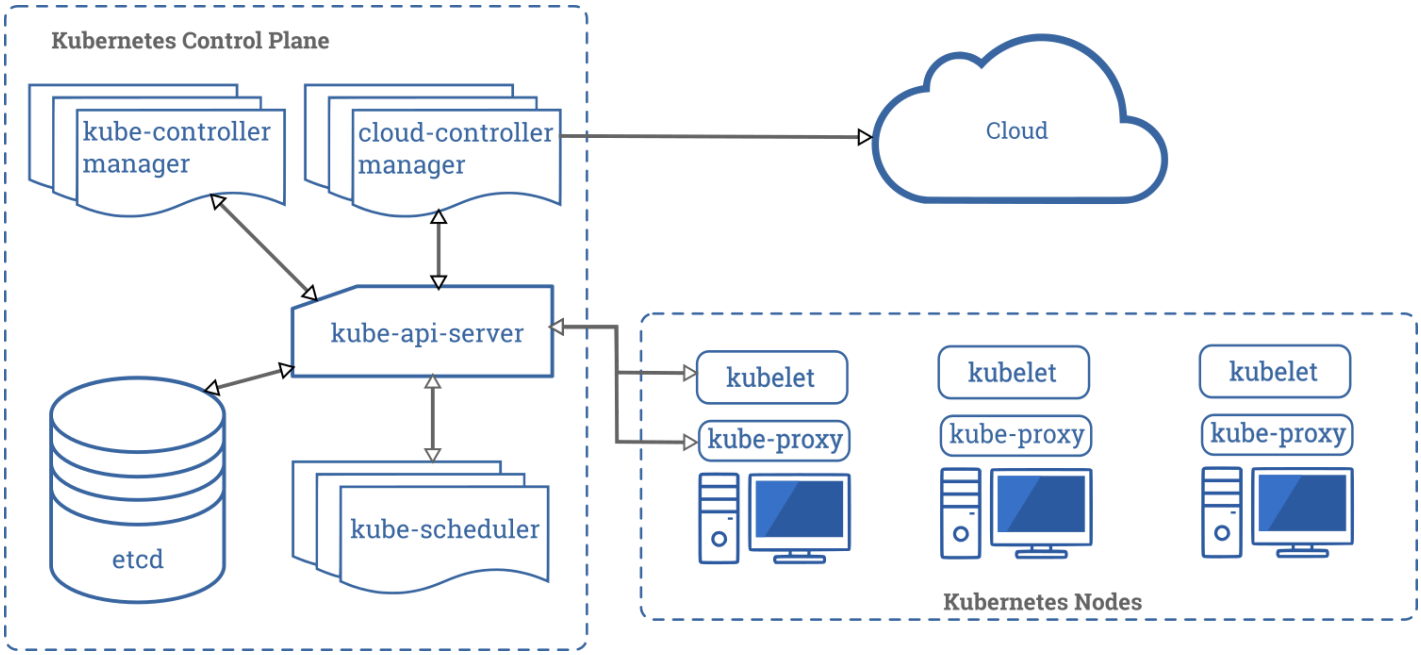
Core Concepts

Kubernetes — это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию. У платформы есть большая, быстро растущая экосистема. Сервисы, поддержка и инструменты Kubernetes широко доступны.
Цель Kubernetes заключается в автоматизированном размещении ваших приложений в виде контейнеров.
Чтобы вы могли легко развернуть столько экземпляров вашего приложения, сколько необходимо, и легко обеспечить взаимодействие между различными службами внутри вашего приложения.
Kubeadm - это инструмент, созданный для обеспечения kubeadm init и kubeadm join как лучших практик "быстрого пути" для создания кластеров Kubernetes. kubeadm выполняет действия, необходимые для запуска минимально жизнеспособного кластера. По своему дизайну он заботится только о загрузке, а не о provisioning машин.
Daemon Sets
So far we have deployed various pods on different nodes in our cluster with the help of replica sets and deployments. We made sure multiple copies of our applications are made available across various different worker nodes.
Demon sets are like replica sets, as in it helps you deploy multiple instances of pods, but it runs one copy of your pod on each node in your cluster:
Whenever a new node is added to the cluster. A replica of the pod is automatically added to that node. And when a node is removed, the pod is automatically removed. ***The demon set ensures that one copy of the pod is always present in all nodes in the cluster.***

Components cluster:



The Kubernetes cluster consists of a set of nodes which may be physical or virtual on-premise or on cloud that host applications in the form of containers.

It doesn't always have to be Docker. Kubernetes supports other run time engines as well like ContainerD or Rocket.

Master NODE

The master node in the Kubernetes cluster the master node is responsible for managing the Kubernetes cluster storing information about the different nodes planning which containers cause where monitoring the notes and containers on them etc. The Master node does all of these - using a set of components together known as the control plane components.

etcd

https://github.com/kodekloudhub/certified-kubernetes-administrator-course/blob/master/docs/02-Core-Concepts/03-ETCD-For-Beginners.md

Etcd is a database that stores information in a key-value format.

It is a distributed, reliable, key value store that is simple, secure and fast.

You get the key and it returns the value and you cannot have duplicate keys.

It is used to store and retrieve small chunks of data such as configuration data that requires fast read and writes.

- It starts a service that listens on **port 2379** by default.
- The ETCD datastore stores information regarding the cluster, such as:
  - Nodes
  - PODs
  - Configs
  - Secrets

    - Accounts
    - Roles
    - Bindings
    - Others
- Every information you see when you run the kubectl get command is from the ETCD server. Every change you make to your cluster, such as adding additional nodes, deploying pods or replica sets are updated in the ETCD server.
- Depending on how you setup your cluster, ETCD is deployed differently.
    - There are two types of kubernetes deployment. One deployed from scratch (с нуля), and other using kubeadm tool. The practice test environments are deployed using the kubeadm tool and later we set up a cluster we set it up from scratch so it's good to know the difference between the two methods
        - Если вы создаете кластер с нуля, то вы развертываете ETCD, самостоятельно загружая исполняемые файлы ETCD, устанавливая их и настраивая ETCD как службу на главном узле. Существует множество опций, передаваемых в службу, некоторые из них связаны с сертификатами. Остальные опции касаются настройки ETCD как кластера. Единственная опция, на которую стоит обратить внимание, это advertised client url.
          Это адрес, по которому ETCD слушает. Он находится на IP-адресе сервера и на порту 2379, который является портом по умолчанию, на котором прослушивается etcd. Это URL, который должен быть настроен на сервере kube-api, когда он пытается связаться с сервером etcd.

        - Если вы настроили свой кластер с помощью kubeadm, то kubeadm развернет для вас сервер ETCD как POD в пространстве имен kube-system.
          Вы можете исследовать базу данных etcd с помощью утилиты etcdctl в этом pod. Чтобы получить список всех ключей, хранящихся в kubernetes, выполните команду etcdctl get.
          Kubernetes хранит данные в определенной структуре каталогов, **корневым каталогом является registry**, а под ним находятся различные конструкции kubernetes, такие как minions или nodes, pods, replicasets, deployments и т.д.
          В среде высокой доступности у вас будет несколько основных узлов в кластере, а затем вы будете иметь несколько экземпляров ETCD, распределенных между основными узлами. В этом случае обязательно укажите, что экземпляры ETCD знают друг о друге, задав нужный параметр в конфигурации службы ETCD. Параметр initial-cluster - это то место, где вы должны указать различные экземпляры службы ETCD.

kube-scheduler

https://github.com/kodekloudhub/certified-kubernetes-administrator-course/blob/master/docs/02-Core-Concepts/07-Kube-Scheduler.md

Определяет, на какой ноде нужно разместить контейнер, основываясь на данных контейнеров. Требования к ресурсам емкость рабочих узлов или любые другие политики или ограничения, такие как требования к контейнерам и допуски или правила близости узлов, которые находятся на них.

На самом деле он не размещает капсулы на узлах. Это работа kubelet. Kubelet "или капитан на корабле" - это тот, who creates the pod on the nodes!

Kube-scheduler только решает, какая POD куда отправится. Давайте рассмотрим, как планировщик делает это немного подробнее. Прежде всего, зачем нужен планировщик? Когда есть много кораблей и много контейнеров, Вы хотите быть уверены, что нужный контейнер окажется на нужном корабле. Например, корабли и контейнеры могут быть разных размеров. Вы хотите убедиться, что корабль имеет достаточную вместимость для размещения этих контейнеров. могут направляться в разные пункты назначения. Вы хотите убедиться, что ваши контейнеры размещены на правильных кораблях, чтобы они попали в нужное место назначения. В kubernetes планировщик решает, на каких узлах размещать PODs в зависимости от определенных критериев. У вас могут быть POD с различными требованиями к ресурсам, У вас могут быть dones в кластере, предназначенные для определенных приложений. Как же планировщик назначает эти POD?

Планировщик рассматривает каждый POD и пытается найти лучший узел для него.

Например, возьмем один из POD. Допустим, у него есть набор требований к процессору и памяти (10 CPU). Планировщик проходит через две фазы, чтобы определить лучшую ноду для POD на первой фазе. Планировщик пытается отфильтровать ноды, которые не соответствуют профилю для этого блока.

Например, ноды, которые не обладают достаточными ресурсами процессора и памяти, запрашиваемыми POD. Поэтому первые две небольших ноды (4 and 8 CPU) отфильтровываются.

Теперь у нас осталось дву ноды, на которых POD может быть размещен (12 and 16 CPU).

Теперь, как планировщик выбирает одну из двух NODEs, планировщик ранжирует ноды, чтобы определить наиболее подходящую для размещения POD. Он использует функцию приоритета для присвоения нодам оценки по шкале от 0 до 10.

Например, планировщик вычисляет количество ресурсов, которые будут свободны на нодах после размещения на них POD.

В данном случае у ноды (16 CPU) будет свободно 6 процессоров, если разместить на нем POD, что на 4 процессора больше. чем у другой ноды, поэтому она получает более высокий рейтинг. И поэтому она выигрывает. Вот как работает планировщик на высоком уровне.

Controller-Manager

https://github.com/kodekloudhub/certified-kubernetes-administrator-course/blob/master/docs/02-Core-Concepts/06-Kube-Controller-Manager.md

Responsible for onboarding new nodes to the cluster handling situations where nodes become unavailable or get gets destroyed and the replication controller ensures that the desired number of containers are running at all

times in your replication.

- Контроллер - это как офис или отдел внутри главного корабля. которые имеют свой собственный набор обязанностей. Например, офис для кораблей будет отвечать за мониторинг и принятие необходимых мер в отношении кораблей.
  Когда прибывает новое судно или когда судно уходит или уничтожается, другой офис может быть тем, который управляет контейнерами на кораблях, заботится о поврежденных или переполненных контейнерах. Таким образом, эти офицеры, во-первых, постоянно следят за состоянием кораблей и, во-вторых, принимают необходимые меры для исправления ситуации. необходимые действия для исправления ситуации.
- В терминах kubernetes контроллер - это процесс, который постоянно отслеживает состояние различных компонентов в системе и работает над приведением всей системы к желаемому состоянию функционирования.
  Например, node controller отвечает за мониторинг состояния nodes и предпринимает необходимые действия для поддержания работы приложения. Он делает это через сервер kube-api.
  Node Controller проверяет состояние узлов каждые 5 секунд! Таким образом, node controller может отслеживать состояние узлов, если он перестает получать heartbeat, то node помечается как недоступный, но он ждет 40 секунд, прежде чем пометить его как недоступный. После того, как узел помечен как недоступный, он дает ему пять минут на восстановление, если этого не происходит, он удаляет POD, назначенные этой ноде и размещает их на здоровых нодах!

  Если PODs являются частью набора реплик, следующим контроллером является replication controller.
  Он отвечает за мониторинг состояния of replica sets и обеспечение того, чтобы необходимое количество POD доступны в любое время в пределах набора. If a pod dies it creates another one.
- В kubernetes существует гораздо больше таких контроллеров. Это всего лишь два примера выше. Все концепции в kubernetes, такие как развертывания, службы, пространства имен, постоянные тома и все интеллектуальные возможности, встроенные в эти конструкции, реализуются через эти различные контроллеры!
  Это своего рода мозг, стоящий за многими вещами в kubernetes.

- Все эти контроллеры упакованы в один процесс, известный как kubernetes controller manager. Когда вы устанавливаете kubernetes controller manager, различные контроллеры также устанавливаются. Когда вы загрузите, распакуете и запустите его как службу, вы сможете увидеть, там есть список опций, где вы предоставляете дополнительные опции для настройки вашего контроллера.
  Помните некоторые настройки по умолчанию для контроллера узлов, которые мы обсуждали ранее, такие как период мониторинга узла период, льготный период и таймаут выселения. Эти параметры добавляются сюда в качестве опций.
  Есть дополнительная опция под названием controllers, которую можно использовать, чтобы указать, какие контроллеры включить. По умолчанию все они включены, но вы можете выбрать включение некоторых из них.
  Поэтому в случае, если какие-то из ваших контроллеров не работают или не существуют, это будет хорошей отправной точкой, чтобы посмотреть.

- Как посмотреть опции сервера Kube-controller-manager?
  Это зависит от того, как вы настроили свой кластер. Если вы настраиваете его с помощью инструмента kubeadm, kubeadm развертывает kube-controller-manager как pod в пространстве имен kube-system на master node. Вы можете увидеть опции в файле определения pod, расположенном в папке etc kubernetes manifests.

  В случае установки без kubeadm, вы можете просмотреть опции, просмотрев службу kube-controller-manager, расположенную в каталоге services. Вы также можете увидеть запущенный процесс и эффективные опции, перечислив процесс на главном узла и поискав kube-controller-manager.

kube-api-server

https://github.com/kodekloudhub/certified-kubernetes-administrator-course/blob/master/docs/02-Core-Concepts/05-Kube-API-Server.md

The kube-apiserver is the primary management component of kubernetes. The kube-api server is responsible for orchestrating all operations within the cluster.

- Сервер Kube-api является основным компонентом управления в kubernetes. Когда вы выполняете команду kubectl, утилита kubectl фактически обращается к kube-apiserver. Сервер kube-api сначала проверяет подлинность запроса и подтверждает его. Затем он извлекает данные из кластера ETCD и отвечает запрошенной информацией.
  На самом деле вам не нужно использовать командную строку kubectl. Вместо этого вы можете обратиться к API напрямую, отправив пост-запрос, как в этом примере: `curl -x POST /api/v1/namespaces/default/pods...[other]`

- Пример создания pod, когда вы делаете это, как и раньше:
  Запрос сначала аутентифицируется, а затем проверяется. В этом случае сервер API создает a POD object without assigning it to a node, updates the information in the ETCD server updates the user that the POD has been created. Планировщик непрерывно следит за сервером API и понимает, что появился новый pod без назначенной ноды. Планировщик определяет нужную ноду, на которую нужно поместить новый POD, и сообщает об этом обратно kube-apiserver.
  Затем сервер API обновляет информацию в кластере ETCD. The API server then passes that information to the kubelet in appropriate worker node. Kubelet создает POD на узле и дает команду механизму выполнения контейнера to deploy the application image.
  После этого kubelet обновляет статус на API-сервере, а API-сервер обновляет данные в кластере ETCD.

  Аналогичная процедура выполняется каждый раз, когда запрашивается изменение. Kube-apiserver находится в центре всех различных задач, которые необходимо выполнить для внесения изменений в кластере.

- Подводя итог, можно сказать, что сервер kube-api отвечает за аутентификацию и проверку запросов, получение и обновление данных в хранилище данных ETCD, Фактически, kube-api сервер является единственным компонентом, который напрямую взаимодействует с хранилищем данных etcd. Другие компоненты, такие как планировщик, kube-controller-manager и kubelet используют API-сервер для выполнения обновлений в кластере в соответствующих областях.

Worker NODEs

Host application as Containers

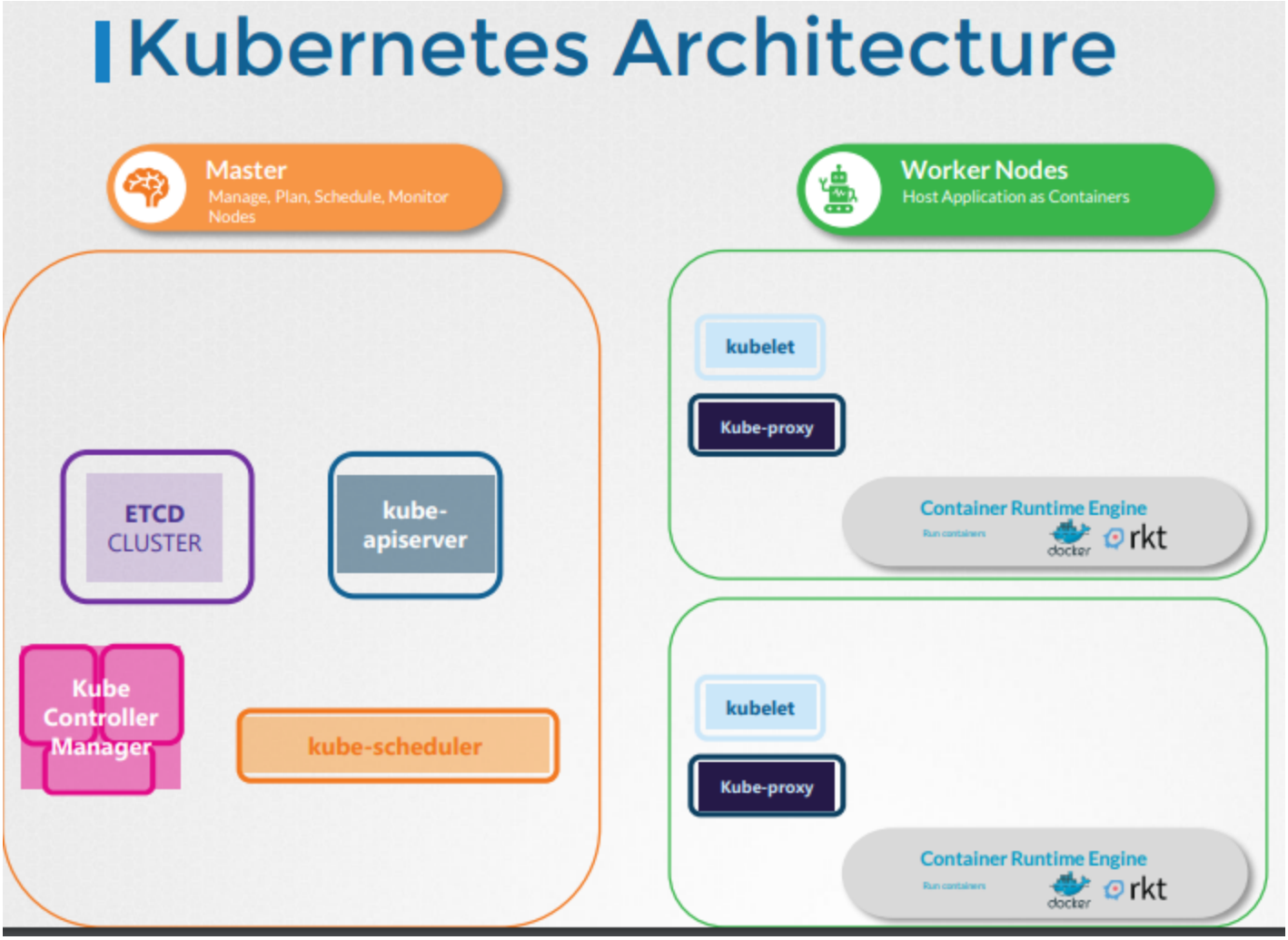
kubelet

A kubelet is an agent that runs on each node in a cluster.
It listens for instructions from the kube-api server and deploys or destroys containers on the nodes as required.

kube-proxy

https://github.com/kodekloudhub/certified-kubernetes-administrator-course/blob/master/docs/02-Core-Concepts/09-Kube-Proxy.md
The Kube-proxy service ensures that the necessary rules are in place on the worker nodes to allow the containers running on them to reach each other. (конфигурирует правила сети на узлах. При помощи них разрешаются сетевые подключения к вашими подам изнутри и снаружи кластера.)

Это достигается путем развертывания в кластере сетевого решения POD. Сеть POD - это внутренняя виртуальная сеть, охватывающая все узлы кластера, к которой подключаются все POD. Через эту сети они могут общаться друг с другом.
Существует множество решений для развертывания такой сети. В данном случае, допустим, есть веб-приложение, развернутое на первой ноде, и приложение базы данных, развернутое на второй. Веб-приложение может получить доступ к базе данных, просто используя IP-адрес POD базы данных. Но нет никакой гарантии, что IP части базы данных всегда будет оставаться одним и тем же. Лучший способ для веб-приложения получить доступ к базе данных - это использование service! Поэтому мы создаем service для предоставления доступа к базе данных приложению на кластере. Теперь веб-приложение может получить доступ к базе данных, используя имя service db. Service также получает IP-адрес, назначенный ему, когда какой-либо модуль пытается получить доступ к сервису, используя его IP-адрес или имя. Он перенаправляет трафик на конечный модуль. В данном случае на базу данных. Но что это за сервис и как он получает IP? Присоединяется ли сервис к той же сети POD?
Сервис не может присоединиться к сети pod, потому что сервис - это не реальная вещь. Это не контейнер, как pod, поэтому у него нет никаких интерфейсов или активно прослушивающего процесса. Это виртуальный компонент, который живет только в шкафу в виде памяти. Но затем мы также сказали, что сервис должен быть доступен во всем кластере из любой точки. Как же это достигается?
Здесь на помощь приходит kube-proxy. Kube-proxy - это процесс, который запускается на каждом ноде кластера kubernetes. Его работа заключается в поиске новых сервисов, и каждый раз, когда создается новый сервис, он создает соответствующие правила на каждой ноде для перенаправления трафика к этим сервисам на внутренние модули.
Один из способов сделать это - использовать правила IPTABLES. Например, создается правило IP таблиц на каждой ноде кластера для перенаправления трафика на IP службы, который является 10.96.0.12, на IP фактического pod, который является 10.32.0.15.



PODs

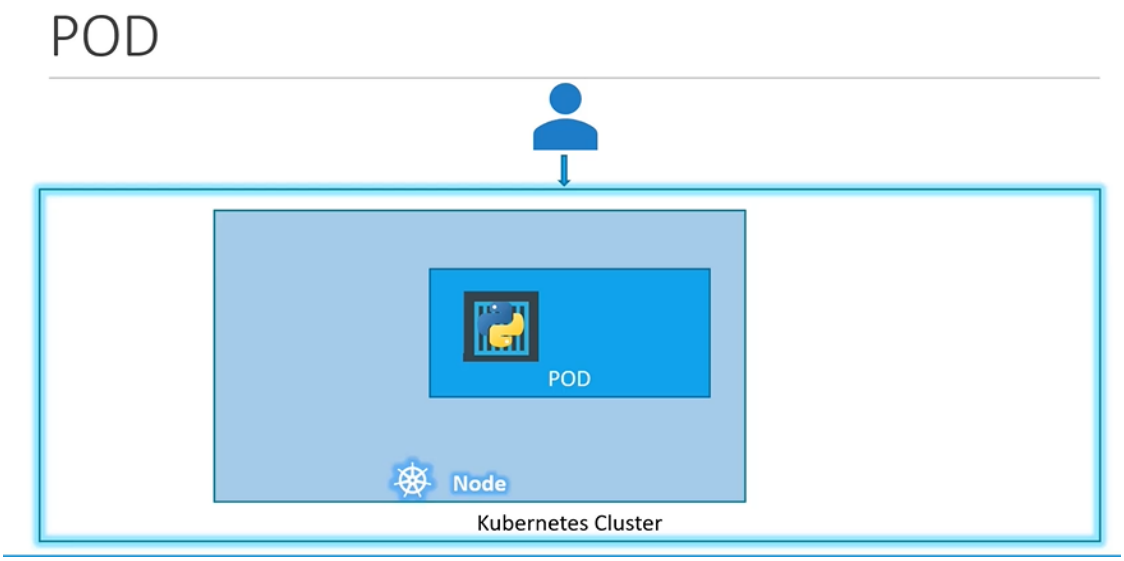
https://github.com/kodekloudhub/certified-kubernetes-administrator-course/blob/master/docs/02-Core-Concepts/10-Pods.md

Kubernetes не развертывает контейнеры непосредственно на worker nodes.

Контейнеры инкапсулируются в объект Kubernetes, известный как pods.

- POD - это один экземпляр приложения.
- POD - это самый маленький объект, который можно создать в Kubernetes.

  Здесь мы видим простейший случай, когда у вас есть одна нода, кластер Kubernetes с одним экземпляром нашего приложения, запущенный в одном контейнере Docker, заключенном в POD.



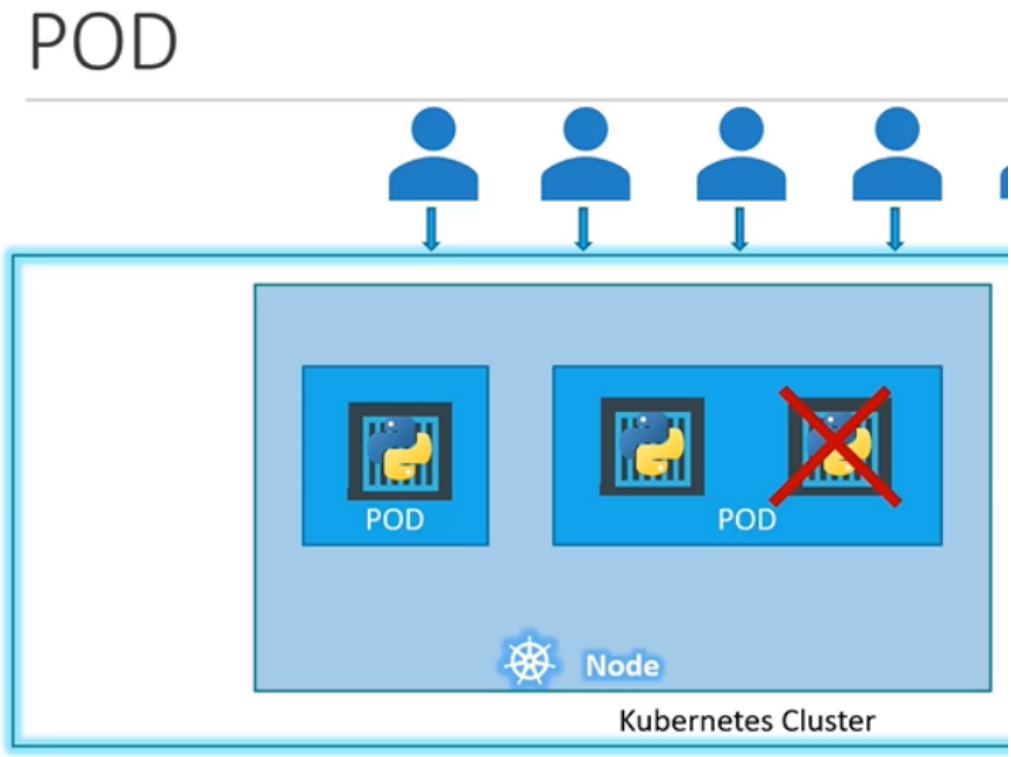
What if the number of users accessing your application increase and you need to scale your application, you need to add additional instances of your web application to share the load.

Now, where would you spin up additional instances?

Do we bring up new container instance within the same pod?

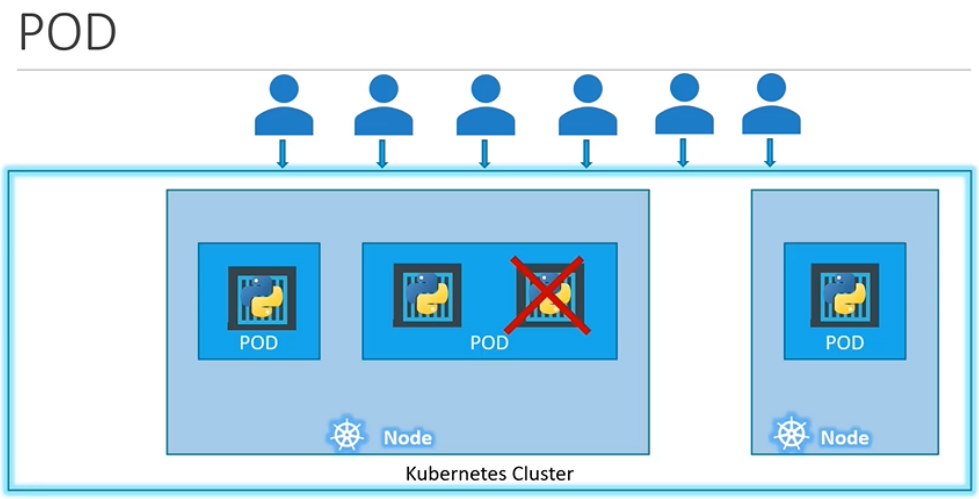
No, we create new pod altogether with a new instance of the same application.

As you can see, we now have two instances of our web application running on two separate pods on the same Kubernetes system or node.



What if the user base further increases and your current node has no sufficient capacity?

Well, then you can always deploy additional pods on a new node. In the cluster, you will have a new node added to the cluster to expand the clusters physical capacity.



So what I'm trying to illustrate in this slide is that pods usually have a 1 to 1 relationship with containers running your application.

To scale up, you create new pods and to scale down you delete existing pod. You do not add additional containers to an existing pod to scale your application.

Но ограничиваемся ли мы одним контейнером в одном POD?

НЕТ.

Иногда может возникнуть сценарий, когда у вас есть вспомогательный контейнер, который может выполнять какую-то вспомогательную задачу для нашего веб-приложения, такую как обработка пользователя, ввод данных, обработка файла, загруженного пользователем, и т.д., и вы хотите, чтобы эти вспомогательные контейнеры жили рядом с вашим приложением.

Общий контекст Pod - это набор пространств имен Linux, cgroups и, возможно, другие аспекты изоляции - то же самое, что изолирует контейнер Docker. Внутри контекста Pod отдельные приложения могут иметь дополнительные суб-изоляции.

С точки зрения концепций Docker, Pod похож на группу контейнеров Docker с общими пространствами имен и общими томами файловой системы.

YAML in Kubernetes

A Kubernetes user or administrator specifies data in a YAML file, typically to define a Kubernetes object. The YAML configuration is called a “manifest”, and when it is “applied” to a Kubernetes cluster, Kubernetes creates an object based on the configuration.

A Kubernetes Deployment YAML specifies the configuration for a Deployment object—this is a Kubernetes object that can create and update a set of identical pods. Each pod runs specific containers, which are defined in the spec.template field of the YAML configuration.

В файле .yaml создаваемого объекта Kubernetes необходимо указать значения для следующих полей:

- **apiVersion** — используемая для создания объекта версия API Kubernetes
- **kind** — тип создаваемого объекта (pod, replica set, deployment or service)
- **metadata** — данные, позволяющие идентифицировать объект (name, UID и необязательное поле namespace)
- **spec** — Здесь мы предоставляем Kubernetes дополнительную информацию, относящуюся к этому объекту. Это словарь, поэтому добавьте под него свойство containers. Containers - это список или массив. Причина, по которой это свойство является списком, заключается в том, что в капсулах может быть несколько контейнеров.

The Deployment object not only creates the pods but also ensures the correct number of pods is always running in the cluster, handles scalability, and takes care of updates to the pods on an ongoing basis. All these activities can be configured through fields in the Deployment YAML.

```
# Install pod from yaml:
kubectl create -f pod-definition.yml

# Create a new pod with the nginx image.
kubectl run nginx --image=nginx

# List of Pods:
kubectl get pods

# Describe Pod:
kubectl describe pod myapp-pod

# Example yaml file:
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
    tier: frontend
spec:
  containers:
  - name: nginx
    image: nginx
  - busybox
    image: busybox

# Delete the webapp Pod:
kubectl delete pod webapp

# Use kubectl run command with --dry-run=client -o yaml option to create a manifest file
kubectl run redis --image=redis123 --dry-run=client -o yaml > redis-definition.yaml

# Change pod configuration:
kubectl edit pod redis

# Apply changes to pod:
```

```
kubectl apply -f redis-definition.yaml
or
vim redis-definition.yaml
```

Some TIPS with yaml:

Create an NGINX Pod

`kubectl run nginx --image=nginx`

Generate POD Manifest YAML file (-o yaml). Don't create it(--dry-run)

`kubectl run nginx --image=nginx --dry-run=client -o yaml`

Create a deployment

`kubectl create deployment --image=nginx nginx`

Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)

`kubectl create deployment --image=nginx nginx --dry-run=client -o yaml`

Generate Deployment YAML file (-o yaml). Don't create it(--dry-run) with 4 Replicas (--replicas=4)

`kubectl create deployment --image=nginx nginx --dry-run=client -o yaml > nginx-deployment.yaml`

Save it to a file, make necessary changes to the file (for example, adding more replicas) and then create the deployment.

`kubectl create -f nginx-deployment.yaml`

OR

In k8s version 1.19+, we can specify the --replicas option to create a deployment with 4 replicas.

`kubectl create deployment --image=nginx nginx --replicas=4 --dry-run=client -o yaml > nginx-deployment.yaml`

ReplicaSets

https://github.com/kodekloudhub/certified-kubernetes-administrator-course/blob/master/docs/02-Core-Concepts/13-ReplicaSets.md

The replication controller helps us run multiple instances of a single part in the Kubernetes cluster, that is providing high availability.

So does that mean you can't use a replication controller if you plan to have a single pod?
NO!!!

> Even if you have a single pod, the replication controller can help by automatically bringing up a new pod when the existing one fails.

Еще одна причина, по которой нам нужен replication controller, - это создание нескольких частей приложения для распределения нагрузки между ними.

Например, у нас есть один pod, обслуживающий набор пользователей. Когда количество пользователей увеличивается, мы развертываем дополнительный pod, чтобы сбалансировать нагрузку между двумя pod.

Если спрос еще больше возрастет и если у нас закончатся ресурсы на первой ноде - мы можем развернуть дополнительные части на других нодах кластера.

Это значит, что replication controller охватывает несколько узлов в кластере.

Example create Replication Controller (old method):

```
# rc-definition.yaml:
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: front-end
spec:
  template:

    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
      spec:
        containers:
        - name: nginx-container
          image: nginx

  replicas: 3

# Run this file:
kubectl create -f rc-definition.yaml

# Get all exists Replication Controllers:
kubectl get replicationcontroller

# We got 3 pods running:
kubectl get pods
```

Example create ReplicaSet:

```
kubectl explain replicaset | grep VERSION
```

```
apiVersion: apps/v1 # нужна определенная версия API Kubernetes, что имеет поддержку ReplicaSet
kind: ReplicaSet
metadata:
  name: myapp-replicaset
```

```
  labels:
    app: myapp
    type: front-end
spec:
 template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
     containers:
     - name: nginx-container
       image: nginx
 replicas: 3
 selector:
   matchLabels:
    type: front-end
```

There is one major difference between replication controller and replica set. **Replica set requires a selector definition.**
The selector section helps the replica set identify what pods fall under it!

- To Create the replicaset:

```
kubectl create -f replicaset-definition.yaml
```

- To list all the replicaset

```
kubectl get replicaset
```

- To list pods that are launch by the replicaset

```
kubectl get pods
```

В кластере могут быть сотни других pod, выполняющих различные приложения. Вот где пригодится маркировка наших pod во время создания.

```
 selector:
   matchLabels:
    type: front-end
```

Мы можем предоставить эти метки в качестве фильтра для набора реплик.
В разделе selector мы используем фильтр **match labels** и предоставляем ту же метку, которую мы использовали при создании pod. Таким образом, набор реплик знает, какие pods нужно отслеживать.

How to scale replicaset?

There are multiple ways to scale replicaset

1. First way is to update the number of replicas in the replicaset-definition.yaml definition file. E.g replicas: 6 and then run

```
replicas: 6
```

and run:

```
kubectl apply -f replicaset-definition.yaml
```

2. Second way is to use **kubectl scale** command.

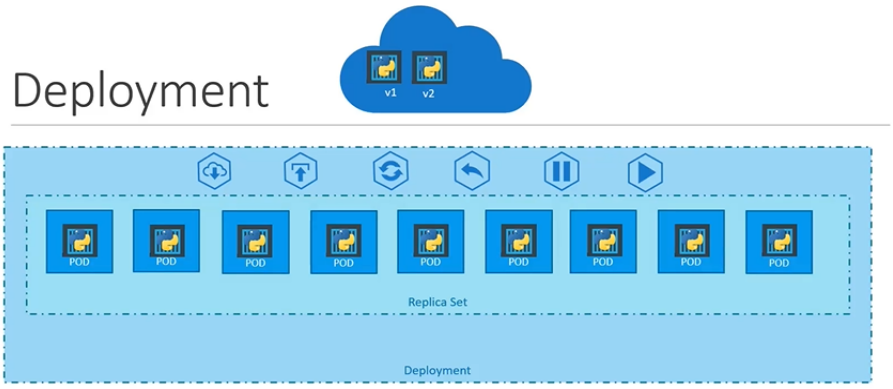
```
kubectl scale --replicas=6 -f replicaset-definition.yaml
or
kubectl scale rs new-replica-set --replicas=5
```

3. Third way is to use kubectl scale command with type and name

```
kubectl scale --replicas=6 replicaset myapp-replicaset
```

Deployments

https://github.com/kodekloudhub/certified-kubernetes-administrator-course/blob/master/docs/02-Core-Concepts/15-Deployments.md



Допустим, у вас есть веб-сервер, который необходимо развернуть в производственной среде. По очевидным причинам вам нужен не один, а много таких экземпляров веб-сервера.
Во-вторых, каждый раз, когда в реестре docker становятся доступны новые версии сборок приложений, вы хотели бы хотели бы без проблем обновлять свои экземпляры Docker.

- Однако при обновлении экземпляров вы не хотите обновлять их все сразу. Это может повлиять на работу пользователей с нашими приложениями, поэтому лучше обновлять их по одному последовательно. Такой тип обновления известен как **rolling updates**.

- Предположим, что одно из обновлений, которое вы выполнили, привело к непредвиденной ошибке, и вас просят отменить недавнее изменение, и вы хотели бы иметь возможность откатить изменения, которые были недавно выполнены.
- Наконец, например, вы хотите внести несколько изменений в вашу среду, например, обновить базовые версии WebServer, а также масштабирование среды и изменение распределения ресурсов и т.д.
- Вы не хотите применять каждое изменение сразу после выполнения команды, вместо этого вы хотите сделать паузу в работе среды, внести изменения, а затем возобновить работу, чтобы все изменения были развернуты вместе. Все эти возможности доступны в kubernetes **Deployments**.

До этого мы обсуждали PODs, которые развертывают отдельные экземпляры нашего приложения, например, веб-приложение. Каждый контейнер инкапсулируется в PODs. Несколько таких POD развертываются с помощью replication controller или ReplicaSet.
Далее идет **Deployment**, который является объектом kubernetes, находящимся выше в иерархии.
Развертывание предоставляет нам возможность беспрепятственно обновлять базовые экземпляры с помощью rolling updates, отменять изменения, приостанавливать и возобновлять изменения по мере необходимости.

Как и в случае с предыдущими компонентами,
Сначала мы создаем файл определения развертывания. Содержимое файла определения развертывания в точности аналогично файлу определения ReplicaSet за исключением типа, который теперь будет развертыванием.

```
kind: Deployment
```

Если мы просмотрим содержимое файла в нем есть apiVersion, которая является apps/v1, метаданные, которые имеют название и метки, и спецификация, которая шаблон, реплики и селектор. Шаблон имеет внутри себя определение pod. Когда файл будет готов, выполните команду kubectl create и укажите файл определения развертывания.

- Once the file is ready, create the deployment using deployment definition file

```
kubectl create -f deployment-definition.yaml
```

- Затем выполните команду kubectl get deployments, чтобы увидеть только что созданное развертывание.

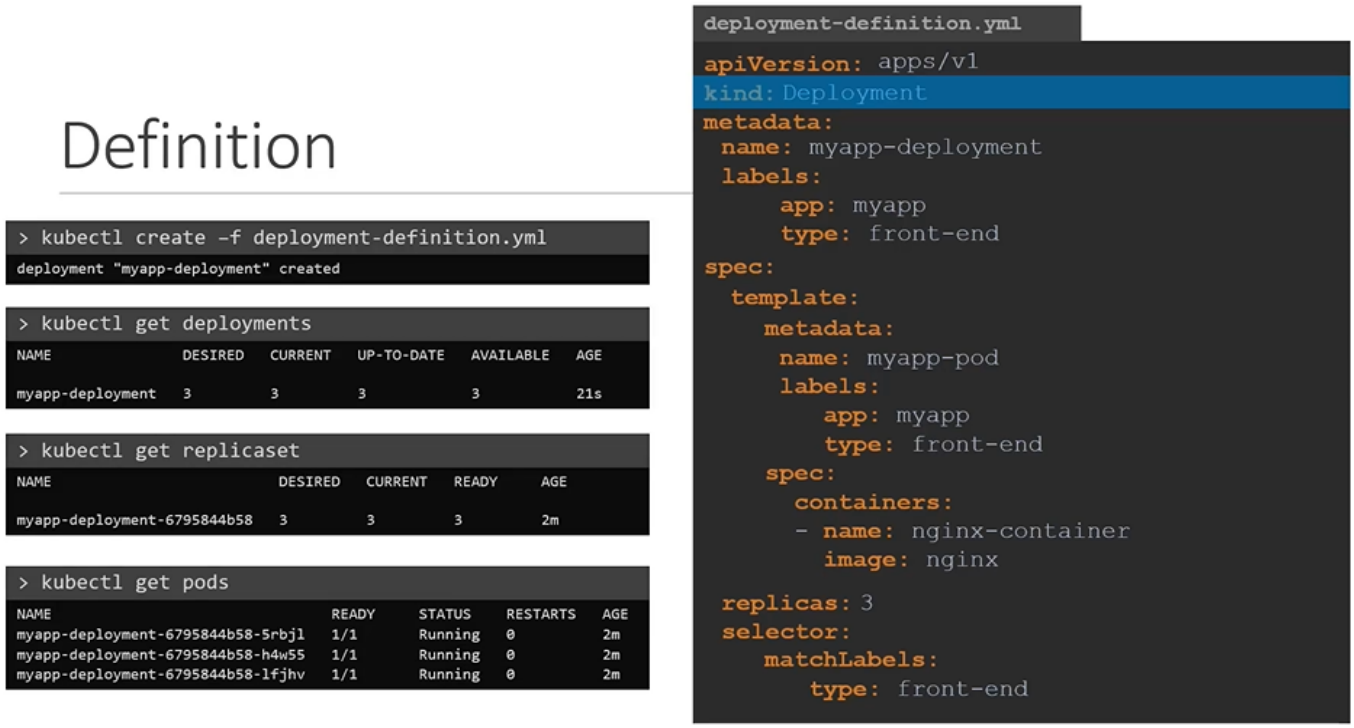
```
kubectl get deployment
```

- При развертывании автоматически создается набор реплик. Поэтому если вы выполните команду kubectl get replcaset. вы сможете увидеть новый набор реплик в имени развертывания.

```
kubectl get replicaset
```

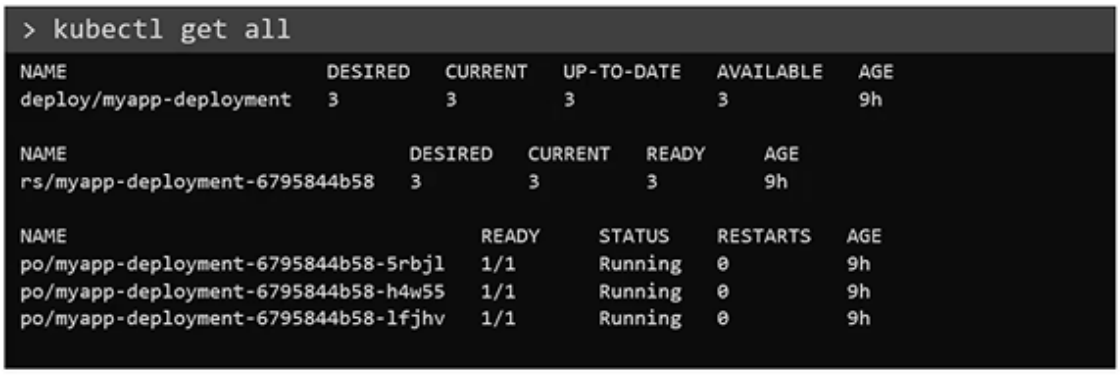
- Наборы реплик в конечном итоге создают капсулы, поэтому если вы выполните команду kubectl get pods вы сможете увидеть pods с именем развертывания и набором реплик.

```
kubectl get pods
```



- To see the all objects at once

```
kubectl get all
```



До сих пор не было особой разницы между наборами реплик и развертываниями, за исключением того, что что развертывания создали новый объект kubernetes под названием deployments.

- Set Node Affinity to the deployment to place the pods on node01 only.
  - Name: blue
  - Replicas: 3
  - Image: nginx

- NodeAffinity: requiredDuringSchedulingIgnoredDuringExecution
  - Key: color
  - value: blue

```
kubectl create deployment blue --image=nginx --replicas=3 --dry-run=client -o yaml > nginx-deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: blue
spec:
  replicas: 3
  selector:
    matchLabels:
      run: nginx
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - image: nginx
        imagePullPolicy: Always
        name: nginx
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: color
                operator: In
                values:
                - blue
```

Services

Services enable communication between various components within and outside of the application. Kubernetes services helps us connect applications together with other applications or users.

For example our application has groups of pods running various sections such as a group for serving front end load to users and other group for running back end processes and a third group connecting to an external data source.
It is services that enable connectivity between these groups of pods services enable the front end application to be made available to end users it helps communication between back end and front end pods and helps in establishing connectivity to an external data source.
Thus services enable free connection between micro services in our application!

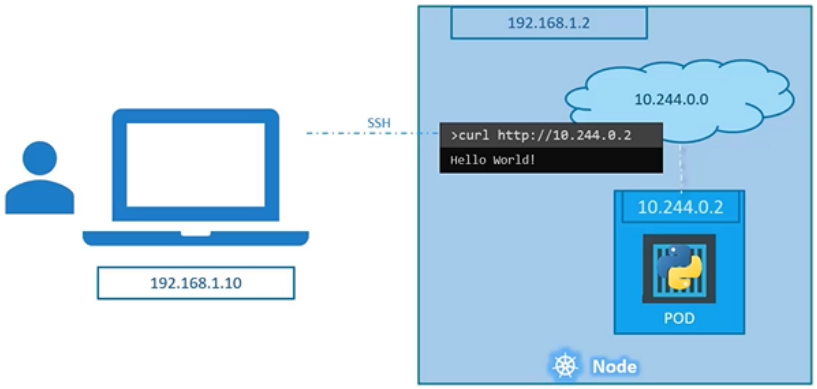
So, for example, we deployed our pod having a web application running on it. How do we as an external user access the web page?

First of all let us look at the existing setup.

The Kubernetes Node has an IP address and that is 192.168.1.2.

My laptop is on the same network as well. so it has an IP address 192.168.1.10.

The internal POD network is in the range 10.244.0.0 and the POD has an IP 10.244.0.2.



Clearly, I cannot ping or access the POD at address 10.244.0.2 as its in a separate network. So what are the options to see the webpage?

First, if we were to SSH into the kubernetes node at 192.168.1.2, from the node, we would be able to access the POD's webpage by doing a curl or if the node has a GUI, we could fire up a browser and see the webpage in a browser following the address http://10.244.0.2.

But this is from inside the kubernetes Node and that's not what I really want. I want to be able to access the web server from my own laptop without having to SSH into the node and simply by accessing the IP of the kubernetes node.

So we need something in the middle to help us map requests to the node from our laptop through the node to the POD running the web container.

This is where the kubernetes service comes into play.

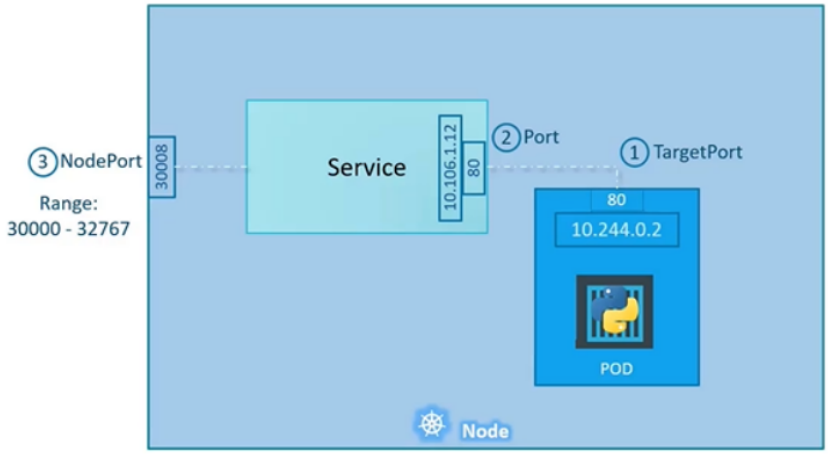
The kubernetes service is an object just like PODs, Replicasetor Deployments. One of its use case is to listen to a port on the Node and forward requests on that port to a port on the POD running the web application.

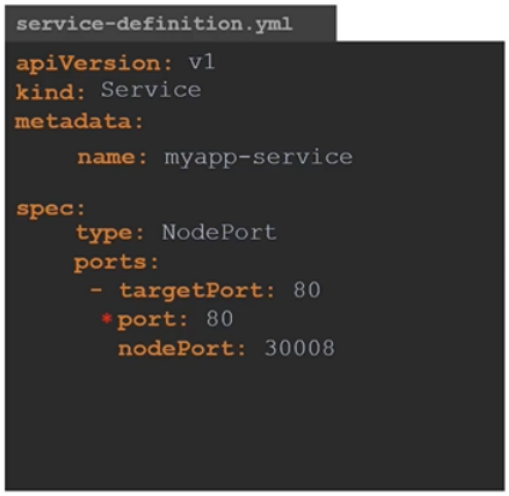
Service Types

NodePort

There are 3 types of service types in kubernetes:

```
apiVersion: v1
kind: Service
metadata:
 name: myapp-service
spec:
 types: NodePort
 ports:
 - targetPort: 80
   port: 80
   nodePort: 30008
```





If you look at it there are three ports involved.
The port on the POD were the actual web server is running is 80.
And it is referred to as the targetPort because that is were the service forwards the requests to.
The second port is the port on the service itself it is simply referred to as the port. Remember these terms are from the viewpoint of the service. The service is in fact like a virtual server inside the node inside the cluster. It has its own IP address and that IP address is called the cluster IP of the service.
And finally we have the port on the node itself which we use to access the web server externally and that is known as the node port. As you can see it is 30008.
That is because NodePorts can only be in a valid range which by default is from 30000 to 32767.
Remember that out of these the only mandatory field is port!
If we don't provide a target port - it is assumed to be the same as port.
And if you don’t provide a nodePort a free port in the valid range between 30000 and 32767 is automatically allocated.
There could be 100s of other PODs with web services running on port 80.
So how do we do that as we did with the replica sets previously?
A technique that you will see very often in kubernetes, it is uses labels and selectors to link these together.

```
apiVersion: v1
kind: Service
metadata:
 name: myapp-service
spec:
 type: NodePort
 ports:
 - targetPort: 80
   port: 80
   nodePort: 30008
 selector:
   app: myapp
   type: front-end
```

service-definition.yml

```
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
    selector:
       app: myapp
       type: front-end
```

pod-definition.yml

```
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:


spec:
  containers:
  - name: nginx-container
    image: nginx
```

This links the service to the pod.

- To create the service:

```
kubectl create -f service-definition.yaml
```

- To list the services

```
kubectl get services
```

- To access the application from CLI instead of web browser

```
curl http://192.168.1.2:30008
```

BUT!

What do you do when you have multiple PODs?

For example, in a production environment you have multiple similar pods running our web application they all have the same labels with a key app and set to a value of my app the same label is used as a selector during the creation of the service.

So when the service is created it looks for a matching pod with the label and finds three of them. The service then automatically selects all the three pods as endpoints to forward the external requests coming from the user.

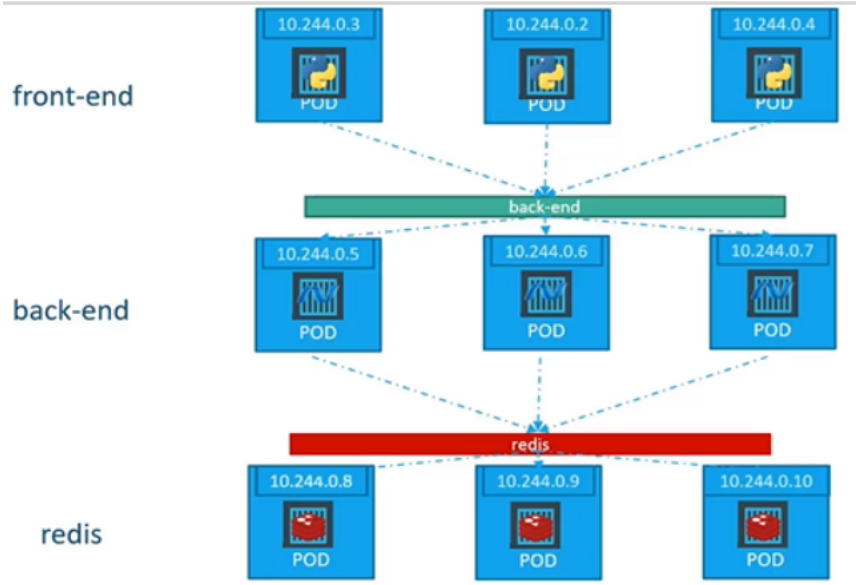
And if you're wondering what algorithm it uses to balance the load across the three different pods **it uses a random algorithm** does the service acts as a built in load balancer to distribute load across different pods.
To summarize in any case whether it be a single pod on a single node, multiple pods on a single node or multiple pods on multiple nodes - the service is created exactly the same without you having to do any additional steps during the service creation when pods are removed or added.

ClusterIP

In this case the service creates a Virtual IP inside the cluster to enable communication between different services such as a set of frontend servers to a set of backend servers.

OK. A full stack web application typically has different kinds of pods hosting different parts of an application. You may have a number of pods running a front end web server another set of pods running a back end server, a set of PODs running a key-value store like Redis, another set of PODs running a persistent database like MySQL.
The web front end server needs to communicate to the back end servers and and the backend-workers need to connect to database as well as the redis services etc..
So what is the right way to establish connectivity between these services or tiers of my application?
The pods all have an IP address assigned to them as we can see on the screen but these IP as we know are not static. These pods can go down any time and new pods are created all the time. And so you cannot rely on these IP addresses for internal communication between the application.
Also what if the first front-end POD at 10.244.0.3 need to connect to a backend service? Which of the three would it go to and who makes that decision?

A kubernetes service can help us group these PODs together and provide a single interface to access the PODs in a group.
For example a service created for the backend PODs will help group all the backend PODs together and provide a single interface for other PODs to access this service. The requests are forwarded to one of the PODs under the service randomly.
Similarly create additional services for Redis and allow the backend parts to access the redis systems through the service.
This enables us to easily and effectively deploy a microservices based application on kubernetes cluster.
Each layer can now scale or move as required without impacting communication between the various services.
Each service gets an IP name assigned to it inside the cluster and that is the name that should be used by other pods to access the service.



- To Create a Service type of ClusterIP:

```
apiVersion: v1
kind: Service
metadata:
 name: back-end
spec:
 types: ClusterIP
 ports:
 - targetPort: 80
   port: 80
 selector:
   app: myapp
   type: back-end
```

```
kubectl create -f service-definition.yaml
```

Loadbalancer

So we have seen the node port service that helps us make an external facing application available on a port on the worker nodes.

Например, у нас есть приложение для голосования и приложение для получения результатов.

Теперь мы знаем, что эти pods размещаются на рабочих nodes в кластере. Допустим, у нас есть кластер из четырех nodes. Чтобы сделать приложения доступными для внешних пользователей, мы создаем service типа node port.

Теперь services с типом node port помогают принимать трафик на порты на nodes и маршрутизировать трафик к соответствующим pods. Но какой URL вы дадите своим конечным пользователям для доступа к приложениям?

Вы можете получить доступ к любому из этих двух приложений, используя IP любого из nodes и высокий порт ,на котором открыт сервис. Таким образом, это будет комбинация IP и порта для приложения для голосования и комбинация IP и порта для приложения результатов.

Обратите внимание, что даже если ваши порты размещены только на двух nodes, они все равно будут доступны по IP из всех узлов в кластере, скажем, части для приложения для голосования развернуты только на узлах с IP 70 и 71. Они все равно будут доступны через порты всех узлов в кластере.
Вы можете предоставить эти URL своим пользователям для доступа к приложению, но это не то, что нужно конечным пользователям. Им нужен один URL, например, example, voting ABC.com или example result ABC.com для доступа к приложению. **Как же этого добиться?**
Одним из способов достижения этой цели является создание новой виртуальной машины для балансировки нагрузки, установка и настройка подходящий балансировщик нагрузки, например, proxy или engine x и т.д. Затем настройте балансировщик нагрузки на перенаправление трафик на базовые узлы. Настройка всех этих внешних балансировщиков нагрузки, а затем их поддержка и управление могут быть утомительной задачей.
Однако если бы мы работали на поддерживаемой облачной платформе, такой как Google Cloud, AWS или Azure, я мог бы использовать встроенный балансировщик нагрузки этой облачной платформы.
Kubernetes имеет поддержку для интеграции с собственными балансировщиками нагрузки определенных облачных провайдеров и настраивать и конфигурировать их для нас. Поэтому все, что вам нужно сделать, это установить тип service для фронтальных services на load balancer вместо Port Node.

```
apiVersion: v1
kind: Service
metadata:
 name: myapp-service
spec:
    type: LoadBalancer
 ports:
 - targetPort: 80
   port: 80
   nodePort: 30008
```

Теперь помните, что это работает только с поддерживаемыми облачными платформами. Так что GCP, AWS и Azure определенно поддерживаются.
Итак, если вы установите тип службы на балансировщик нагрузки. в неподдерживаемой среде, например VirtualBox или любой другой среде, то это будет иметь тот же эффект, что и установка type service: Port Node. эффект, как и установка на Port Node, где службы будут открыты на высококлассном порту на nodes. Он просто не будет выполнять какую-либо конфигурацию внешнего балансировщика нагрузки.

Namespaces

Когда кластер впервые настраивается - kubernetes создает набор pods и services для своих внутренних целей, таких как те, которые требуются сетевому решению.

названием **kube-system**.
To list the pods in kube-system namespace:

```
kubectl get pods --namespace=kube-system
```

Третье пространство имен, созданное Кубером автоматически, называется **kube-public**. Здесь создаются ресурсы, которые должны быть доступны всем пользователям.

- Check all namespaces:

```
kubectl get namespaces
```

- Create a POD in the "finance" namespace with name: "redis" and Image Name: "redis"

```
kubectl run redis --image=redis -n finance
```

When to use it?

If you wanted to use the same cluster for both **dev** and **production** environment but at the same time isolate the resources between them you can create a different namespace for each of them.

Each of these namespace can have its own set of policies that define who can do what!

Ресурсы в пространстве имен могут ссылаться друг на друга просто по своим именам. В этом случае часть веб-приложения может обратиться к службе db, просто используя имя хоста db service, если это необходимо.

```
mysql.connect("db-service")
```

Web-приложение Pod может также обращаться к службе в другом пространстве имен.
Для этого необходимо добавить имя пространства имен к имени службы.
Например, чтобы web pod в пространстве имен по умолчанию подключился к базе данных в среде dev или пространстве имен.

```
mysql.connect("db-service.dev.svc.cluster.local")
```

local - the default domain name of the kubernetes cluster
svc - the subdomain for service

- Here we have a pod definition file:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
     app: myapp
     type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

- To create the pod with the pod-definition file in another namespace, use the **--namespace** option:

```
kubectl create -f pod-definition.yaml --namespace=dev
```

- If you want to make sure that this pod gets you created in the dev env all the time, even if you don't specify in the command line, you can move the **--namespace** definition into the **pod-definition file**:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  namespace: dev
  labels:
     app: myapp
     type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

- To create a new namespace, create a namespace definition as shown below and then run **kubectl create**:

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
```

```
kubectl create -f namespace-dev.yaml
```

- Another way to create a namespace:

```
kubectl create namespace dev
```

- By default, we will be in a **default** namespace. To switch to a particular namespace permenently run the below command:

```
kubectl config set-context $(kubectl config current-context) --namespace=dev
```

- To view pods in all namespaces:

```
kubectl get pods --all-namespaces
```

Also we can create a **resource quota**. Start with a definition file for resource quota specify the namespace for which you want to create the quota and then under spec provide your limits such as 10 pods 10 CPU units 10 GB byte of memory etc..

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: dev
spec:
  hard:
    pods: "10"
```

```
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi
```

Imperative vs Declarative

Допустим, вы хотите посетить дом друга, расположенный по адресу D.
Раньше вы нанимали такси и давали водителю пошаговые инструкции о том, как добраться до места назначения. Например, направо на улицу B, затем налево на улицу C, затем еще раз налево и направо на улицу D и остановиться у дома, указав, что и как делать. Это является императивный подход.
С другой стороны, сегодня, когда вы заказываете такси, скажем, через Uber, вы просто указываете конечный пункт назначения, например, доехать до дома Тома. И в этом случае речь идет о декларативном подходе. Мы не даем пошаговых инструкций. Вместо этого мы просто указываем конечный пункт назначения.

Указывать, что делать, а не как делать - это и есть декларативный подход.

ЕЩЕ РАЗ:

- **Imperative:**

```
1. Provision a VM by the name "web-server"
2. Insatll NGINX Software on it
3. Edit configuration file to use port "8080"
4. Edit configuration file to web path "/var/www/nginx"
5. Load web pages to "/var/www/nginx" from GIT Repo - X
6. Start NGINX server
```

- **Declarative**

```
VM Name: web-server
Package: nginx
Port: 8080
Path: /var/www/nginx
Code: GIT Repo - X
```

Kubernetes should be able to read the configuration files and decide by itself what needs to be done to bring the infrastructure to the expected state.
So in the declarative approach, you will run the cube to apply command for creating, updating or deleting an object.
Imperative commands:

- kubectl run --image=nginx nginx
- kubectl create deployment --image=nginx nginx

- kubectl deployment nginx --port 80
  - kubectl edit deployment nginx
  - kubectl scale deployment nginx --replicaset=5
  - kubectl set image deployment my-deployment '*=nginx:1.13' - сделать апдейт всех имеджей в поде

NOTICE!
One way is to use the **edit** command and specify the object name.

```
kubectl edit deployment nginx
```

So when this command is run, it opens a YAML definition file similar to the one you used to create the object, but with some additional fields, such as the status fields that you see here which are used to store the status of the pod.
This is not the file you used to create the object. This is a similar pod definition file within the Kubernetes memory. You can make changes to this file and save and quit, and those changes will be applied to the live object.
However, note that there is a difference between the live object and the definition file that you have locally. The change you made using the edit command is not really recorded anywhere after the change is applied.

When the new change is applied, the **previous change to the image is lost**! So you can use the **edit** command if you are making a change and you're sure that you're not going to rely on the object configuration.

But a better approach to that is to first edit the local version of the object configuration file with the required changes. That is, by updating the image name in YAML file and then running the **replace** command to update the object.

```
kubectl replace -f nginx.yaml
```

В дальнейшем внесенные изменения записываются и могут быть отслежены в рамках процесса проверки изменений.
Поэтому иногда вам может понадобиться полностью удалить и создать объекты заново. В таких случаях вы можете выполнить ту же команду, но с опцией force, как показано ниже:

```
kubectl replace --force -f nginx.yaml
```

Это все еще императивный подход, потому что вы все еще указываете Kubernetes, **как** создать или обновить эти объекты.

```
Basic Commands (Beginner):
  create          Create a resource from a file or from stdin
  expose          Take a replication controller, service, deployment or pod and expose it as a new
Kubernetes service
  run             Run a particular image on the cluster
  set             Set specific features on objects
```

- Create a **service: redis-service** to expose the redis application within the cluster on port 6379.

```
kubectl expose pod redis --name  redis-service --type ClusterIP --port 6379
```

```
kubectl get services
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
kubernetes      ClusterIP   10.43.0.1      <none>        443/TCP    16m
redis-service   ClusterIP   10.43.249.31   <none>        6379/TCP   114s
```

```
kubectl expose pod nginx --type=NodePort --port=80 --name=nginx-service --dry-run=client -o yaml
```

- Create a deployment named webapp using the image "kodekloud/webapp-color" with 3 replicas.

```
kubectl create deployment webapp --image=kodekloud/webapp-color --replicas=3
```

--dry-run : По умолчанию, как только команда будет запущена, ресурс будет создан.
--dry-run=client : Если вы просто хотите проверить свою команду, используйте эту опцию. Это не создаст ресурс, а сообщит вам, может ли ресурс быть создан и правильна ли ваша команда.
-o yaml : Это выведет определение ресурса в формате YAML на экран.

Kubectl Apply Command

Команда Applied принимает во внимание **local configuration file**, **life object configuration** на Kubernetes и **last applied configuration**, прежде чем принять решение о том, какие изменения должны быть внесены.
Поэтому при выполнении команды apply, если объект еще не существует, объект создается!
Когда объект создан, в Kubernetes создается конфигурация объекта, аналогичная той, что мы создали локально, но с дополнительными полями для хранения статуса объекта. Это жизненная конфигурация объекта на кластере Kubernetes (life object configuration) .
Именно так Kubernetes внутренне хранит информацию об объекте, независимо от того, какой подход вы используете для создания объекта.
Но когда вы используете команду apply для создания объекта, она делает нечто большее. YAML-версия (local configuration file) объекта, который мы написали, преобразуется в JSON-формат, и он затем

сохраняется в качестве (last applied configuration) для любых обновлений объекта. Все три варианта сравниваются, чтобы определить, какие изменения должны быть сделаны на живом объекте.
Например, когда image обновлен до 1.19 в нашем local configuration file, мы выполняем команду apply. Это значение сравнивается со значением в life object configuration, и если есть разница, то life object configuration обновляется новым значением. После любого изменения last applied configuration формат JSON всегда обновляется до последнего, чтобы всегда быть актуальным.
Так зачем же нам тогда действительно нужна last applied configuration?
Так, если поле было удалено, скажем, например, была удалена label type.
И теперь, когда мы выполняем команду, мы видим, что в last applied configuration была метка, но она не присутствует в local configuration file. Это означает, что поле должно быть удалено из life object configuration.

- Если поле присутствовало в life object configuration и не присутствовало в local configuration file или last applied configuration, то оно будет оставлено как есть.
- Но если поле отсутствует в local configuration file, но присутствует в last applied configuration, это означает, что на предыдущем шаге или когда бы мы в последний раз ни выполняли команду apply, это конкретное поле было там, и теперь оно удаляется.

Таким образом, last applied configuration помогает нам выяснить, какое поле было удалено из local configuration file.
Не следует смешивать императивный и декларативный подходы при управлении объектами Kubernetes.
So once you use the applied command going forward, whenever a change is made, the apply command compares all three sections: the local pod definition file, the live object configuration and the last applied configuration stored within the life object configuration file **for deciding what changes are to be made to life object configuration**.

kubectl explain deployment | replicaset | pod | ect
kubectl explain deployment.spec
kubectl explain deployment.spec.strategy

Scheduling