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)

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

```
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
```


THANK
YOU!