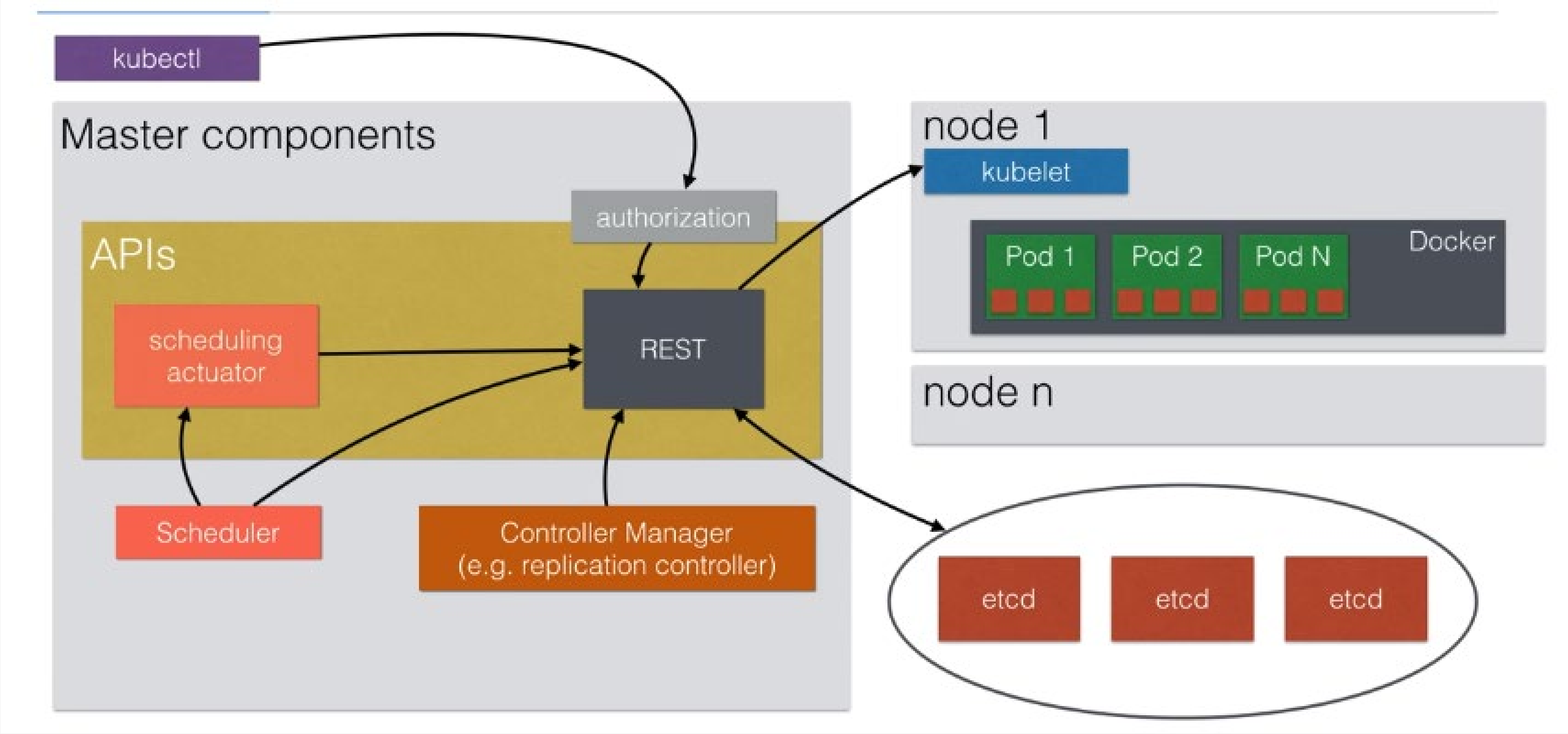
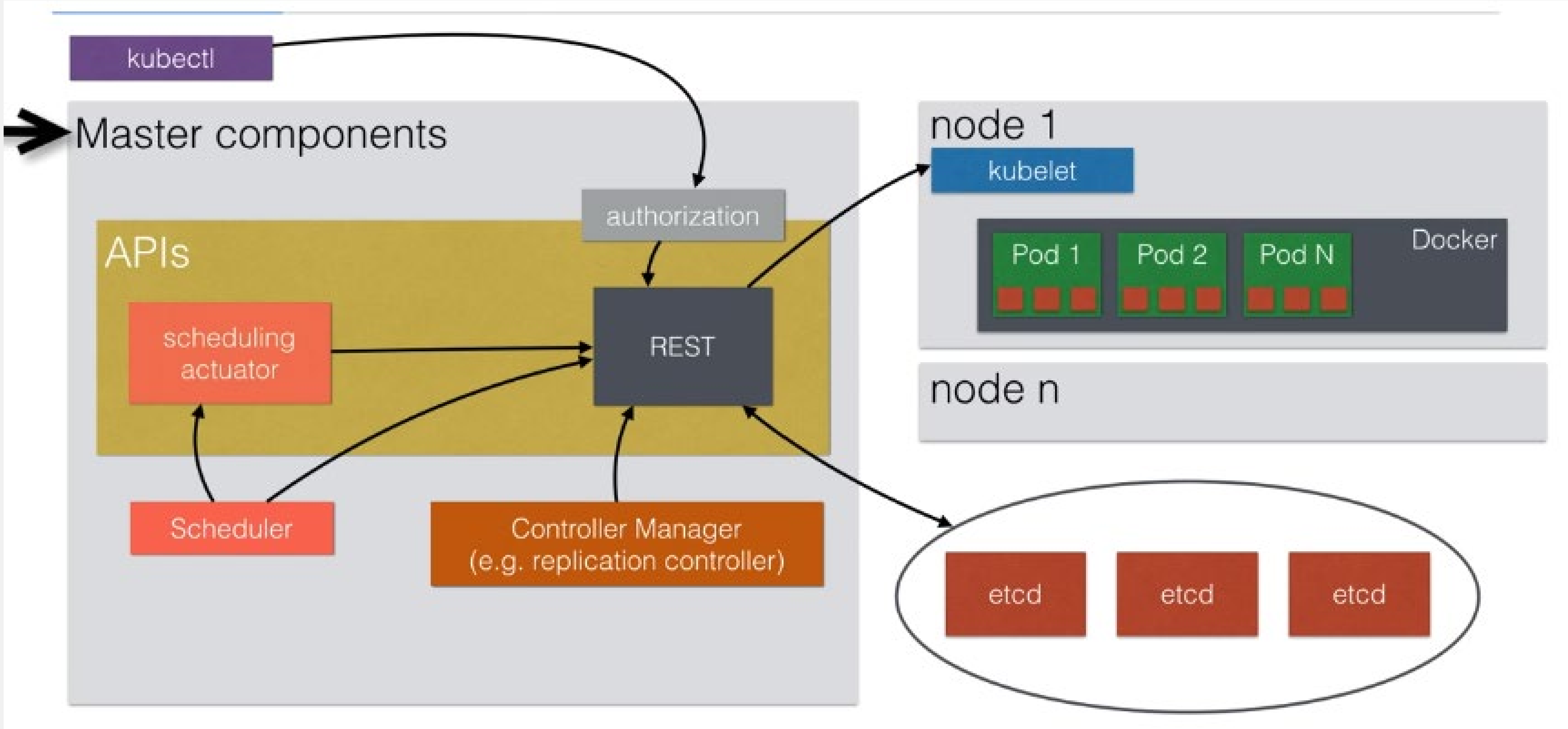
Kubernetes Master Services

Master Server

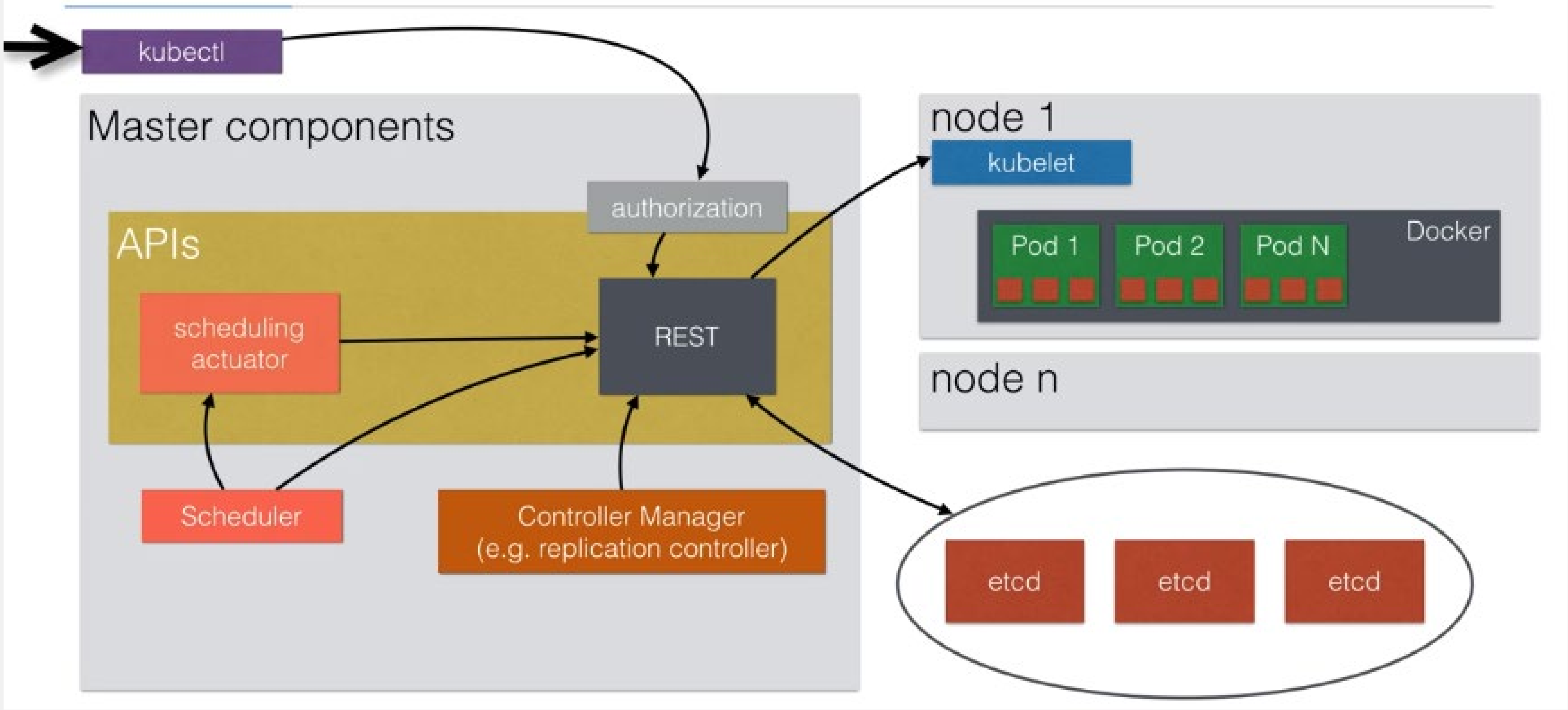
Architecture Overview



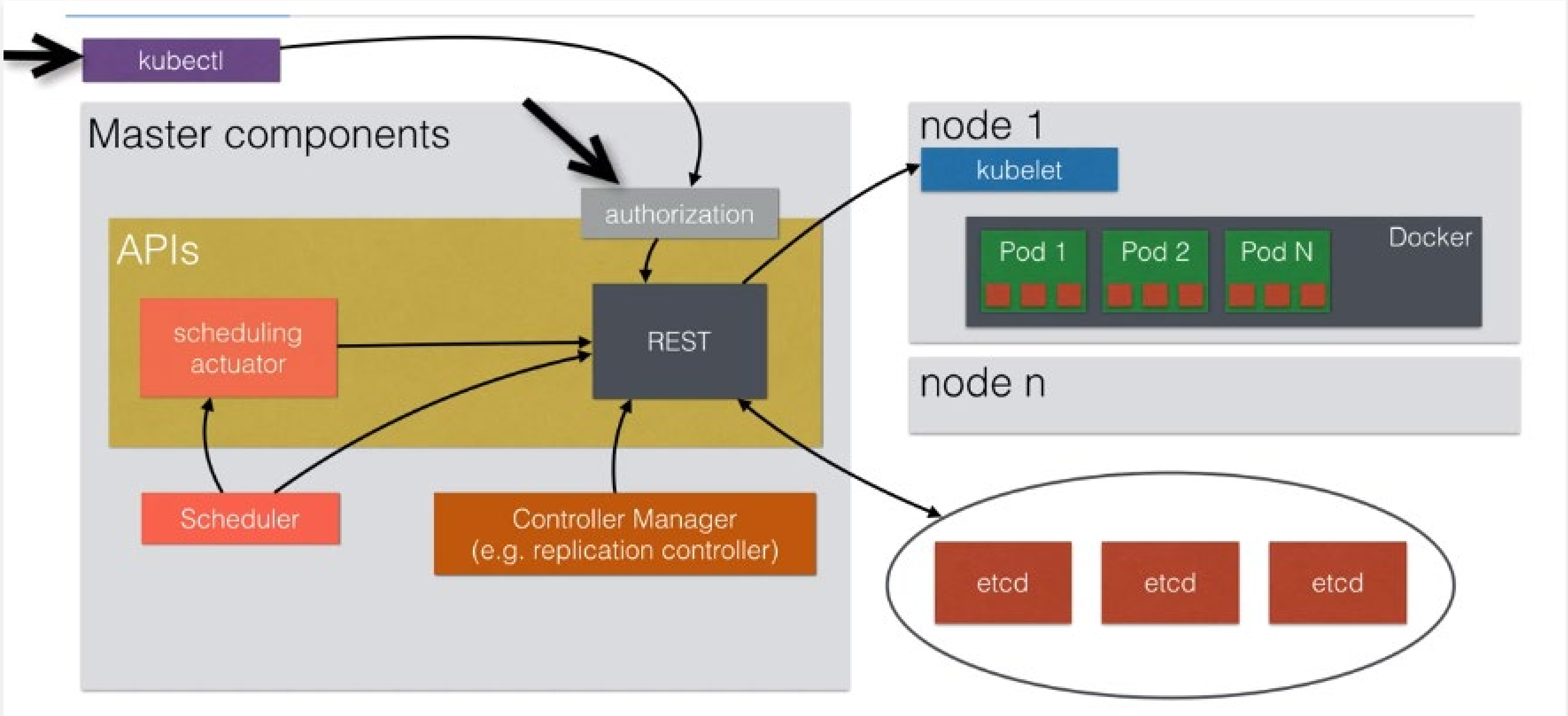
Architecture Overview



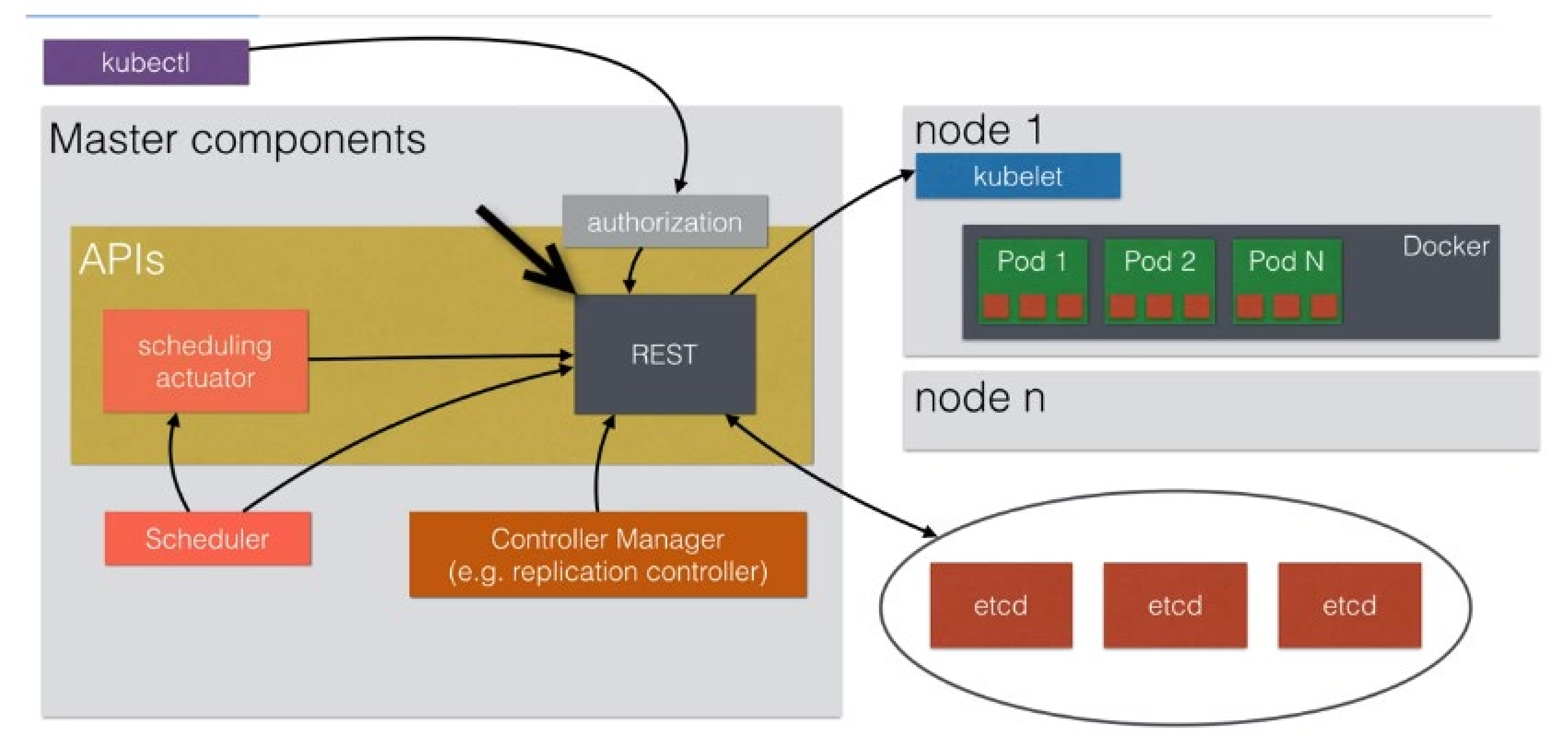
Architecture Overview



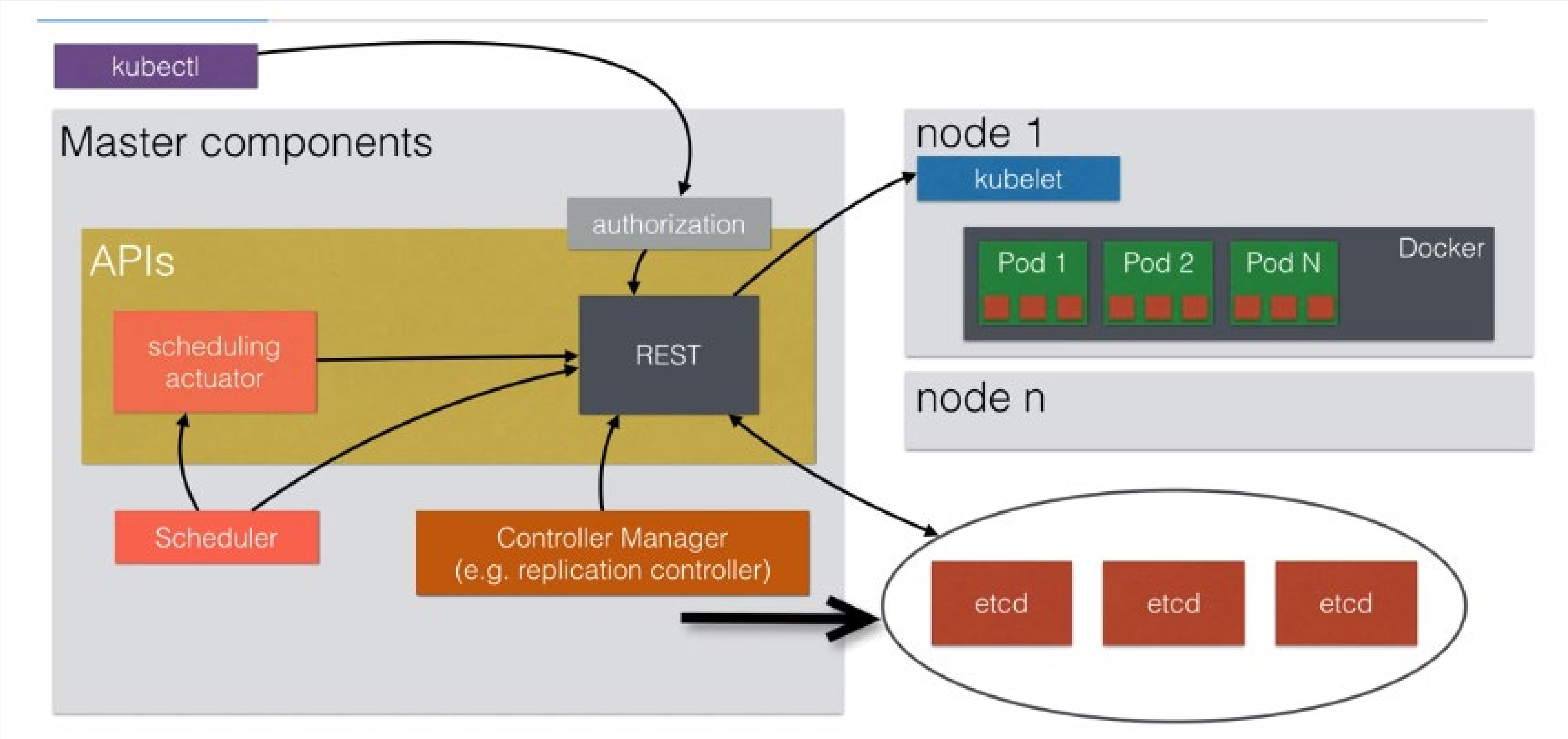
Architecture Overview



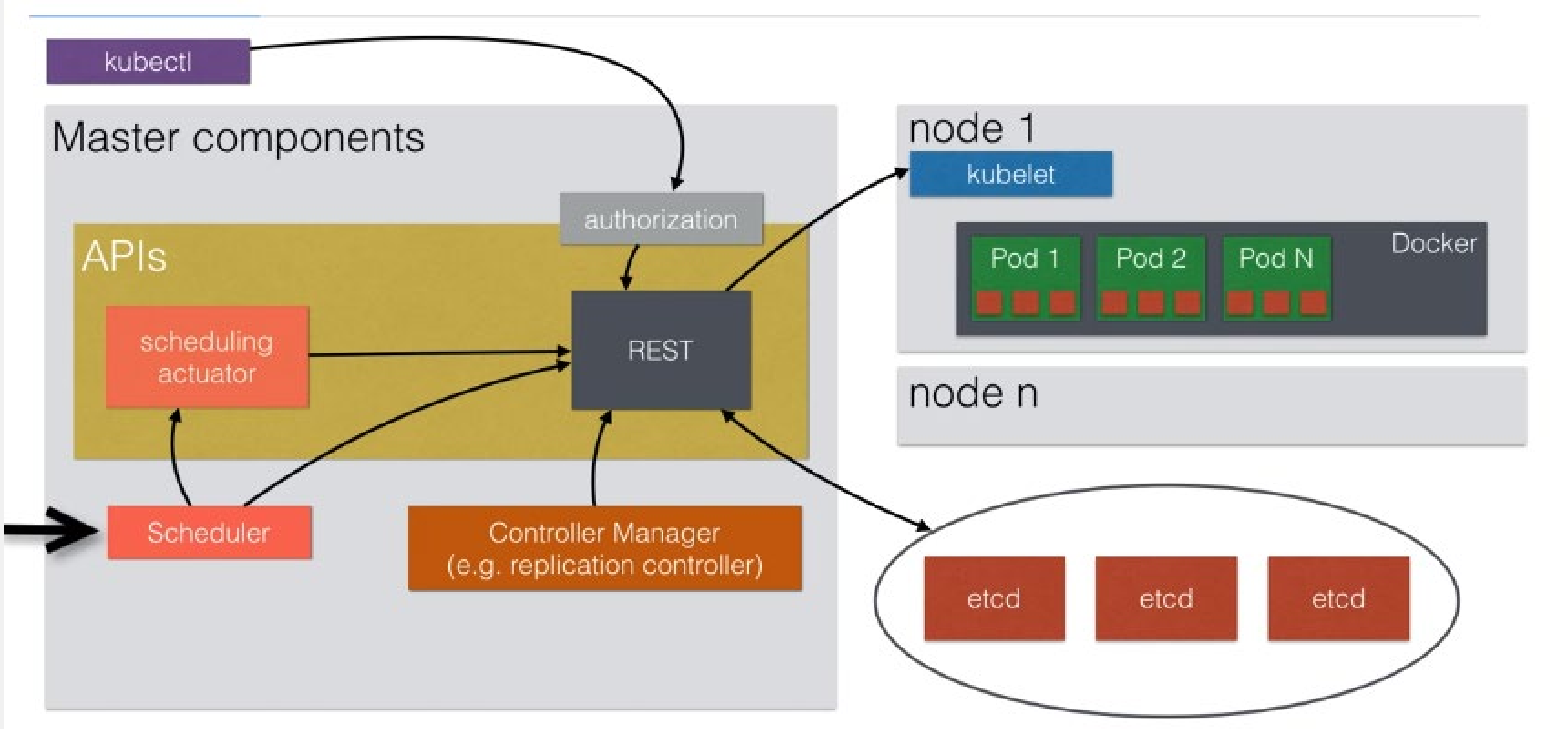
Architecture Overview



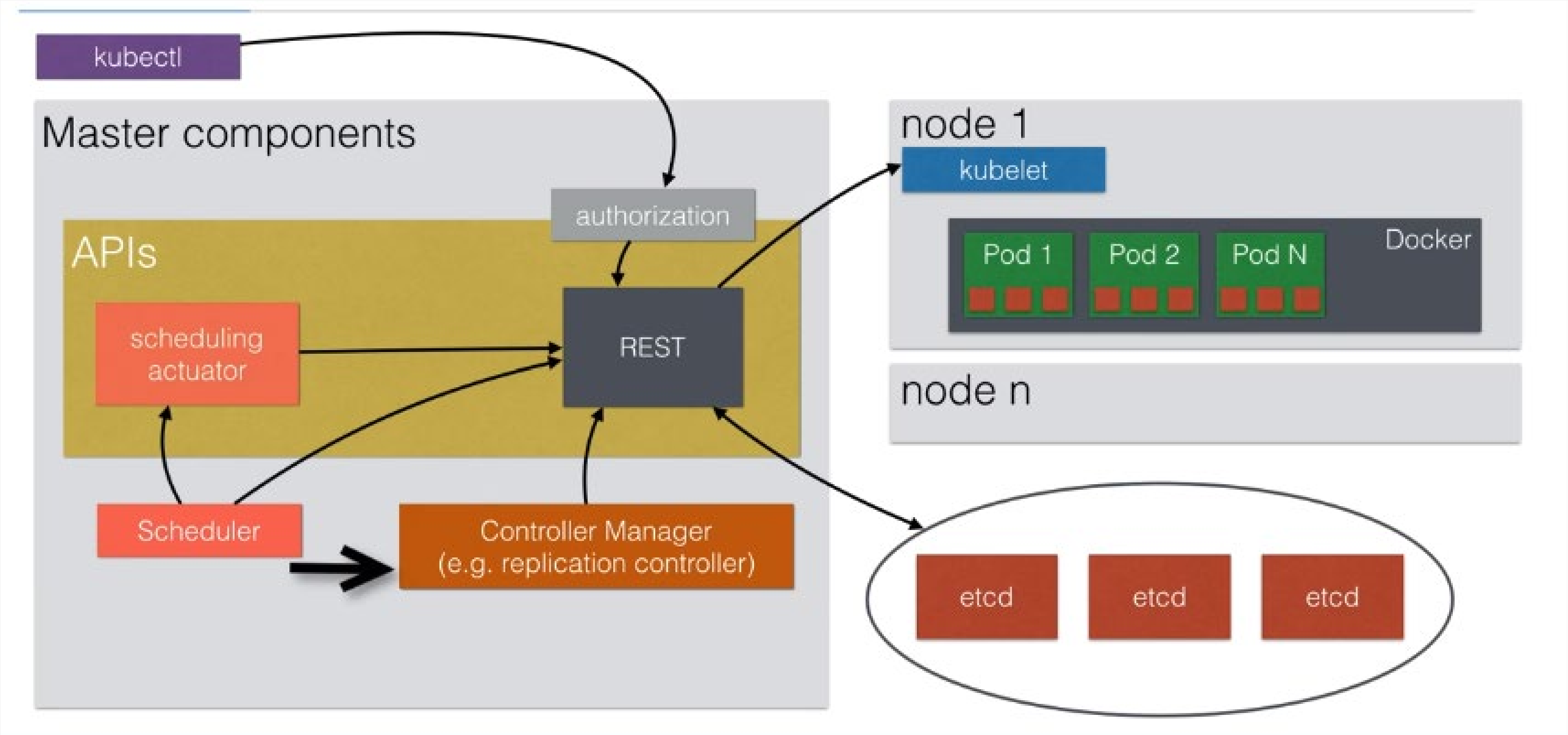
Architecture Overview



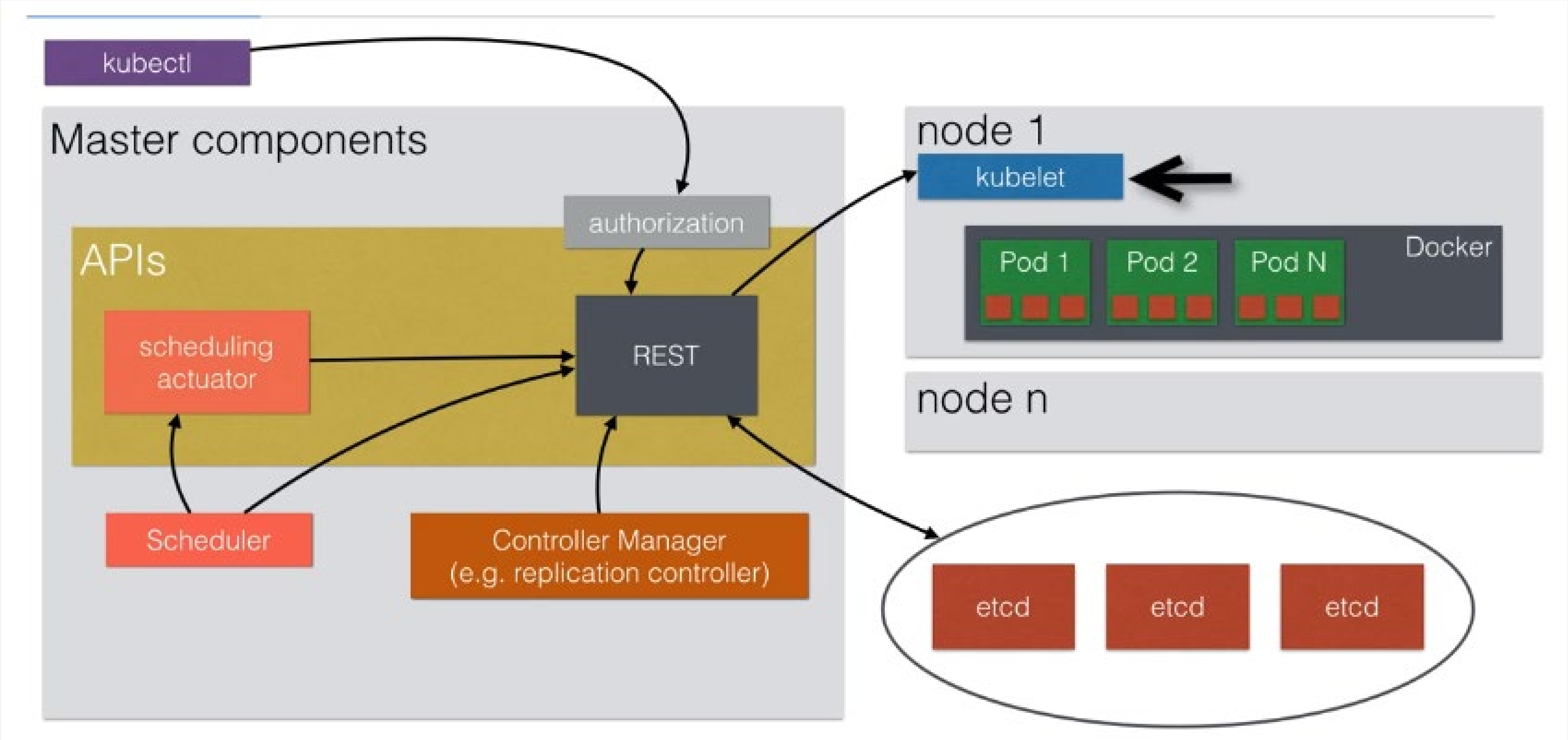
Architecture Overview



Architecture Overview



Architecture Overview



Resource Quotas

Resource Quotas

- When a Kubernetes cluster is used by multiple **people** or **teams**, **resource management** becomes more important
  - You want to be able to **manage the resources** you give to a person or a team
  - You don't want one person or team **taking up all the resources** (e.g. CPU/Memory) of the cluster
- You can divide your cluster in **namespaces** (explained in next lecture) and enable resource quotas on it
  - You can do this using the **ResourceQuota** and **ObjectQuota** objects

Resource Quotas

- Each container can specify **request capacity** and **capacity limits**
    - **Request capacity** is an **explicit request** for resources
        - The scheduler can use the **request capacity** to make decisions on where to put the pod on
        - You can see it as a **minimum amount of resources the pod needs**
    - **Resource limit** is a limit imposed to the container
        - The container will not be able to utilize more resources than specified

Resource Quotas

- Example of resource quotas:
  - You run a **deployment** with a **pod** with a **CPU resource** request of **200m**
  - 200m = 200 millicpu (or also 200 millicores)
  - 200m = 0.2, which is 20% of a CPU core of the running node
    - If the node has 2 cores, it's still 20% of a single core
  - You can also put a limit, e.g. on 400m
  - Memory quotas are defined by MiB or GiB

Resource Quotas

- If a capacity quota (e.g. mem / cpu) has been specified by the administrator, then each pod needs to specify capacity quota during creation
  - The administrator can specify default request values for pods that don’t specify any values for capacity
  - The same is valid for limit quotas
- If a resource is requested more than the allowed capacity, the server API will give an error 403 FORBIDDEN - and kubectl will show an error

Resource Quotas

- The administrator can set the following resource limits within a namespace:

Resource	Description
requests.cpu	The sum of **CPU requests** of all pods cannot exceed this value
requests.mem	The sum of **MEM requests** of all pods cannot exceed this value
requests.storage	The sum of **storage requests** of all persistent volume claims cannot exceed this value
limits.cpu	The sum of **CPU limits** of all pods cannot exceed this value
limits.memory	The sum of **MEM limits** of all pods cannot exceed this value

Resource Quotas

- The administrator can set the following object limits:

Resource	Description
configmaps	total number of **configmaps** that can exist in a namespace
persistentvolumeclaims	total number of **persistent volume claims** that can exist in a namespace
pods	total number of **pods** that can exist in a namespace
replicationcontrollers	total number of **replicationcontrollers** that can exist in a namespace
resourcequotas	total number of **resource quotas** that can exist in a namespace
services	total number of **services** that can exist in a namespace
services.loadbalancer	total number of **load balancers** that can exist in a namespace
services.nodeports	total number of **nodeports** that can exist in a namespace
secrets	total number of secrets that can exist in a namespace

Namespaces

Namespaces

- Namespaces allow you to create **virtual clusters** within the same physical cluster
- Namespaces **logically separates** your cluster
- The standard namespace is called “**default**” and that’s where all resources are launched in by default
  - There is also namespace for kubernetes specific resources, called **kube-system**
- Namespaces are intended when you have **multiple teams / projects** using the Kubernetes cluster

Namespaces

- The name of resources need to be unique within a namespace, but not across namespaces
    - e.g. you can have the deployment “helloworld” multiple times in different namespaces, but not twice in one namespace
- You can divide resources of a Kubernetes cluster using namespaces
    - You can limit resources on a per namespace basis
    - e.g. the marketing team can only use a maximum of 10 GiB of memory, 2 loadbalancers, 2 CPU cores

Namespaces

- First you need to create a new namespace

```
$ kubectl create namespace myspace
```

- You can list namespaces:

```
$ kubectl get namespaces
NAME         LABELS    STATUS
default        <none>    Active
kube-system  <none>    Active
myspace        <none>    Active
```

- You can set a default namespace to launch resources in:

```
$ export CONTEXT=$(kubectl config view | awk '/current-context/ {print $2}')
$ kubectl config set-context $CONTEXT —namespace=myspace
```

Namespaces

- You can then create resource limits within that namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
  namespace: myspace
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Namespaces

- You can also create object limits:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
  namespace: myspace
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
```

Demo Placeholder

Namespace quotas

User Management

User Management

- There are **2 types** of users you can create
    - A **normal user**, which is used to access the user externally
        - e.g. through kubectl
        - This user is **not managed using objects**
    - A **Service user**, which is **managed by an object in Kubernetes**
        - This type of user is used to **authenticate within** the cluster
        - e.g. from inside a pod, or from a kubelet
        - These credentials are managed like **Secrets**

User Management

- There are multiple **authentication strategies** for normal users:
  - Client Certificates
  - Bearer Tokens
  - Authentication Proxy
  - HTTP Basic Authentication
  - OpenID
  - Webhooks

User Management

- Service Users are using **Service Account Tokens**
- They are stored as **credentials using Secrets**
  - Those Secrets are also mounted in pods to allow communication between the services
- Service Users are **specific to a namespace**
- They are created automatically by the API or manually using **objects**
- Any API call **not authenticated** is considered as an **anonymous** user

User Management

- Independently from the authentication mechanism, normal users have the following **attributes**:
  - a Username (e.g. user123 or user@email.com)
  - a UID
  - Groups
  - Extra fields to store extra information

User Management

- After a normal users authenticates, it will have access to everything
- To **limit** access, you need to configure **authorization**
- There are again multiple offerings to choose from:
  - AlwaysAllow / AlwaysDeny
  - ABAC (Attribute-Based Access Control)
  - RBAC (Role Based Access Control)
  - Webhook (authorization by remote service)

RBAC

Authorization

- There are multiple **authorization** module available:
  - **Node**: a special purpose authorization mode that authorizes API requests made by **kubelets**
  - **ABAC**: attribute-based access control
    - Access rights are controlled by policies that combine attributes
    - e.g. user “alice" can do anything in namespace “marketing”
    - ABAC does not allow very granular permission control

Authorization

- **RBAC**: role based access control
  - Regulates access using **roles**
  - Allows admins to dynamically configure permission policies
  - This is what I'll use in the demo
- **Webhook**: sends authorization request to an external REST interface
  - Interesting option if you want to write your own **authorization server**
  - You can parse the incoming **payload** (which is JSON) and reply with access granted or access denied

RBAC

- To enable an **authorization mode**, you need to pass --authorization-mode= to the API server at startup
  - For example, to enable RBAC, you pass —authorization-mode=RBAC
- Most tools now provision a cluster with **RBAC enabled by default** (like kops and kubeadm)
  - For minikube, it'll become default at some point (see https://github.com/kubernetes/minikube/issues/1722)

RBAC

- You can **add RBAC resources** with *kubectl* to grant permissions
    - You first describe them in **yaml** format, then apply them to the cluster
- First you define **a role**, then you can **assign users/groups** to that role
- You can create roles **limited to a namespace** or you can create roles where the **access applies to all namespaces**
    - **Role** (single namespace) and **ClusterRole** (cluster-wide)
    - **RoleBinding** (single namespace) and **ClusterRoleBinding** (cluster-wide)

RBAC Role

- RBAC Role granting read access to pods and secrets within default namespace

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods", "secrets"]
  verbs: ["get", "watch", "list"]
```

RBAC Role

- Next step is to assign users to the newly created role

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: bob
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

RBAC Role

- If you rather want to create a role that spans all namespaces, you can use ClusterRole

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: pod-reader-clusterwide
rules:
- apiGroups: [""]
  resources: ["pods", "secrets"]
  verbs: ["get", "watch", "list"]
```

RBAC Role

- If you need to assign a user to a cluster-wide role, you need to use ClusterRoleBinding

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader-clusterwide
  apiGroup: rbac.authorization.k8s.io
```

Demo Placeholder

Namespace quotas

Node Maintenance

Node Maintenance

- It is the **Node Controller** that is responsible for managing the Node objects
  - It assigns **IP space** to the node when a new node is launched
  - It keeps the **node list** up to date with the available machines
  - The node controller is also monitoring the **health of the node**
    - If a node is **unhealthy it gets deleted**
    - Pods running on the unhealthy node will then get **rescheduled**

Node Maintenance

- When adding a new node, the **kubelet** will attempt to register itself
- This is called **self-registration** and is the default behavior
- It allows you to **easily add more nodes** to the cluster without making API changes yourself
- A new node object is **automatically** created with:
    - The metadata (with a name: IP or hostname)
    - Labels (e.g. cloud region / availability zone / instance size)

Node Maintenance

- When you want to **decommission** a node, you want to do it gracefully
  - You drain a node before you shut it down or take it out of the cluster
- To drain a node, you can use the following command:

```
$ kubectl drain nodename --grace-period=600
```

- If the node runs pods not managed by a controller, but is just a single pod:

```
$ kubectl drain nodename --force
```

Demo Placeholder

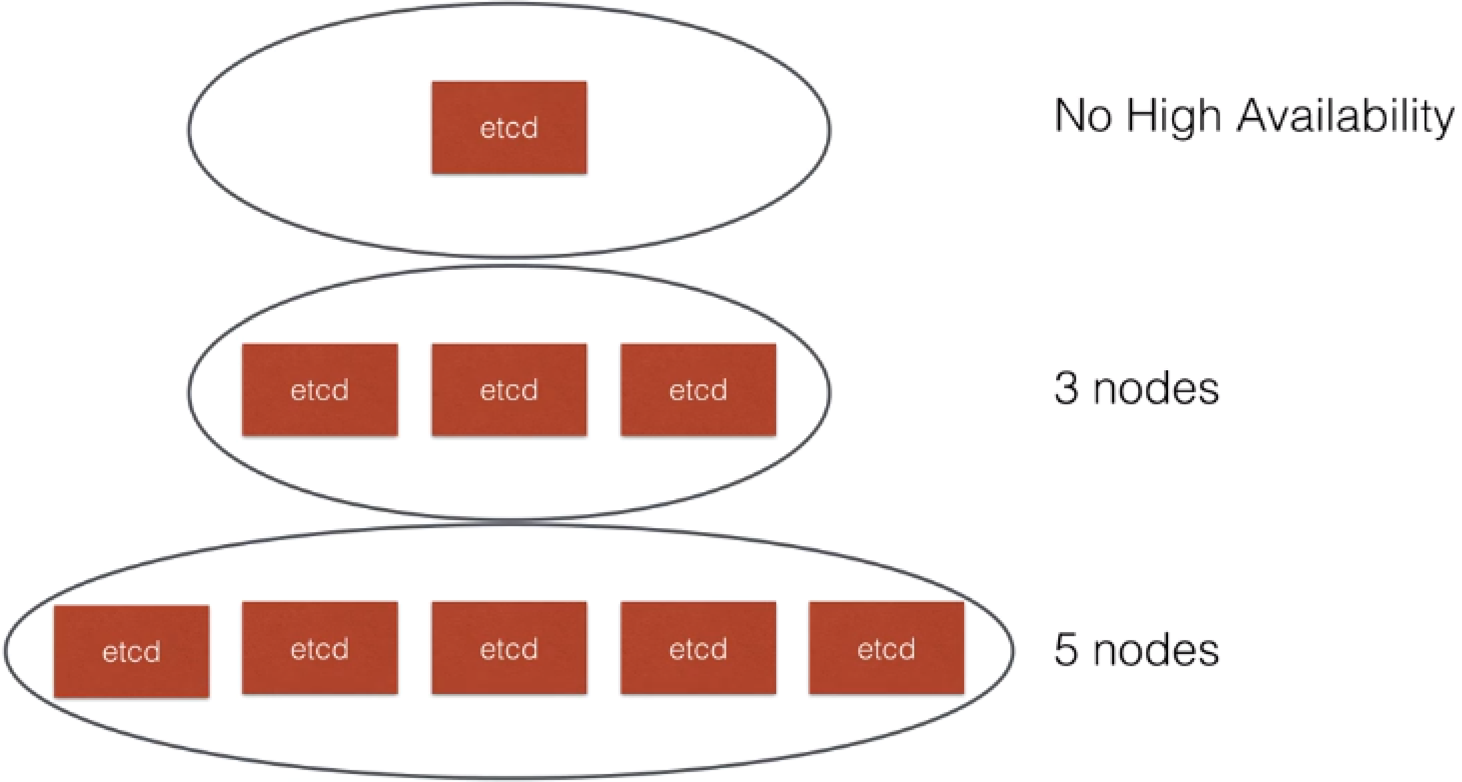
Drain the node

High Availability

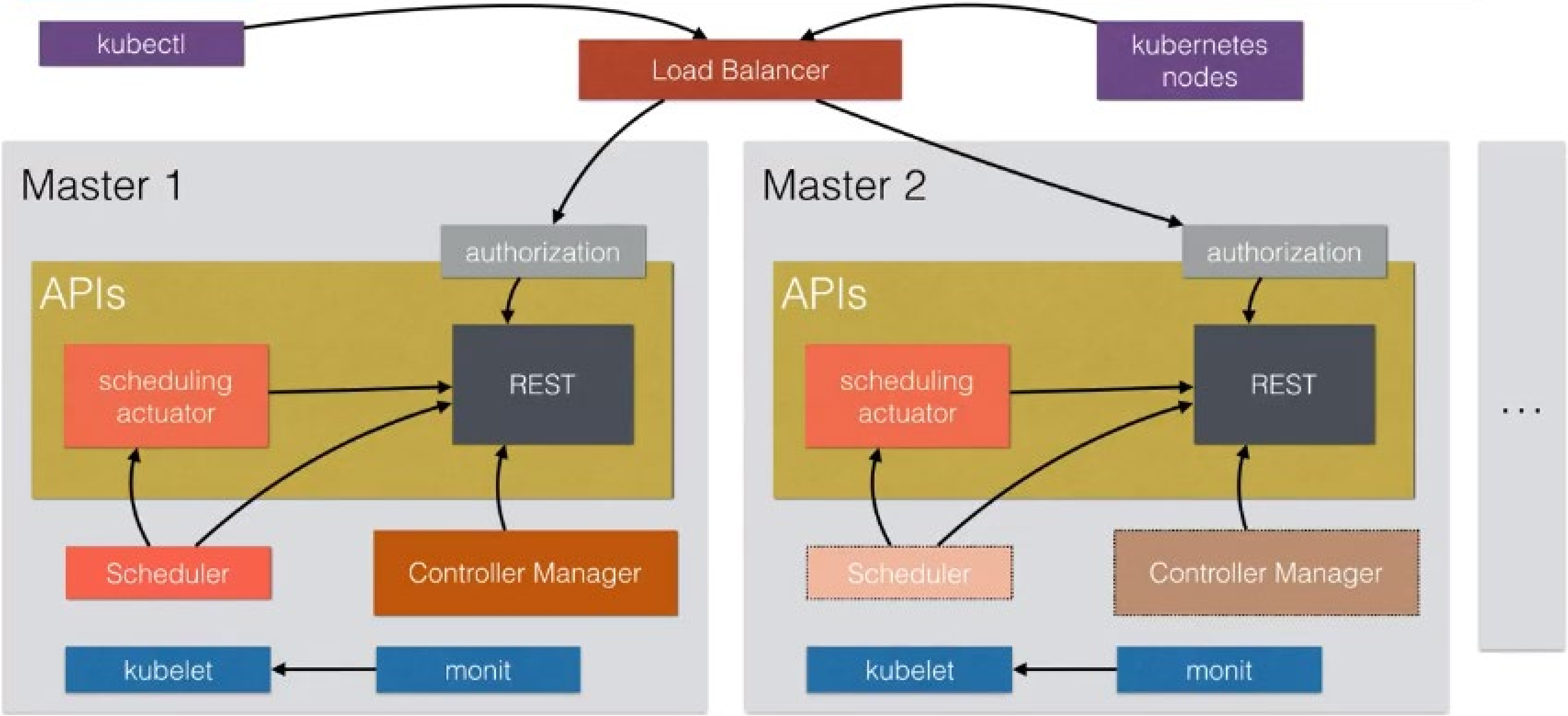
High Availability

- If you're going to run your cluster in production, you're going to want to have all your master services in a **high availability (HA)** setup
- The setup looks like this:
  - **Clustering etcd**: at least run 3 etcd nodes
  - **Replicated API servers** with a LoadBalancer
  - Running multiple instances of the **scheduler** and the **controllers**
    - Only one of them will be the leader, the other ones are on stand-by

Architecture overview - HA



Architecture overview - HA



High Availability

- A cluster like minikube doesn't need HA - it's only a one node cluster

- If you're going to use a production cluster on AWS, **kops** can do the heavy lifting for you

- If you're running on an other cloud platform, have a look at the **kube deployment tools** for that platform

    - **kubeadm** is a tool that is in alpha that can set up a cluster for you

- If you're on a platform without any tooling, have a look at http://kubernetes.io/docs/admin/high-availability/ to implement it yourself

Thank You