**What is Kubernetes?**

Get introduced to Kubernetes and understand its architecture.

**We'll cover the following**

* [An overview of Kubernetes](https://www.educative.io/courses/programming-with-kubernetes/what-is-kubernetes#An-overview-of-Kubernetes)
* [Design principles](https://www.educative.io/courses/programming-with-kubernetes/what-is-kubernetes#Design-principles)
  + [Declarative APIs](https://www.educative.io/courses/programming-with-kubernetes/what-is-kubernetes#Declarative-APIs)
  + [Self-healing](https://www.educative.io/courses/programming-with-kubernetes/what-is-kubernetes#Self-healing)
  + [Abstractions and decoupling](https://www.educative.io/courses/programming-with-kubernetes/what-is-kubernetes#Abstractions-and-decoupling)
  + [Immutable infrastructure](https://www.educative.io/courses/programming-with-kubernetes/what-is-kubernetes#Immutable-infrastructure)
* [Kubernetes architecture](https://www.educative.io/courses/programming-with-kubernetes/what-is-kubernetes#Kubernetes-architecture)
* [Creating a cluster](https://www.educative.io/courses/programming-with-kubernetes/what-is-kubernetes#Creating-a-cluster)

**An overview of Kubernetes**

**Kubernetes** is a container orchestration platform that helps better manage containerized applications in distributed environments. It comes with capabilities such as:

* Declarative APIs
* Zero downtime deployment
* Dynamic scheduling with various strategies
* Automated scaling
* Seamless rollout
* Container-centric infrastructure

It dramatically reduces the effort of maintaining applications running in a distributed system, so that a team can concentrate on making the core part, i.e., the application.

* Kubernetes has dramatically changed how applications are built and deployed. It has made essential abstractions that hide the complexity of managing containers, which empowers us to create more sophisticated applications with reliability, scalability, and resiliency. It’s the de facto backbone of current cloud-native application developments because it brings in more velocity and efficiency than before.
* Kubernetes is also portable and provider-agnostic, so it can run on various infrastructures such as bare-metal machines, virtual machines, and public or private cloud platforms. It can even run on our Raspberry Pi.
* Kubernetes adoption has truly gone mainstream. It has been globally used across organizations of all types and sizes. The next frontier for Kubernetes is not adoption anymore, but extension and customization.
* Kubernetes is a well-designed and influential project that allows users to easily run scalable and highly available containerized workloads. It provides lots of highly abstracted objects and APIs that can help users efficiently design and develop containerized services.

**Design principles**

While the architecture, concepts, and components of Kubernetes seem daunting at first glance, it provides unparalleled and robust feature sets. By understanding the building blocks of Kubernetes, we can get to know how to fully leverage its capabilities to run our workloads with scalability and high availability.

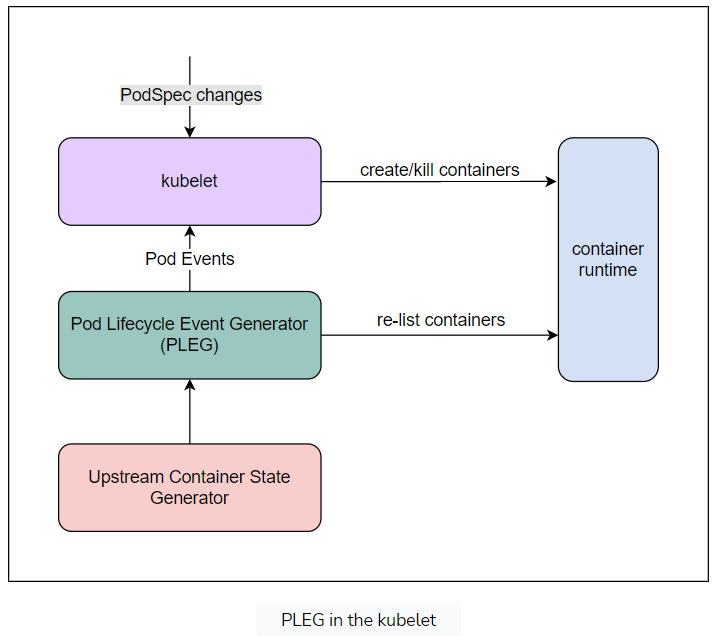
Before diving into Kubernetes architecture, let’s take some time to discuss the design principles of Kubernetes first. These principles will not only help us better understand the architecture of Kubernetes, but also provide us with more hints and guidelines when we try to extend and customize it.

**Declarative APIs**

Perhaps the most important and distinguished design principle in Kubernetes is the **declarative API**, which can simply be thought of as the desired state of our system. Kubernetes ensures every declarative object reflects and matches its desired state. We only need to define what our system should look like in a desired state and leave the rest to Kubernetes. This powers our systems to be self-healing and available, without the need for human intervention.

All the controllers in Kubernetes will reconcile declarative objects, detect the differences between the current and desired states, and proceed with the necessary operations that can drive those objects back to the desired state. This is a rule that must be followed when we implement our controllers, and will also be discussed in depth in later lessons.

Taking the kubelet as an example, the PLEG (Pod Lifecycle Event Generator) controller in the kubelet periodically re-lists all containers to discover changes and generate pod events on mismatched configurations. Then, the kubelet will take the necessary actions to bring those containers back to desired states.



**Self-healing**

Kubernetes is a self-healing system. Though this could be included in the part above about declarative APIs, self-healing is worthy of being regarded as another essential design principle. It’s also a rule that must be followed for any external controllers or operators.

On the one hand, Kubernetes itself is self-healing. With the help of informers, all components inside Kubernetes keep some states or caches that will be periodically refreshed. Those caches also help relieve the burden for kube-apiserver. Even if an event is missed or the network suffers a temporary loss, it will soon come back to life again.

On the other hand, Kubernetes will continuously take actions to ensure those objects are in the desired states, including guarding against any failures and exceptions, automated horizontal scaling, etc. Additionally, this helps improve the stability, availability, and reliability of our system. Let’s suppose we have declared an application with three replicas to Kubernetes; it will continuously make sure there are precisely three healthy replicas running out there, no more and no less. If someone manually destroys a replica, Kubernetes will create one and bring it to the desired state. Likewise, Kubernetes will remove redundant replicas.

**Abstractions and decoupling**

Kubernetes is designed as a loosely coupled distributed system whose components are running separately. It has provided many abstractions as well so that we can use different implementations. For example, if we run Kubernetes on AWS, ELB (Elastic Load Balancing) can be used as an external service load balancer. Because of these abstractions, we can easily customize and extend Kubernetes to what we need. This is how Kubernetes can be so powerful.

At the same time, Kubernetes naturally supports running decoupled services by making abstractions. Kubernetes uses Pod objects to describe a microservice instance. An abstract object Service is used to expose an application running on a set of Pod objects; a Pod is ephemeral, so it would be terminated and replaced at any time. Such a decoupling design helps keep track of healthy Pod objects and bind their IP addresses automatically. As a result, those microservices could be scaled separately because they’re logically independent. Developers can apply new changes into production with a higher velocity.

**Immutable infrastructure**

Last but not least, Kubernetes adheres to the principles of immutable infrastructure.

Everything is prone to failure, and things do go wrong over time. Time is often wasted on chasing down the root causes and debugging the runtime. In most cases, the real causes are misconfigurations.

Kubernetes makes Pod objects the immutable infrastructure. A Pod is the fundamental building block in Kubernetes, consisting of one or more closely related containers. The benefits of using immutable infrastructure are tremendous, such as simplifying operations by treating Pod objects as cattle, reducing configuration drifts, being less error-prone, having a consistent environment, etc. By replacing, instead of maintaining, failed/erroneous instances, it becomes much easier to keep high availability and ensure the service-level agreements. Also, it’s feasible to roll back to a previous state if an error occurs.

**Kubernetes architecture**

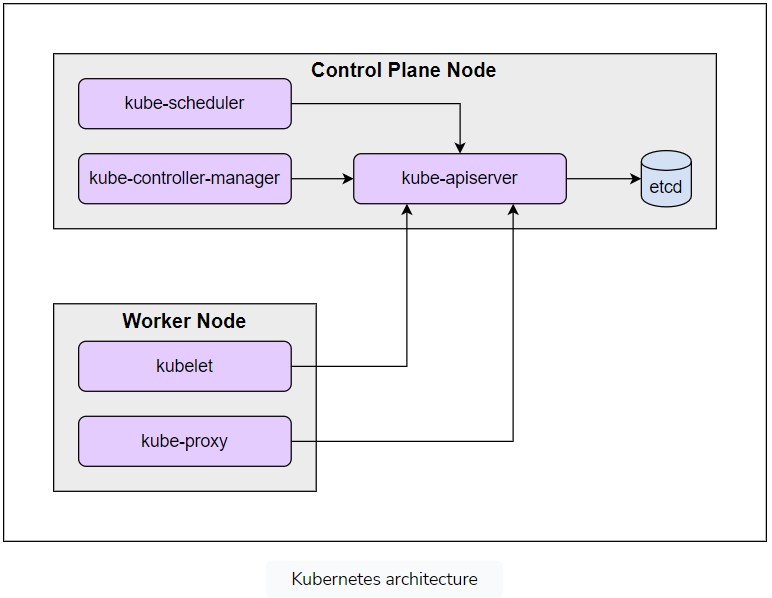
Now, let’s take a closer look at Kubernetes architecture. Kubernetes runs on a bunch of nodes (i.e., bare-metal servers, virtual machines, etc.), which can be categorized into two types:

* The **control plane nodes** (previously known as the “master nodes” host the Kubernetes control plane, acting as a gateway. They manage the overall cluster by exposing REST APIs for users and clients, picking candidate nodes (i.e., scheduling), orchestrating communications between other components, etc. Multiple components, including kube-apiserver, kube-scheduler, kube-controller-manager, and etcd comprise the control plane.
* The **worker nodes** are where the actual applications are running. On worker nodes, kubelet and kube-proxy are deployed.

The etcd cluster stores the entire cluster’s state, including configurations, declarations, and the statuses of all objects. It serves as the brain of a Kubernetes cluster. Thus, a highly available etcd cluster is usually deployed with an odd number of nodes, communicated by the Raft algorithm.

The control plane provides most of the core logic in Kubernetes. It’s strongly recommended to have multiple master nodes for high availability. If any individual control plane node goes down, or if one of the control plane components faces any temporary failures, the whole cluster will work unaffected.

All the components in Kubernetes are loosely coupled. We’ll later elaborate on these five components in the following lessons.



There’s no intrinsic difference between control plane nodes and worker nodes. We could run the kubelet on control plane nodes. However, this isn’t advisable, because it might bring in security risks and affect control plane stability. Most installers, like kubeadm, will set taints to control plane nodes by default. If we do want to schedule Pod objects on these nodes, we could run the commands given below to remove such taints:

1

kubectl taint nodes --all node-role.kubernetes.io/control-plane- node-role.kubernetes.io/master-

The output will be as follows:

1

2

3

node "node-01" untainted

node "node-02" untainted

...

The command above will remove the taints node-role.kubernetes.io/control-plane and node-role.kubernetes.io/master from any nodes, including the control plane nodes. Legacy taint name node-role.kubernetes.io/master is officially removed starting from version 1.25 (click [this tracking issue](https://github.com/kubernetes/enhancements/issues/2067) to learn more).

**Creating a cluster**

There are quite a few methods or tools to create a Kubernetes cluster. In this lesson, we’ll cover a few of them.

The Kind method is not suggested, because it’s hard to configure and customize.

The kubeadm method is the officially recommended way to bootstrap a minimum viable Kubernetes cluster that conforms to best practices. In this course, we strongly suggest using kubeadm to create a Kubernetes cluster due to the following reasons:

* We don’t need a very large cluster. A small cluster with two nodes is enough for this course.
* It’s super easy and straightforward. It only takes two steps, kubeadm init and kubeadm join, to create a cluster.
* It helps eliminate complex configurations and lets us focus on the main parts. In most of the cases, default configurations are enough for our use cases. It is easier to demonstrate what we’ve changed and customize the clusters using simple configurations.
* It’s convenient to purge the cluster with kubeadm reset and recreate a new one when our Kubernetes cluster gets into trouble.
* It’s easy to get a Kubernetes cluster of any version. Every release of kubeadm has a range of supported Kubernetes versions. We can bootstrap a Kubernetes cluster with our desired version by choosing a matched kubeadm. This can be very useful for performing fast validations to verify our customization.

**The Heart of Kubernetes: The kube-apiserver**

Learn about the kube-apiserver.

**We'll cover the following**

* [The kube-apiserver](https://www.educative.io/courses/programming-with-kubernetes/the-heart-of-kubernetes-the-kube-apiserver#The-kube-apiserver)
* [Inside the kube-apiserver](https://www.educative.io/courses/programming-with-kubernetes/the-heart-of-kubernetes-the-kube-apiserver#Inside-the-kube-apiserver)
  + [Stage 1: Authentication](https://www.educative.io/courses/programming-with-kubernetes/the-heart-of-kubernetes-the-kube-apiserver#Stage-1-Authentication)
  + [Stage 2: Authorization](https://www.educative.io/courses/programming-with-kubernetes/the-heart-of-kubernetes-the-kube-apiserver#Stage-2-Authorization)
  + [Stage 3: Admission control](https://www.educative.io/courses/programming-with-kubernetes/the-heart-of-kubernetes-the-kube-apiserver#Stage-3-Admission-control)
  + [Stage 4: Validating and storing it persistently](https://www.educative.io/courses/programming-with-kubernetes/the-heart-of-kubernetes-the-kube-apiserver#Stage-4-Validating-and-storing-it-persistently)
* [How to interact with Kubernetes](https://www.educative.io/courses/programming-with-kubernetes/the-heart-of-kubernetes-the-kube-apiserver#How-to-interact-with-Kubernetes)
* [The client-go library](https://www.educative.io/courses/programming-with-kubernetes/the-heart-of-kubernetes-the-kube-apiserver#The-client-go-library)

**The kube-apiserver**

The kube-apiserver is the most essential and complicated component in Kubernetes.

It sits in the center of Kubernetes as the primary management point, receiving and handling all RESTful requests, such as creating/updating/deleting a Pod, querying a group of resources, etc. It accepts various kinds of information from requests and disseminates them to all clients. The kubectl client is one of the most commonly used clients for interacting with the kube-apiserver. We can think of the kube-apiserver as the front-end server for the control plane. This is why we regard the kube-apiserver as the “heart” of Kubernetes.

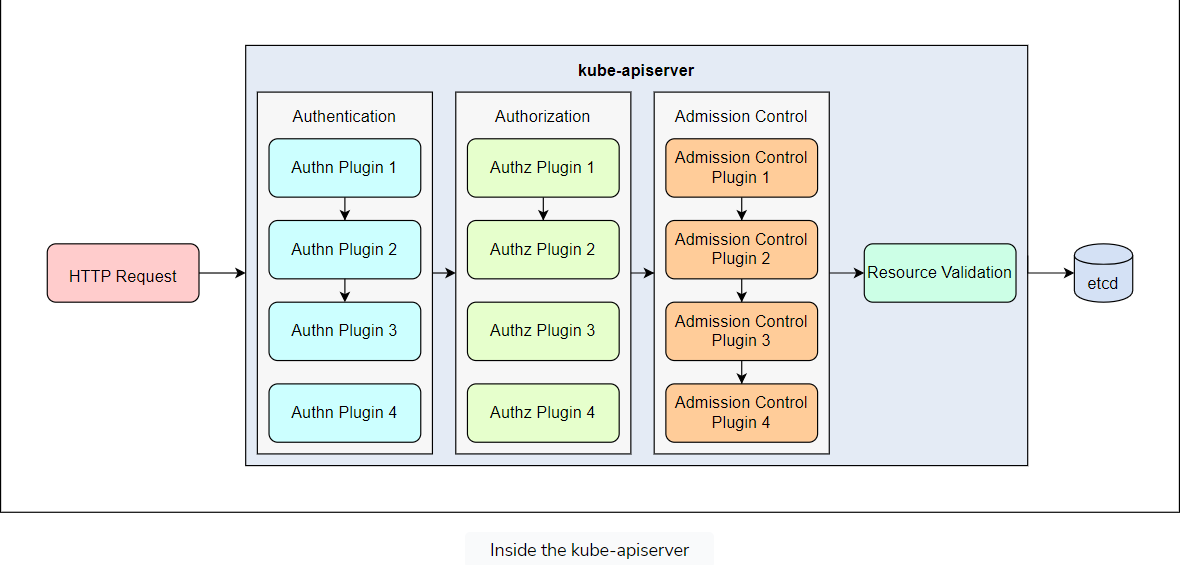
It’s the only component allowed to communicate with the back-end etcd storage. This is an excellent way to decouple applications with data, which makes the kube-apiserver stateless and horizontally scalable. The kube-apiserver is so essential that it should be deployed with multiple replicas to provide high availability service for the entire cluster. Additionally, security must be taken into account, which is crucial for such a distributed system.

The kube-apiserver provides authentications and authorizations for all requests, including those from other Kubernetes components, such as kube-scheduler, kube-controller-manager, etc.

The kube-apiserver will also create and maintain RESTful APIs for every CRD (Custom Resource Definition).

**Inside the kube-apiserver**

Now, let’s take a closer look at the kube-apiserver. The graph below shows what happens inside the kube-apiserver when receiving a request.



### Stage 1: Authentication

Security is vital. First, the kube-apiserver needs to **authenticate** the client sending the request, because the safety of the kube-apiserver concerns the entire cluster. The kube-apiserver does allow requests from anonymous clients, but it isn’t suggested in any production environments. In Kubernetes, we have various objects, like Deployment, Pod, Service, etc., but there is no such object called User. So, we can’t create a User object via an API call. Kubernetes User objects are created in the following ways:

* **mTLS (mutual TLS)** for authentication. Any request that presents a valid certificate signed by the CA (certificate authority) of our cluster is considered to be valid. (In an mTLS, both clients and servers are using valid certificates signed by a common CA.) The username derives from the common name (CN) field in the subject of the client certificate (e.g., /CN=Shelton").
* Bootstrap **and** ServiceAccount: These are tokens managed by Kubernetes, which can be used for authentication as well. These two kinds of built-in tokens in Kubernetes are used widely, especially the ServiceAccount token. It can be associated with a Pod running in the Kubernetes cluster. The ServiceAccount token can be mounted into a Pod at the fixed location /run/secrets/kubernetes.io/serviceaccount/token, so that an in-cluster process can read and use it to talk to the kube-apiserver. The ServiceAccount token can be explicitly associated with a Pod by the field podSpec.serviceAccountName. The long tokens are recognized and interpreted to the User in another form. Such a long ServiceAccount token will be construed with the user name system:serviceaccount:(NAMESPACE):(SERVICEACCOUNT).
* **Static token**: We may also provide a static file that contains a group of usernames and tokens/passwords by the flag --token-auth-file=SOMEFILE. However, this is not advisable in any production environment. This is a static configuration that the kube-apiserver will not reload unless it restarts.
* **OpenID Connect tokens:** Kubernetes also supports using external OAuth2 providers.
* **Webhook tokens:** To better integrate with other authentication systems, Kubernetes provides webhooks for customization. It’s easy to write our implementations to verify the identity. We’ll dive into this in-depth and show a completed tutorial on this in a later lesson.

The kube-apiserver builds up a union authentication chain that connects these authentication plugins. The plugins will be called one by one until one of them confirms the identity that sends the request. If none can identify the user, an unauthorized response with a status code of 401 will be returned. The user name may come from a ServiceAccount, user ID, CN name, or group name that belongs to the user. This user name will be used in the next stage—authorization.

### Stage 2: Authorization

In **authorization**, the main task is to determine user privileges. For example, Alice can delete a Pod in Kubernetes, while Bob is only allowed to create a Pod.

During the authorization stage, the kube-apiserver can also be configured to use one or more authorization plugins, like in the authentication stage. Identical to that, the request gets authorized if one of the plugins admits it.

**Note:** The flag --authorization-mode strings specifies an ordered and comma-delimited list of AlwaysAllow,AlwaysDeny,ABAC,Webhook,RBAC,Node. The default value is AlwaysAllow. Typically, we explicitly set it to --authorization-mode=Node,RBAC.

* ABAC: This stands for “attribute-based access control” and is the same as the static token above. It’s configured with a static file that combines users and policies.
* RBAC: This stands for “role-based access control” and is the most commonly used method for authorization in Kubernetes. Other external authorization systems can also be integrated into Kubernetes as authorization plugins.

As soon as the request passes authorization, it comes to the following stage.

### Stage 3: Admission control

Only requests trying to create, modify, or delete a resource will be sent to this stage. In this stage, the kube-apiserver also builds a chain for a group of admission control plugins. These plugins can intercept requests or inject some default data, such as initializing missing fields in the resource specifications, overriding some values, or even rejecting a request.

The kube-apiserver does provide a list of enabled plugins.

**Note:** The flag --enable-admission-plugins strings specifies a group of comma-delimited admission plugins that should be enabled, where the order doesn’t matter.

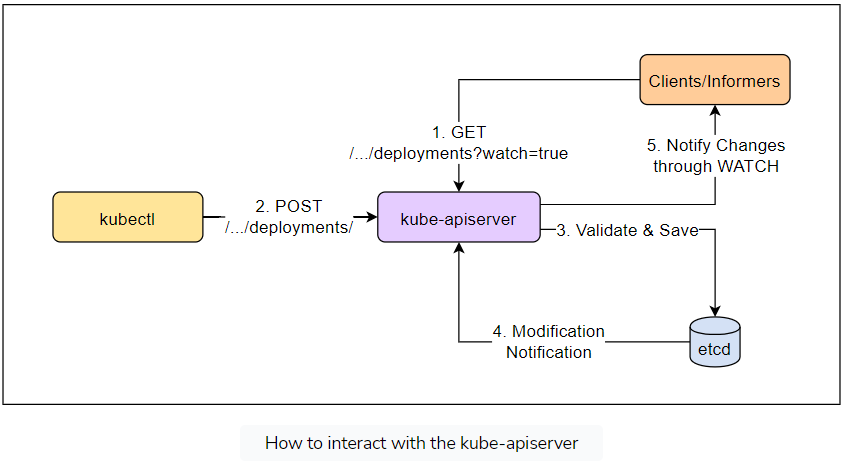
Different from the two stages above, the request must pass through all the configured admission control plugins.

### Stage 4: Validating and storing it persistently

After the request passes through the above stage, the kube-apiserver can validate the object, such as port range, string length, and qualified domain name, and then store it in etcd and return a response.

## How to interact with Kubernetes

The kube-apiserver is the heart of Kubernetes that all the other components talk to and fetch needed data from to process. The kube-apiserver provides a WATCH mechanism, which is the most charming feature design. The kube-apiserver doesn’t tell all the other components/controllers what to do. All it does is enable those components/controllers to observe changes in their interested objects. As mentioned in the previous lesson, all the components are running in their own ways, which is one of the golden rules for designing and writing a good controller/operator.



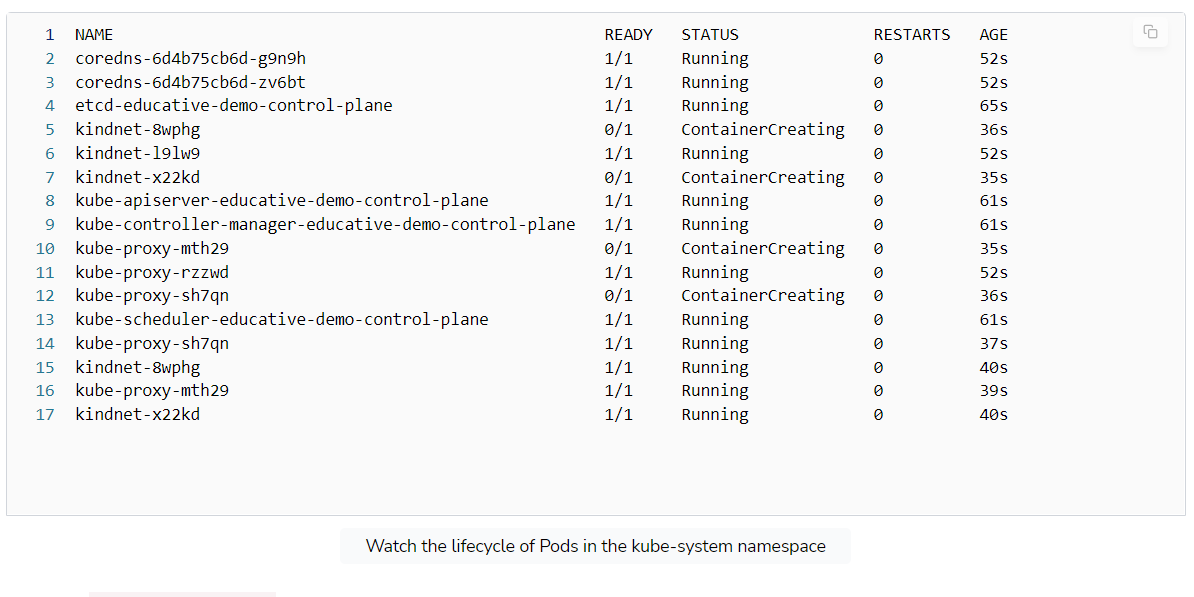
Clients watch for changes by opening an HTTP connection (with the query parameter watch=true) to the kube-apiserver. With this connection, a stream of notifications on the watched objects will be received by all connected clients that are watching these objects. Every time an object is created, updated, and deleted, the kube-apiserver will send out a new version of the objects to all connected clients.

It’s the WATCH mechanism in the kube-apiserver that brings these loosely coupled components to work in perfect harmony. Using WATCH, it’s more efficient to receive the changes, instead of polling/re-listing periodically. Additionally, it greatly helps reduce the body sizes of the request payloads and mitigate the burden of itself, especially when the cluster grows bigger with more objects served. This is illustrative of how to design and write a good controller/operator. This golden rule uses WATCH to get notified of changes instead of re-listing.

It’s simple to use kubectl to watch resources. For example, when deploying a Pod, we don’t need to get or list the Pod objects repeatedly with kubectl get pod. Instead, we can append the flag -w (or --watch) at the end of our command and get notified of each change (creating, updating, and deleting) made to a Pod.

kubectl get pod -n kube-system -w

The output will be similar to the following:



## The client-go library

Kubernetes also provides a [native programmable library](https://github.com/kubernetes/client-go) client-go that can be used to watch resources. We’ll demonstrate this in later lessons.