# Kubernetes

# Advanced Usage

# Overview

| Advanced Kubernetes Usage | Technologies |
|---|---|
| Logging | ElasticSearch, fluentd, Kibana, LogTrail |
| Authentication | OpenID Connect (OIDC), Auth0 |
| Authorization | Kubernetes RBAC |
| Packaging | Helm |
| The Job | Resource |
| Job Scheduling | CronJob |
| Deploying on Kubernetes | Spinnaker |
| Microservices on Kubernetes | Linkerd |
| Federation | kubefed |
| Monitoring | Prometheus |

# Logging with fluentd

- Logging is important to show errors, information and debugging data about the application
- When you only run one app, it is pretty obvious how to look for the logs
    - You'd just open the "app.log"-file to see what's going on
    - Or if deployed as a pod, with kubectl logs
- With Kubernetes 1 application will be running as one or many pods
    - Finding an error will be much more difficult: what pod do you have to look at?

- Up until now you might have been using “kubectl logs” to get the log details of a pod
- To get the correct information, you might have to look up:
  - The pod name
  - The container names
- And run kubectl logs for every container in every pod that is running for an application
- Note: you can use both kubectl log and kubectl logs
- The solution for this problem is to do Log Aggregation

- It’s not Kubernetes specific
  - It’s already applied for years, even with syslog (a standard for message logging since 1980s)
  - Log Aggregation is nowadays often implemented with more modern tools
    - the ELK Stack (ElasticSearch + Logstash + Kibana)
    - Several hosted services like loggly.com, papertrailapp.com

- I'll show you how to setup centralized logging using:
  - Fluentd (For log forwarding)
  - ElasticSearch (For log indexing)
  - Kibana (For visualisation)
  - LogTrail (an easy to use UI to show logs)
- This solution can be easily customized, you can create custom dashboards to show what is important for you
- https://eric-v-documentation.readthedocs.io/en/latest/kubeadvanced.html#setup-aws

# Authentication

- Currently, by default, X509 certificates are used to authenticate yourself to the kubernetes api-server
- These certificates were issued when your cluster was setup for the first time
    - If you’re using RKE, minikube or kops, this was done for you
- In the first Kubernetes course, I showed you how to create new users by generating new certificates and signing them with the Certificate Authority that is used by Kubernetes

# Alternative authentication methods - HTTP Basic authentication

- The kubernetes API server is based on HTTP, so one of the options is to use http basic authentication
- HTTP Basic authentication only requires us sending a username and password to the API server
- While this is very simple to do, a username-password combination is less secure and still difficult to maintain within Kubernetes
- To enable basic authentication, you can use a static password file on the Kubernetes master
- The path to this static password file needs to be passed to the apiserver as an argument:
  - —basic-auth-file=/path/to/somefile

- The file needs to be formatted as:
  - `password,user,uid,"group1,group2,group3"`
- Basic auth has a few downsides:
  - It's currently supported for convenience while the kubernetes team is working on making the more secure methods easier to use
  - To add a user (or change the password), the apiserver needs to be restarted

# Authentication alternatives - proxy

- Another way to handle authentication is to use a proxy
- When using a proxy, you can handlxe the authentication part yourself
- You can write your own authentication mechanism and provide the username, and groups to the kubernetes API once the user is authenticated
- This is a good solution if Kubernetes doesn't support the authentication method you are looking for
- The proxy setup will need the following steps:
    - The proxy needs a client certificate signed by the certificate authority that is passed to the api server using --requestheader-client-ca-file
    - The proxy needs to do handle the authentication (using a form, basic auth, or another mechism)
    - Once the user is authenticated, the proxy needs to forward the request to the kubernetes API server and set an HTTP header with the login
- This login http header is determined by a flag passed to the API server, for example: --requestheader-username-headers=X-Remote-User
    - In this example, the proxy need to set the X-Remote-User after authentication
- --requestheader-group-headers=X-Remote-Group can be used as an argument to set the group header
- —requestheader-extra-headers-prefix=X-Remote-Extra- allows you to set extra headers with extra information about the user

# Authentication alternatives - OpenID

- Another (better) alternative is to use OpenID Connect tokens
- OpenID Connect is built on top of OAuth2
- It allows you to securely authenticate and then receive an ID Token
- This ID Token can be verified whether it really originated from the authentication server, because it's signed (using HMAC SHA256 or RSA)
- This token is a JSON Web Token (JWT)
    - JWT contains known fields like username and optionally groups
- Once this token is obtained it can be used as credential to authenticate to the apiserver
- You can pass --token=<yourtoken> when executing kubectl commands
- kubectl can also automatically renew your token_id when it expires
    - Although this doesn't work with all identity providers
- Using this token, you can also authenticate to the Kubernetes UI
    - To make this easier, I created a reverse proxy that can authenticate you using OpenID Connect and will then pass your token to the UI

# Authentication alternative - Auth0

- Setup Identity Provider (auth0 account)
- Create auth0 client for Kubernetes
- Setup cluster with oidc (OpenID Connect)
- Deploy authentication server (for UI proxy + to hand out bearer tokens)
- Change variables to match auth0
- Create DNS record for auth server
- Try logging in to the UI through authentication server
- https://eric-v-documentation.readthedocs.io/en/latest/kubeadvanced.html#auth0

# Packaging - Helm

- Helm is the package manager for Kubernetes
- Helm helps you to manage Kubernetes applications
- Helm is maintained by the CNCF - The Cloud Native Computing Foundation (together with Kubernetes, fluentd, linkerd, and others)
- Helm was started by Google and Deis
- Deis provides a PaaS on Kubernetes and was bought by Microsoft in April 2017

- Helm uses a packaging format called charts
    - A chart is a collection of files that describe a set of Kubernetes resources
    - A single chart can deploy an app, a piece of software or a database for example
    - It can have dependencies, e.g. to install wordpress chart, you need a mysql chart
    - You can write your own chart to deploy your application on Kubernetes using helm
- Charts use templates that are typically developed by a package maintainer
- They will generate yaml files that Kubernetes understands

- You can think of templates as dynamic yaml files, which can contain logic and variables
- This is an example of a template within a chart:
  - ```
    apiVersion: v1
    kind: ConfigMap
    metadata:
      name: {{ .Release.Name }}-configmap
    data:
      myvalue: "Hello World"
      drink: {{ .Values.favoriteDrink }}
    ```
- The ``favoriteDrink`` value can then be overridden by the user when running helm install
- Overriding values can be useful to make sure the app is configured in a way you want
    ```

# The Job Resource

- Up until now we've always seen pods as long running services that don't stop (like webservers)
- You can also schedule pods as Jobs rather than with a ReplicationController / ReplicaSet
- With a ReplicationController / ReplicaSet a pod will be indefinitely restarted if the pod stops
- With the Job resource, pods are expected to run a specific task and then exit
- There are 3 main types of jobs:
    - 1. Non parallel Jobs
    - 2. Parallel Jobs with fixed completion count
    - 3. Parallel Jobs with work queue

- Non Parallel jobs:
  - The Job resource will monitor the job, and restart the pod if it fails or gets deleted
  - You can still influence this with a restartPolicy attribute
  - When the pod successfully completes, the job itself will be completed
- An example of a non-parallel job:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

- Parallel jobs with a fixed completion count:
  - In this case a Job can run multiple pods in parallel
  - You specify a fixed completion count
  - The job is complete when completion count == successful exited pods
  - To use this, add “completions: ” to the specification of your job
- Parallel jobs with a work queue:
  - In this case the pods should coordinate with themselves (or an external service) to determine what each should work on
  - When any pod terminates with success, no more pods are created and the job is considered completed
    - This because the pods should know themselves when all the work is done, at the point when a pod terminates with success, all pods should terminate, because the job is completed

# Scheduling using the crontab resource

- A Cron Job can schedule Job resources based on time
  - Once at a specified time
    - e.g. Run this Job once, this night at 3 AM
  - Recurrently
    - e.g. Run this Job every night at 3 AM
- CronJob is comparable with crontab in Linux/Unix systems
- CronJob Schedule format (Cron notation, see also https://en.wikipedia.org/wiki/Cron)
- ```
  # ┌──────────────── minute (0 - 59)
  # │ ┌────────────── hour (0 - 23)
  # │ │ ┌──────────── day of month (1 - 31)
  # │ │ │ ┌────────── month (1 - 12)
  # │ │ │ │ ┌──────── day of week (0 - 6) (Sunday to Saturday; 7 is Sunday on some systems)
  # │ │ │ │ │
  # * * * * *
  ```
- Example 1: Every night at 3:20 AM => ``20 3 * * *``
- Example 2: Every 15 minutes => ``*/15 * * * *``
  ```

- An example of a cronjob:
  - 
```
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "25 3 * * *"
  jobTemplate:
    spec:
template:
  spec:
   containers:
   - name: my-cronjob
     image: busybox
     args:
     - /bin/sh
     - -c
     - echo This command runs every night at 3:25 AM
   restartPolicy: OnFailure
```

# Linkerd

- On Kubernetes you can run a lot of microservices on one cluster
- It can quickly become difficult to manage the endpoints of all the different services that make up an application within the cluster:
    - Service discovery in Kubernetes is pretty limited
    - Routing is often on a round-robin based
    - There is no failure handling
        - Except removing pods that fail their healthcheck
    - It's also difficult to visualize the different services
- HTTP APIs can be very basic and there's no failure handling:

    - What happens when app A sends an HTTP GET request to app B, but B is temporary not available?
    - If not written in the APP, the request will just fail and will not retried
- Linkerd can solve these issues for us, and provide us many more features
- Linkerd is a transparent proxy that adds
    - service discovery
    - routing
        - Latency aware load balancing
        - It can shift traffic to do canary deployments
    - failure handling
        - Using retries, deadlines and circuit braking
    - and visibility
        - Using web UIs
- https://eric-v-documentation.readthedocs.io/en/latest/kubeadvanced.html#linkerd