenter

Presenter: Brian Rice

Note to video editor: this video is linked in the course map and was already created. Nothing to edit here for the V2 of the module