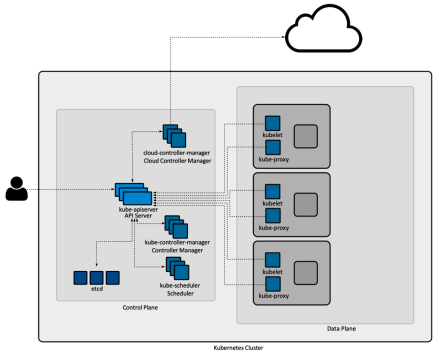
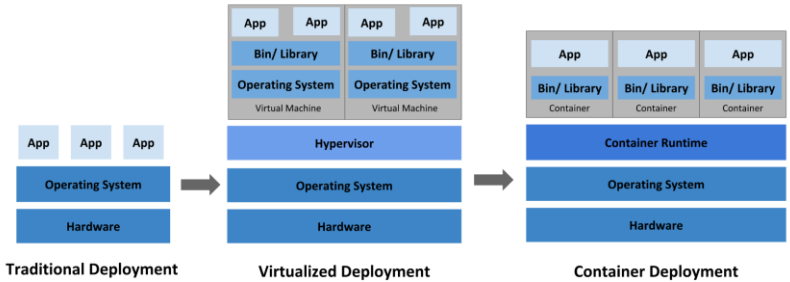
# INTRODUCTION TO CLOUD SYSTEMS

Lecture 3 – Kubernetes p.2

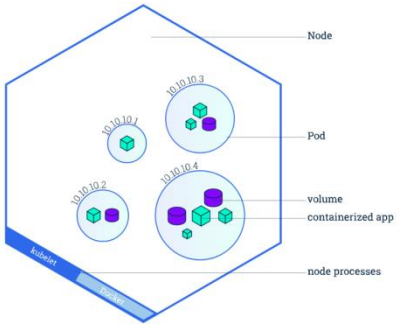
# Previous lecture

kubernetes



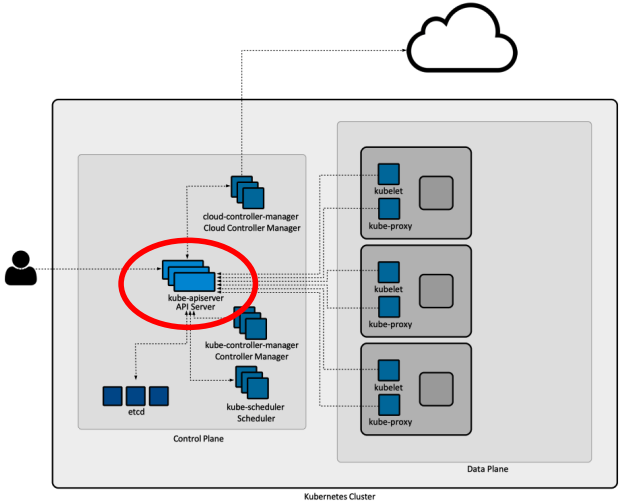

Node overview



# Kubernetes

## Kubernetes API

The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.



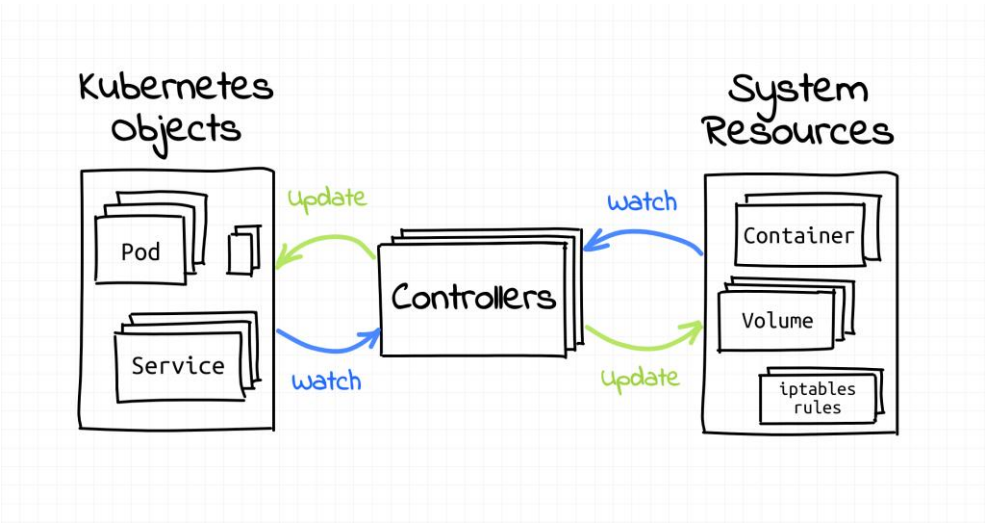

The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events). Most operations can be performed through the kubectl command-line interface or other command-line tools, such as kubeadm, which in turn use the API. However, you can also access the API directly using REST calls.

# Kubernetes

## Kubernetes objects

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

- What containerized applications are running (and on which nodes)
- The resources available to those applications
- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance



# Kubernetes

## Kubernetes objects

A Kubernetes object is a "record of intent"--once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's desired state.

To work with Kubernetes objects--whether to create, modify, or delete them--you'll need to use the Kubernetes API. When you use the kubectl command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. You can also use the Kubernetes API directly in your own programs using one of the Client Libraries.

# Kubernetes

## Kubernetes objects - Object Spec and Status

Almost every Kubernetes object includes two nested object fields that govern the object's configuration: the object spec and the object status. For objects that have a spec, you have to set this when you create the object, providing a description of the characteristics you want the resource to have: its desired state.

The status describes the current state of the object, supplied and updated by the Kubernetes system and its components. The Kubernetes control plane continually and actively manages every object's actual state to match the desired state you supplied.

# Kubernetes

## Kubernetes objects - Object Spec and Status

For example: in Kubernetes, a Deployment is an object that can represent an **application running on your cluster**. When you create the Deployment, you might set the Deployment spec to specify that you want **three replicas** of the application to be running. The Kubernetes system reads the Deployment spec and starts three instances of your desired application--updating the status to match your spec. If any of those instances should fail (a status change), **the Kubernetes system responds to the difference between spec and status by making a correction-**-in this case, starting a replacement instance.

# Kubernetes

## Kubernetes objects - Describing a Kubernetes object

When you create an object in Kubernetes, you must provide the object spec that describes its desired state, as well as some basic information about the object (such as a name). When you use the Kubernetes API to create the object (either directly or via kubectl), that API request must include that information as JSON in the request body. Most often, you provide the information to kubectl in a .yaml file. kubectl converts the information to JSON when making the API request.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods (template)
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

# Kubernetes

## Kubernetes objects - Describing a Kubernetes object

One way to create a Deployment using a .yaml file like the one above is to use the kubectl apply command in the kubectl command-line interface, passing the .yaml file as an argument. Here's an example:

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml
```

# Kubernetes

## Kubernetes objects - Required Fields

In the .yaml file for the Kubernetes object you want to create, you'll need to set values for the following fields:

- **apiVersion** - Which version of the Kubernetes API you're using to create this object
- **kind** - What kind of object you want to create
- **metadata** - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
- **spec** - What state you desire for the object

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods (template)
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

# Kubernetes

## Kubernetes objects - Management

| Management technique | Operates on | Recommended environment | Supported writers | Learning curve |
|---|---|---|---|---|
| Imperative commands | Live objects | Development projects | 1+ | Lowest |
| Imperative object configuration | Individual files | Production projects | 1 | Moderate |
| Declarative object configuration | Directories of files | Production projects | 1+ | Highest |

# Kubernetes

## Kubernetes objects - Management

### Imperative commands

When using imperative commands, a user operates directly on live objects in a cluster. The user provides operations to the kubectl command as arguments or flags.
This is the recommended way to get started or to run a one-off task in a cluster. Because this technique operates directly on live objects, it provides no history of previous configurations.

### Example

Run an instance of the nginx container by creating a Deployment object:

```
kubectl create deployment nginx --image nginx
```

# Kubernetes

## Kubernetes objects - Management

Imperative commands

Advantages compared to object configuration:

- Commands are expressed as a single action word.
- Commands require only a single step to make changes to the cluster.

Disadvantages compared to object configuration:

- Commands do not integrate with change review processes.
- Commands do not provide an audit trail associated with changes.
- Commands do not provide a source of records except for what is live.
- Commands do not provide a template for creating new objects.

# Kubernetes

## Kubernetes objects - Management

### Imperative object configuration

In imperative object configuration, the kubectl command specifies the operation (create, replace, etc.), optional flags and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.

Example

Create the objects defined in a configuration file:

```
kubectl create -f nginx.yaml
```

Delete the objects defined in two configuration files:

```
kubectl delete -f nginx.yaml -f redis.yaml
```

Update the objects defined in a configuration file by overwriting the live configuration:

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

# Kubernetes

## Kubernetes objects - Management

### Imperative object configuration

Advantages compared to imperative commands:

- Object configuration can be stored in a source control system such as Git.
- Object configuration can integrate with processes such as reviewing changes before push and audit trails.
- Object configuration provides a template for creating new objects.

Disadvantages compared to imperative commands:

- Object configuration requires basic understanding of the object schema.
- Object configuration requires the additional step of writing a YAML file.

# Kubernetes

## Kubernetes objects - Management

### Imperative object configuration

Advantages compared to declarative object configuration:

- Imperative object configuration behavior is simpler and easier to understand.
- As of Kubernetes version 1.5, imperative object configuration is more mature.

Disadvantages compared to declarative object configuration:

- Imperative object configuration works best on files, not directories.
- Updates to live objects must be reflected in configuration files, or they will be lost during the next replacement.

# Kubernetes

## Kubernetes objects - Management

Declarative object configuration

When using declarative object configuration, a user operates on object configuration files stored locally, however the user does not define the operations to be taken on the files. Create, update, and delete operations are automatically detected per-object by kubectl. This enables working on directories, where different operations might be needed for different objects.

### Example

Process all object configuration files in the configs directory, and create or patch the live objects. You can first diff to see what changes are going to be made, and then apply:

```
kubectl diff -f configs/

kubectl apply -f configs/
```

Recursively process directories:

```
kubectl diff -R -f configs/

kubectl apply -R -f configs/
```

# Kubernetes

## Kubernetes objects - Management

### Declarative object configuration

Advantages compared to imperative object configuration:

- Changes made directly to live objects are retained, even if they are not merged back into the configuration files.
- Declarative object configuration has better support for operating on directories and automatically detecting operation types (create, patch, delete) per-object.

Disadvantages compared to imperative object configuration:

- Declarative object configuration is harder to debug and understand results when they are unexpected.
- Partial updates using diffs create complex merge and patch operations.

# Kubernetes

## Kubernetes objects - Object Names and IDs

Each object in your cluster has a Name that is unique for that type of resource. Every Kubernetes object also has a UID that is unique across your whole cluster.

For example, you can only have one Pod named myapp-1234 within the same namespace, but you can have one Pod and one Deployment that are each named myapp-1234.

# Kubernetes

## Kubernetes objects - Object Names and IDs

### Names

A client-provided string that refers to an object in a resource URL, such as /api/v1/pods/some-name.

Only one object of a given kind can have a given name at a time. However, if you delete the object, you can make a new object with the same name.

# Kubernetes

## Kubernetes objects - Object Names and IDs

### Names

Four types of commonly used name constraints for resources:

### DNS Subdomain Names

Most resource types require a name that can be used as a DNS subdomain name as defined in RFC 1123. This means the name must:

- contain no more than 253 characters
- contain only lowercase alphanumeric characters, '-' or '.'
- start with an alphanumeric character
- end with an alphanumeric character

# Kubernetes

## Kubernetes objects - Object Names and IDs

### Names

Four types of commonly used name constraints for resources:

### RFC 1123 Label Names

Some resource types require their names to follow the DNS label standard as defined in RFC 1123. This means the name must:

- contain at most 63 characters
- contain only lowercase alphanumeric characters or '-'
- start with an alphanumeric character
- end with an alphanumeric character

# Kubernetes

## Kubernetes objects - Object Names and IDs

### Names

Four types of commonly used name constraints for resources:

### RFC 1035 Label Names

Some resource types require their names to follow the DNS label standard as defined in RFC 1035. This means the name must:

- contain at most 63 characters
- contain only lowercase alphanumeric characters or '-'
- start with an alphabetic character
- end with an alphanumeric character

# Kubernetes

## Kubernetes objects - Object Names and IDs

### Names

Four types of commonly used name constraints for resources:

### Path Segment Names

Some resource types require their names to be able to be safely encoded as a path segment. In other words, the name may not be "." or ".." and the name may not contain "/" or "%".

# Kubernetes

## Kubernetes objects - Object Names and IDs

### Names

Here's an example manifest for a Pod named nginx-demo.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

# Kubernetes

## Kubernetes objects - Object Names and IDs

### UIDs

A Kubernetes systems-generated string to uniquely identify objects.

Every object created over the whole lifetime of a Kubernetes cluster has a distinct UID. It is intended to distinguish between historical occurrences of similar entities.

Kubernetes UIDs are universally unique identifiers (also known as UUIDs). UUIDs are standardized as ISO/IEC 9834-8 and as ITU-T X.667.

# Kubernetes

## Kubernetes objects - Namespaces

In Kubernetes, namespaces provides a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects (e.g. Deployments, Services, etc) and not for cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes, etc).

# Kubernetes

## Kubernetes objects - Namespaces

### When to Use Multiple Namespaces?

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.

Namespaces are a way to divide cluster resources between multiple users (via resource quota).

# Kubernetes

## Kubernetes objects - Namespaces

### Working with Namespaces

You can list the current namespaces in a cluster using:

```
kubectl get namespace

NAME              STATUS   AGE
default           Active   1d
kube-node-lease   Active   1d
kube-public       Active   1d
kube-system       Active   1d
```

# Kubernetes

## Kubernetes objects - Namespaces

Working with Namespaces

Kubernetes starts with four initial namespaces:

```
NAME              STATUS   AGE
default           Active   1d
kube-node-lease   Active   1d
kube-public       Active   1d
kube-system       Active   1d
```

- *default* The default namespace for objects with no other namespace
- *kube-system* The namespace for objects created by the Kubernetes system
- *kube-public* This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.
- *kube-node-lease* This namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane can detect node failure.

# Kubernetes

## Kubernetes objects - Namespaces

### Working with Namespaces

### Setting the namespace for a request

*To set the namespace for a current request, use the --namespace flag.*

```
kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
kubectl get pods --namespace=<insert-namespace-name-here>
```

### Setting the namespace preference

You can permanently save the namespace for all subsequent kubectl commands in that contex

```
kubectl config set-context --current --namespace=<insert-namespace-name-here>
# Validate it
kubectl config view --minify | grep namespace:
```