This scenario explains how to launch a simple, multi-tier web application using Kubernetes and Docker. The Guestbook example application stores notes from guests in Redis via JavaScript API calls. Redis contains a master (for storage), and a replicated set of redis 'slaves'.

Core Concepts
The following core concepts will be covered during this scenario. These are the foundations of understanding Kubernetes.

Pods : In Kubernetes, a pod is the smallest and simplest unit in the Kubernetes object model. A pod represents a single instance of a running process in a cluster, and it can contain one or more containers that share the same network namespace and file system.

Each pod in Kubernetes has a unique IP address within the cluster, and it can communicate with other pods and services within the cluster using that IP address. Pods can also share volumes, which allows them to share data between containers in the same pod.

When a pod is created, Kubernetes automatically assigns it to a node in the cluster. The node is responsible for running the containers in the pod, and it provides the resources that the containers need, such as CPU, memory, and storage.

Pods are usually created and managed by a higher-level abstraction called a controller, such as a Deployment or a StatefulSet. Controllers ensure that the desired number of replicas of a pod are running at all times, and they can automatically recover from failures by creating new pods to replace ones that have failed.

Replication Controllers : Replication Controllers (RCs) are one of the core components of the Kubernetes system that manage the lifecycle of pod replicas. A Replication Controller ensures that a specified number of replicas of a pod are running at any given time.

The Replication Controller continuously monitors the state of the pods it is managing, and if it detects that a pod has failed or has been terminated for any reason, it automatically creates a new pod to replace it. This makes Replication Controllers a key element in ensuring that the desired level of availability and fault tolerance is maintained for the applications running on a Kubernetes cluster.

To use a Replication Controller, you define a pod template that describes the desired state of the pod(s) that the Replication Controller should manage. You also specify the number of replicas that should be created, and the Replication Controller will ensure that this number is maintained at all times.

In addition to managing pod replicas, Replication Controllers also provide rolling updates to the pods. This means that when a new version of a pod template is deployed, the Replication Controller gradually replaces the old replicas with the new ones, ensuring that the application is always available during the update process.

While Replication Controllers are a powerful tool for managing pod replicas, they have been superseded by newer controllers in Kubernetes, such as Deployments and StatefulSets, which offer additional features and capabilities. However, Replication Controllers are still widely used and are an important component of many Kubernetes deployments.

Services : In Kubernetes, a Service is an abstraction layer that provides a consistent and reliable way to access a group of pods that perform the same function. Services act as a single entry point for clients to access the pods, even as pods are created, deleted, or replaced.

When you create a Service in Kubernetes, you specify a selector that identifies a group of pods that the Service should target. The Service then creates a stable virtual IP address and DNS name that clients can use to access the pods. The Service also load balances traffic across the pods, distributing requests evenly to each pod in the group.

There are several types of Services in Kubernetes, including:

ClusterIP: This is the default type of Service, and it provides a stable IP address and DNS name for clients to access the pods within the cluster.

NodePort: This type of Service exposes the pods to the outside world by creating a port on each node in the cluster that maps to the Service. This allows clients to access the pods from outside the cluster.

LoadBalancer: This type of Service automatically provisions a load balancer in the cloud environment to distribute traffic to the pods. This is typically used when running Kubernetes on a cloud provider such as AWS or GCP.

ExternalName: This type of Service allows you to create a DNS alias for an external service outside of the Kubernetes cluster.

Services play a critical role in Kubernetes because they allow applications to be abstracted away from the underlying infrastructure, making it easier to manage and scale applications. By providing a stable IP address and DNS name, Services also make it easy for clients to access applications running on a Kubernetes cluster, regardless of which pod the application is running on.

NodePorts : I believe you mean "NodePorts" in Kubernetes. NodePort is a type of Service in Kubernetes that allows external clients to access the pods in a cluster by exposing a specific port on each node in the cluster.

When you create a NodePort Service, Kubernetes assigns a port in the range of 30000-32767 to the Service. Clients can then access the Service using the IP address of any node in the cluster and the assigned port.

For example, if you create a NodePort Service with port 80, clients can access the Service using the IP address of any node in the cluster and the port number assigned by Kubernetes (e.g. http://node-ip:node-port).

NodePort Services are useful when you need to expose an application running in the cluster to the outside world, or when you need to access a Service from a machine outside of the cluster. However, they should be used with caution, as they can expose the pods to the public internet and potentially create security risks.

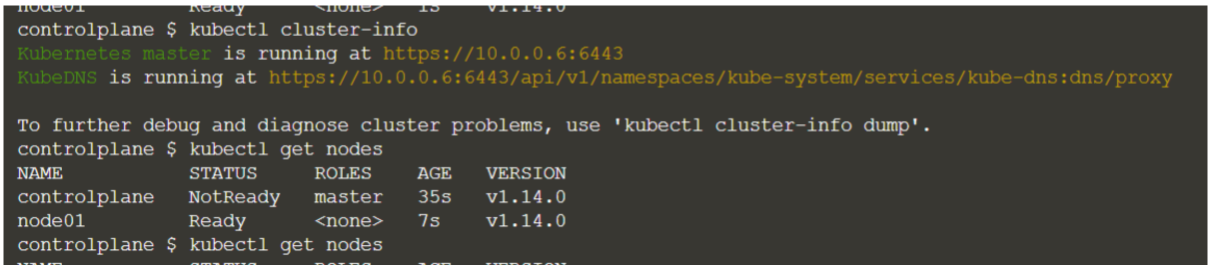
To secure NodePort Services, you can use firewalls or other network security measures to restrict access to the ports used by the Service. Additionally, you can use tools like Ingress or load balancers to provide more advanced routing and load balancing capabilities.

Step 1 - Start Kubernetes
To start we need a running Kubernetes cluster.
Task
Start a single-node cluster using the helper script. The helper script will launch the API, Master, a Proxy and DNS discovery. The Web App uses DNS Discovery to find the Redis slave to store data.

launch.sh

Health Check
Check everything is up using the following health Check: kubectl cluster-info
kubectl get nodes

If the node returns NotReady then it is still waiting. Wait a couple of seconds before retrying.



```
node01         Ready      <none>   1s    v1.14.0
controlplane $ kubectl cluster-info
Kubernetes master is running at https://10.0.0.6:6443
KubeDNS is running at https://10.0.0.6:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
controlplane $ kubectl get nodes
NAME           STATUS     ROLES    AGE   VERSION
controlplane   NotReady   master   35s   v1.14.0
node01         Ready      <none>   7s    v1.14.0
controlplane $ kubectl get nodes
```

In Kubernetes, a single-node cluster is a cluster that consists of a single Kubernetes node, which runs all of the control plane components, worker processes, and user applications on a single machine.

Running a single-node cluster can be useful for development, testing, or small-scale production deployments where high availability is not a primary concern. It can also be a good way to get started with Kubernetes and experiment with its features without having to set up a larger cluster.

However, running a single-node cluster can have some limitations and drawbacks. For example, if the node fails or becomes unavailable, the entire cluster will go down. Additionally, since all of the control plane components and worker processes run on the same machine, there may be resource constraints that limit the amount of work that can be done.

To mitigate these issues, it is generally recommended to run multi-node clusters in production environments. Multi-node clusters distribute the workloads across multiple nodes, providing higher availability and scalability. They also allow for better resource utilization and can be configured to tolerate node failures.

That being said, a single-node cluster can still be a useful tool for certain use cases and scenarios, especially when resources are limited or when a small and simple Kubernetes setup is sufficient for the needs of the application or workload.

In a Kubernetes cluster, the API server, the control plane components, the kubelet agents, and the worker nodes all work together to manage and run the applications and services. Here's how they work together:

API server: The API server is the primary interface for managing the Kubernetes cluster. It exposes the Kubernetes API, which allows users and applications to interact with the cluster. The API server runs on the master node and accepts requests from the kubectl command-line tool, Kubernetes Dashboard, and other Kubernetes components.

Master node: The master node is the control plane component of the Kubernetes cluster. It runs the API server, etcd (the distributed key-value store that stores the cluster's state), the Kubernetes scheduler (which schedules pods to run on worker nodes), and the Kubernetes controller manager (which manages the state of the cluster).

Worker nodes: The worker nodes are the compute nodes in the Kubernetes cluster. They run the kubelet agent, which communicates with the API server and manages the state of the pods running on the node. They also run the container runtime, such as Docker or containerd, which pulls the container images and runs them as pods.

Proxy: The Kubernetes proxy is responsible for routing traffic to the appropriate pod. It runs on each node in the cluster and listens to the Kubernetes API server for changes in the Service and Endpoint objects. It then sets up iptables rules to forward traffic to the correct pod.

DNS discovery: Kubernetes uses a built-in DNS service to provide service discovery within the cluster. Each Service object in the cluster is assigned a DNS name, which can be used by other pods in the cluster to communicate with the Service. When a pod tries to communicate with a Service, the DNS service resolves the DNS name to the IP addresses of the pods in the Service.

Together, these components provide the foundation for running and managing containerized applications in a Kubernetes cluster. They work together to provide a scalable, resilient, and flexible platform for deploying and managing applications and services.

Step 2 - Redis Master Controller
The first stage of launching the application is to start the Redis Master. A Kubernetes service deployment has, at least, two parts. A replication controller and a service.

The replication controller defines how many instances should be running, the Docker Image to use, and a name to identify the service. Additional options can be utilized for configuration and discovery. Use the editor above to view the YAML definition.

If Redis were to go down, the replication controller would restart it on an active node.

Create Replication Controller
In this example, the YAML defines a redis server called redis-master using the official redis running port 6379.

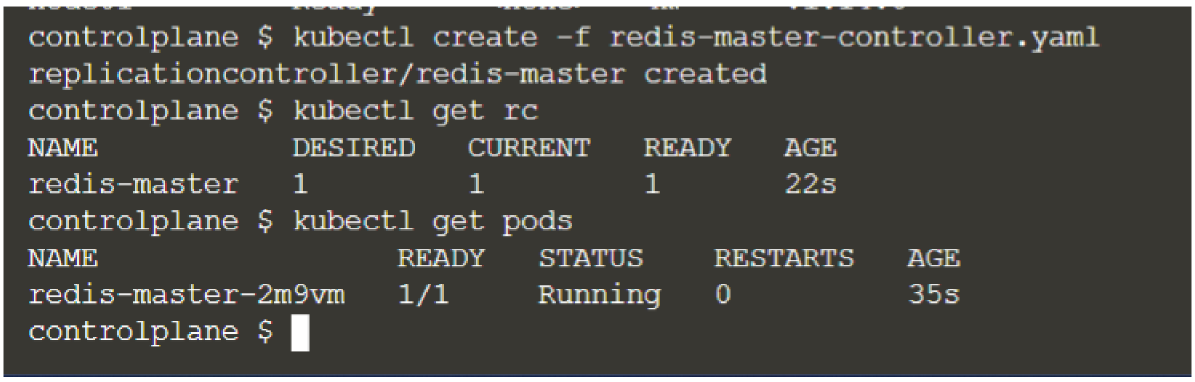
The kubectl create command takes a YAML definition and instructs the master to start the controller.

What's running?
The above command created a Replication Controller. The Replication

kubectl get rc

All containers described as Pods. A pod is a collection of containers that makes up a particular application, for example Redis. You can view this using kubectl

kubectl get pods



```
controlplane $ kubectl create -f redis-master-controller.yaml
replicationcontroller/redis-master created
controlplane $ kubectl get rc
NAME           DESIRED   CURRENT   READY   AGE
redis-master   1         1         1       22s
controlplane $ kubectl get pods
NAME                 READY   STATUS    RESTARTS   AGE
redis-master-2m9vm   1/1     Running   0          35s
controlplane $
```

Step 3 - Redis Master Service
The second part is a service. A Kubernetes service is a named load balancer that proxies traffic to one or more containers. The proxy works even if the containers are on different nodes.

Services proxy communicate within the cluster and rarely expose ports to an outside interface.

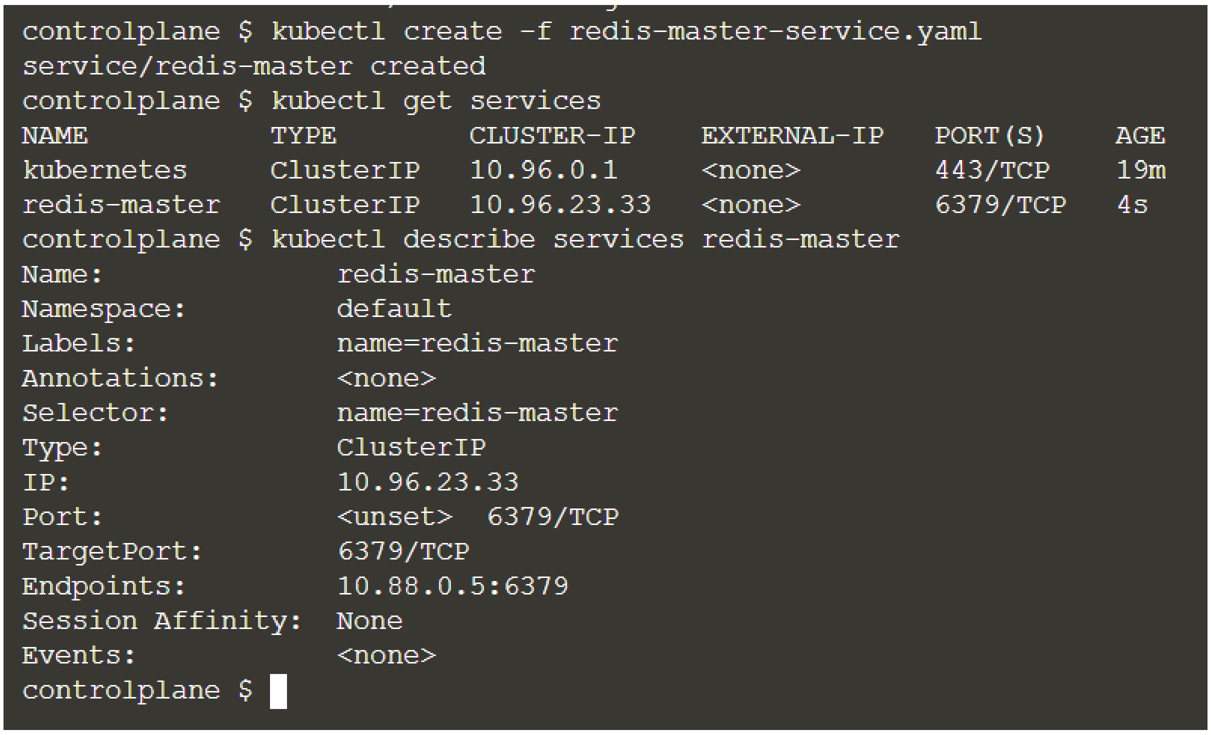
When you launch a service it looks like you cannot connect using curl or netcat unless you start it as part of Kubernetes. The recommended approach is to have a LoadBalancer service to handle external communications.

Create Service
The YAML defines the name of the replication controller, redis-master, and the ports which should be proxied.

kubectl create -f redis-master-service.yaml

List & Describe Services
kubectl get services

kubectl describe services redis-master



```
controlplane $ kubectl create -f redis-master-service.yaml
service/redis-master created
controlplane $ kubectl get services
NAME           TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
kubernetes     ClusterIP   10.96.0.1     <none>        443/TCP    19m
redis-master   ClusterIP   10.96.23.33   <none>        6379/TCP   4s
controlplane $ kubectl describe services redis-master
Name:              redis-master
Namespace:         default
Labels:            name=redis-master
Annotations:       <none>
Selector:          name=redis-master
Type:              ClusterIP
IP:                10.96.23.33
Port:              <unset>  6379/TCP
TargetPort:        6379/TCP
Endpoints:         10.88.0.5:6379
Session Affinity:  None
Events:            <none>
controlplane $
```

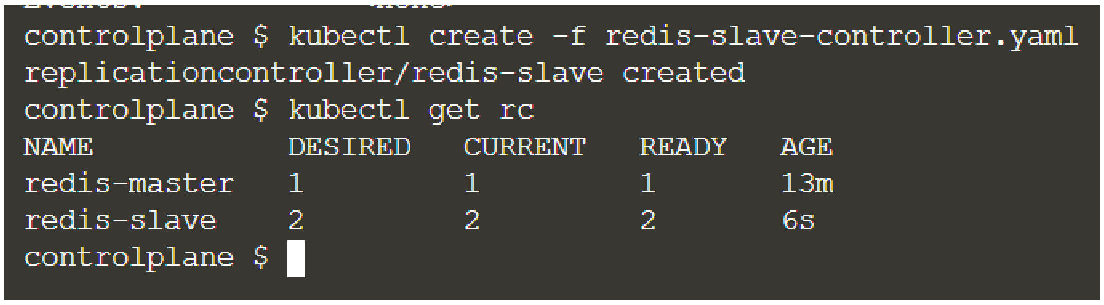
Step 4 - Replication Slave Pods
In this example we'll be running Redis Slaves which will have replicated data from the master. More details of Redis replication can be found at http://redis.io/topics/replication

As previously described, the controller defines how the service runs. In this example we need to determine how the service discovers the other pods. The YAML represents the GET_HOSTS_FROM property as DNS. You can change it to use Environment variables in the yaml but this introduces creation-order dependencies as the service needs to be running for the environment variable to be defined.

Start Redis Slave Controller
In this case, we'll be launching two instances of the pod using the image kubernetes/redis-slave:v2. It will link to redis-master via DNS.

kubectl create -f redis-slave-controller.yaml

List Replication Controllers
kubectl get rc



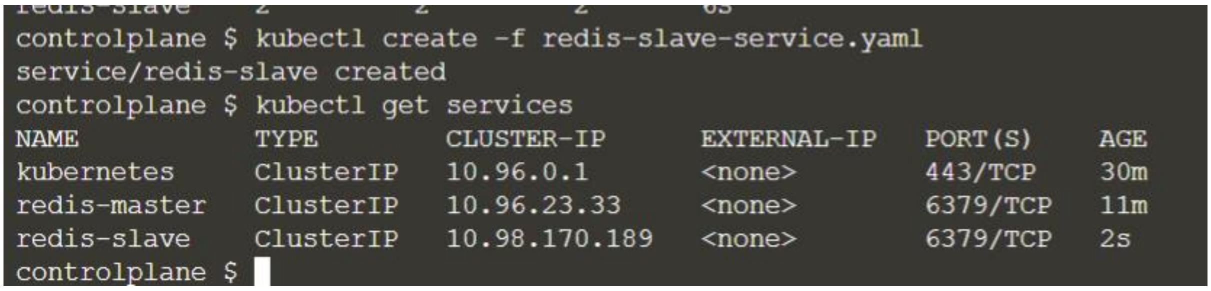
```
controlplane $ kubectl create -f redis-slave-controller.yaml
replicationcontroller/redis-slave created
controlplane $ kubectl get rc
NAME           DESIRED   CURRENT   READY   AGE
redis-master   1         1         1       13m
redis-slave    2         2         2       6s
controlplane $
```

Step 5 - Redis Slave Service
As before we need to make our slaves accessible to incoming requests. This is done by starting a service which knows how to communicate with redis-slave.

Because we have two replicated pods the service will also provide load balancing between the two nodes.

Start Redis Slave Service
kubectl create -f redis-slave-service.yaml

kubectl get services



```
controlplane $ kubectl create -f redis-slave-service.yaml
service/redis-slave created
controlplane $ kubectl get services
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP    30m
redis-master   ClusterIP   10.96.23.33     <none>        6379/TCP   11m
redis-slave    ClusterIP   10.98.170.189   <none>        6379/TCP   2s
controlplane $
```

Step 6 - Frontend Replicated Pods
With the data services started we can now deploy the web application. The pattern of deploying a web application is the same as the pods we've deployed before.

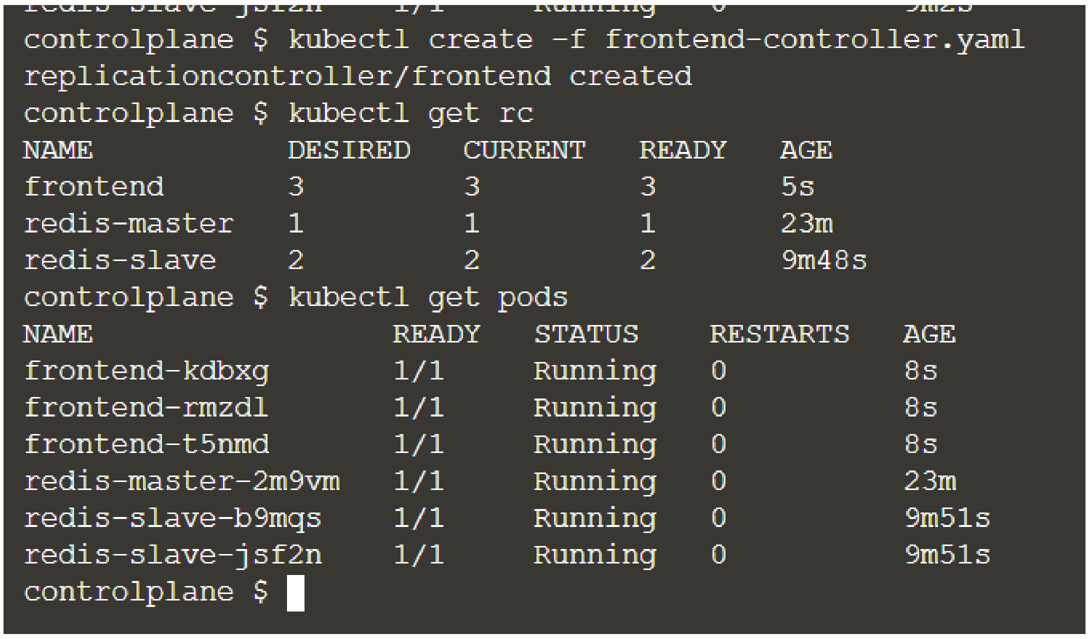
Launch Frontend
The YAML defines a service called frontend that uses the image _gcr.io/googlesamples/gb-frontend:v3. The replication controller will ensure that three pods will always exist.

kubectl create -f frontend-controller.yaml

List controllers and pods
kubectl get rc

kubectl get pods

PHP Code
The PHP code uses HTTP and JSON to communicate with Redis. When setting a value requests go to redis-master while read data comes from the redis-slave nodes.



```
controlplane $ kubectl create -f frontend-controller.yaml
replicationcontroller/frontend created
controlplane $ kubectl get rc
NAME           DESIRED   CURRENT   READY   AGE
frontend       3         3         3       5s
redis-master   1         1         1       23m
redis-slave    2         2         2       9m48s
controlplane $ kubectl get pods
NAME                 READY   STATUS    RESTARTS   AGE
frontend-kdbxg       1/1     Running   0          8s
frontend-rmzdl       1/1     Running   0          8s
frontend-t5nmd       1/1     Running   0          8s
redis-master-2m9vm   1/1     Running   0          23m
redis-slave-b9mqs    1/1     Running   0          9m51s
redis-slave-jsf2n    1/1     Running   0          9m51s
controlplane $
```

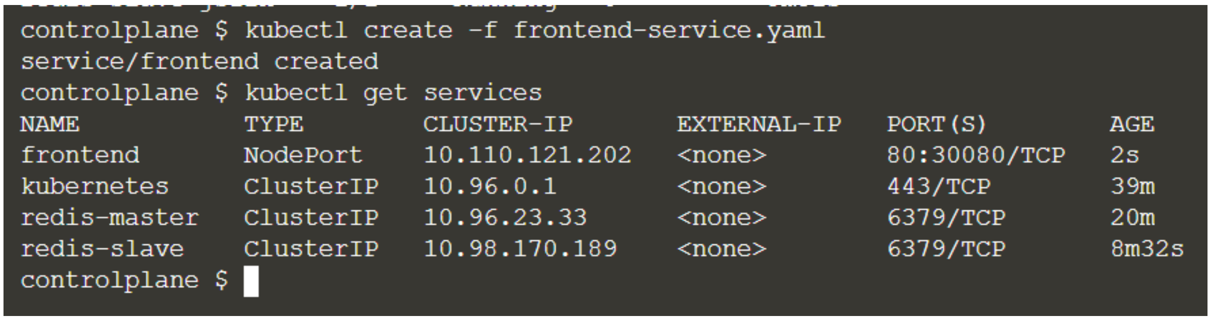
Step 7 - Guestbook Frontend Service
To make the frontend accessible we need to start a service to configure the proxy.

Start Proxy
The YAML defines the service as a NodePort. NodePort allows you to set well-known ports that are shared across your entire cluster. This is like -p 80:80 in Docker.

In this case, we define our web app is running on port 80 but we'll expose the service on 30080.

kubectl create -f frontend-service.yaml

kubectl get services



```
controlplane $ kubectl create -f frontend-service.yaml
service/frontend created
controlplane $ kubectl get services
NAME           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
frontend       NodePort    10.110.121.202   <none>        80:30080/TCP   2s
kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP        39m
redis-master   ClusterIP   10.96.23.33      <none>        6379/TCP       20m
redis-slave    ClusterIP   10.98.170.189    <none>        6379/TCP       8m32s
controlplane $
```

Step 8 - Access Guestbook Frontend
With all controllers and services defined Kubernetes will start launching them as Pods. A pod can have different states depending on what's happening. For example, if the Docker Image is still being downloaded then the Pod will have a pending state as it's not able to launch. Once ready the status will change to running.

View Pods Status
You can view the status using the following command:

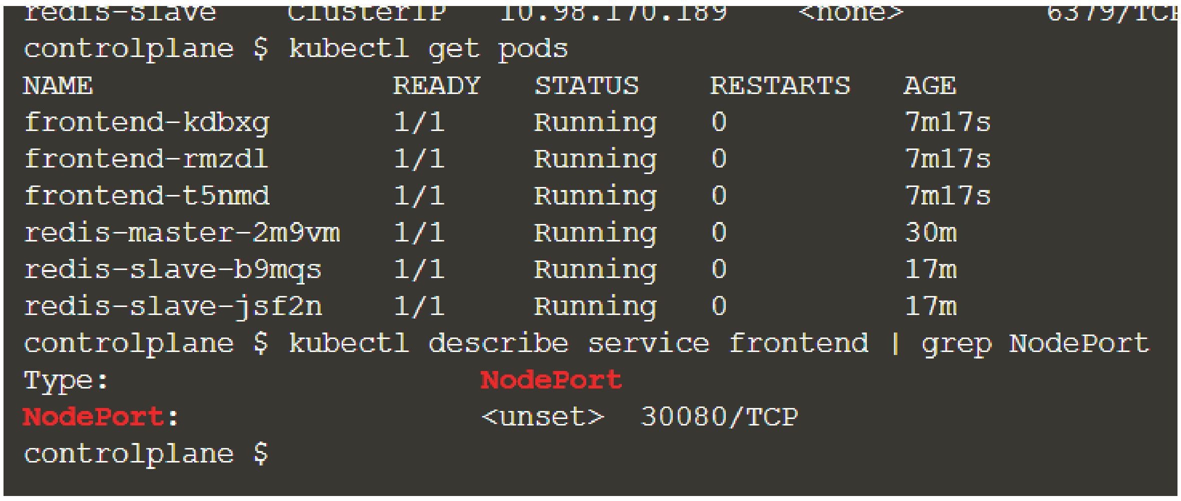
kubectl get pods

Find NodePort
If you didn't assign a well-known NodePort then Kubernetes will assign an available port randomly. You can find the assigned NodePort using kubectl.

kubectl describe service frontend | grep NodePort

View UI
Once the Pod is in running state you will be able to view the UI via port 30080. Use the URL to view the page https://6084f0cce2ce495aa49fcb12e526dd16-167772166-30080-host08nc.environments.katacoda.com

Under the covers the PHP service is discovering the Redis instances via DNS. You now have a working multi-tier application deployed on Kubernetes.



```
redis-slave    ClusterIP   10.98.170.189    <none>        6379/TC
controlplane $ kubectl get pods
NAME                 READY   STATUS    RESTARTS   AGE
frontend-kdbxg       1/1     Running   0          7m17s
frontend-rmzdl       1/1     Running   0          7m17s
frontend-t5nmd       1/1     Running   0          7m17s
redis-master-2m9vm   1/1     Running   0          30m
redis-slave-b9mqs    1/1     Running   0          17m
redis-slave-jsf2n    1/1     Running   0          17m
controlplane $ kubectl describe service frontend | grep NodePort
Type:                    NodePort
NodePort:                <unset>  30080/TCP
controlplane $
```