TDDC88/TDDC93: Software Engineering

Lab 2

Kubernetes

Purpose:

- To give you a fundamental understanding and practical experience with Kubernetes, the most widely used container/cluster orchestration system.
- To give you an idea of how to deploy a containerized application on a Kubernetes cluster.
- To give you an idea of how to scale that deployment (up and down)
- To give you an idea of Kubernetes services and how to set up web ingress to reach our service from the browser.

Kubernetes (k8s):

Parts of this tutorial have been heavily adapted from the following sources:

https://kubernetes.io/docs/tutorials/kubernetes-basics/
https://gitlab.liu.se/henhe83/kubernetes-krash
https://kubernetes.io/docs/concepts/overview/

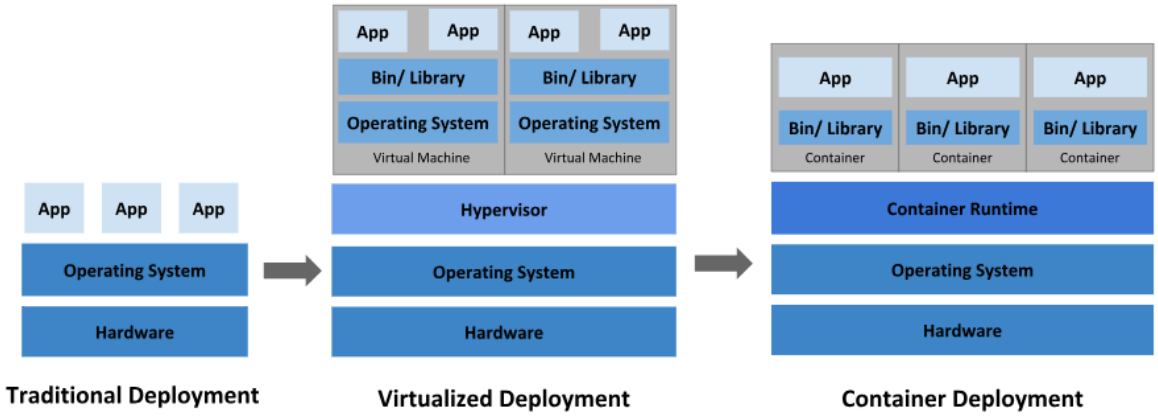
What is Kubernetes (K8s)?

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s". To give you a one-line definition, Kubernetes is an open-source container orchestration engine for automating deployment, scaling, and management of containerized applications.

Why do we need Kubernetes? Why is it useful?

Let's take a look at why Kubernetes is so useful by going back in time.



Traditional deployment era: Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

Virtualized deployment era: As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application.

Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines.

Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

Container deployment era: Containers are like VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Like a VM, a container has its own file system, share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

- **Agile application creation and deployment:** increased ease and efficiency of container image creation compared to VM image use.
- **Continuous development, integration, and deployment:** provides for reliable and frequent container image build and deployment with quick and efficient rollbacks (due to image immutability).
- **Dev and Ops separation of concerns:** create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- **Observability:** not only surfaces OS-level information and metrics, but also application health and other signals.
- **Environmental consistency across development, testing, and production:** runs the same on a laptop as it does in the cloud.
- **Cloud and OS distribution portability:** runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
- **Application-centric management:** raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- **Loosely coupled, distributed, elastic, liberated micro-services:** applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
- **Resource isolation:** predictable application performance.
- **Resource utilization:** high efficiency and density.

What can Kubernetes do for you?

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example: Kubernetes can easily manage a canary deployment for your system.

Kubernetes provides you with:

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes can load balance and distribute the network traffic so that the deployment is stable.

- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

What are some standard Kubernetes components?

Kubernetes is a powerful container orchestration platform that helps manage and deploy containerized applications at scale. It consists of several standard components that work together to provide its functionality. Here are some standard components,

- **Containers**: Containers are lightweight, standalone, and executable software packages that contain everything needed to run a piece of software, including the code, runtime, system tools, libraries (dependencies), and settings.
- **Pods**: Pods are the smallest deployable units in Kubernetes. They can contain one or more closely related containers that share the same network namespace and storage. Pods are used to group containers that need to work together.
- **Nodes**: Nodes are the individual machines (physical or virtual) that make up a Kubernetes cluster. Each node runs the necessary services to manage containers and is responsible for running Pods.
- **Kubelet**: The Kubelet is an agent that runs on each node in the cluster. It ensures that the containers within a Pod are running and healthy. It takes care of container lifecycle management.
- **Ingress**: Ingress is an API object that manages external access to services within a cluster. It provides routing rules to forward HTTP and HTTPS traffic from external sources to the appropriate services.

- **Deployment**: A Deployment is a higher-level resource that provides declarative updates to applications. It ensures that a specified number of replicas of an application are running and handles rolling updates and rollbacks.
- **Replica**: Replicas in Kubernetes refer to the number of identical copies or instances of a Pod or a set of Pods that are running to ensure high availability, fault tolerance, and scalability. The concept of replicas is used to distribute the load and ensure that the application remains accessible even if individual Pods fail.
- **Service**: A Service defines a set of Pods and a policy for how to access them. It provides a stable IP address and DNS name for external access to a set of Pods, even as the Pods' IPs and locations change.
- **Cluster**: A Cluster is the entire Kubernetes system, consisting of a master control plane and a set of worker machines (nodes) that run applications. It provides a unified platform for deploying, managing, and scaling containerized applications.
- **Cluster Controller**: A Cluster Controller is responsible for maintaining the desired state of various cluster resources.
- **Stateless and Stateful**: These terms refer to the nature of applications in Kubernetes. Stateless applications don't rely on the local state of the underlying infrastructure, making them easier to manage and scale. Stateful applications, on the other hand, require persistence (like applications with database volumes) and rely on stable network identities.
- **Persistence**: Persistence in Kubernetes refers to the ability of an application or service to maintain its data and state beyond the lifecycle of individual containers or Pods. This is crucial for applications that require data to be retained even when containers are restarted, rescheduled, or replaced.
- **Token**: In Kubernetes, a token is a piece of information used for authentication and authorization. It grants access to the Kubernetes API and other resources based on user or service identity.
- **Certificate Authority Data**: This refers to the root certificate authority's public key used for encrypting and verifying communication between various components of the Kubernetes cluster. It ensures secure communication within the cluster.

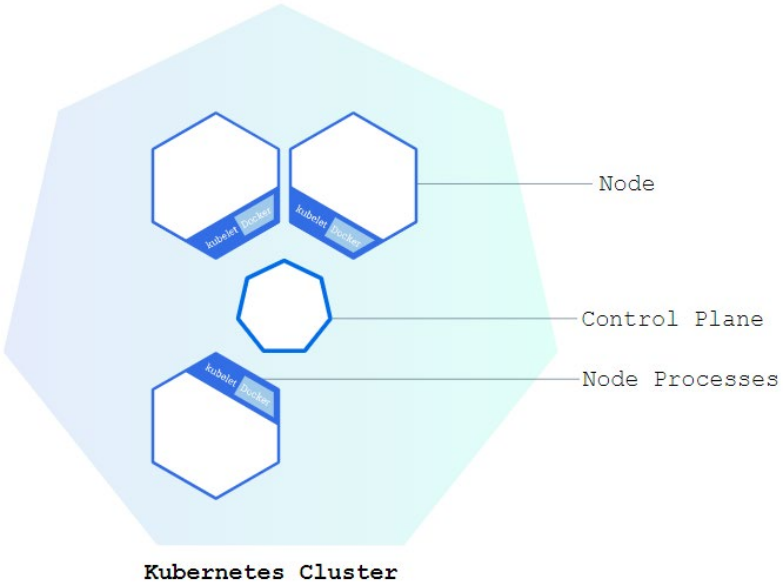
These components work together to create a robust and scalable platform for managing containerized applications in a Kubernetes cluster.

What are Kubernetes Clusters?

Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit. The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them specifically to individual machines. To make use of this new model of deployment, applications need to be packaged in a way that decouples them from individual hosts: they need to be containerized. Containerized applications are more flexible and available than in past deployment models, where applications were installed directly onto specific machines as packages deeply integrated into the host. Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way. Kubernetes is an open-source platform and is production-ready.

A Kubernetes cluster consists of two types of resources:

- The **Control Plane** coordinates the cluster.
- **Nodes** are the workers that run applications.



The Control Plane is responsible for managing the cluster. The Control Plane coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.

A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster. Each node has a Kubelet, which is an agent for managing the node and communicating with the Kubernetes control plane. The node should also have tools for handling container operations, such as containerd or CRI-O. A Kubernetes cluster that handles production traffic should have a minimum of three nodes because if one node goes

down, both an etcd member and a control plane instance are lost, and redundancy is compromised. You can mitigate this risk by adding more control plane nodes.

When you deploy applications on Kubernetes, you tell the control plane to start the application containers. The control plane schedules the containers to run on the cluster's nodes. Node-level components, such as the kubelet, communicate with the control plane using the Kubernetes API, which the control plane exposes. End users can also use the Kubernetes API directly to interact with the cluster.

Getting Started: What you need

- You need to connect to a Linux computer on LiUs network, either via ThinLinc or RDP, or just be physically on campus.
- You need access to a GitLab project configured for kubernetes-access
- You need to read all the theory and definitions in this document.

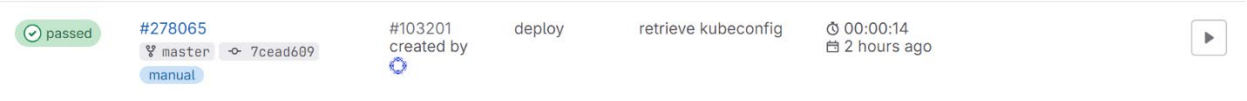
Find the needed information to connect

In order to connect to LiU's Kubernetes instance, you would need an access token. An access token is an authentication artifact that allows a client application to access server resources securely. For this lab, the access tokens you need are placed inside the console logs of the Continuous Integration/Continuous Deployment (CI/CD) jobs pipeline nested within the repository you were given. To find it, go to your repository.

For example, if you are taking TDDC88 (Software Engineering) in 2023 as group F, subgroup 12, go to https://gitlab.liu.se/tddc88-2023/F-12/ (Please note that this link can be different depending on your group number, subgroup number, course code or the year when you take this course).

In the sidebar or left panel, go to `Build -> Jobs`. (You may have to move the mouse pointer to the very left side of the screen with the browser maximized as this panel is normally hidden).

You should see a `manual` job called `retrieve kubeconfig`. Under the `coverage` column, you will see a `play` button on the right side. Click `play` ▶



A console should open showing you the job execution logs. Take your time to read all the services that were started as it will give you an idea of how the environment was created. Once you scroll down, you should see three dashes (---). Copy everything from `apiVersion: v1` to the end of your personal `token.` Do not copy the three dashes or the informative text (like cleaning up project, job succeeded) at the end of the console. This clipboard content is your *personal* Kubernetes configuration file you need to connect to the cluster instance. In this console output, here are a few of the listed components,

- **Kubernetes cluster name** - Just an arbitrary name given to the cluster.
- **Token** – Your personal access token used to communicate with the cluster.
- **CA Certificate** - A certificate to verify that you are talking to the correct cluster.
- **Project namespace** - The namespace you are granted access to.

Take note of these parameters and keep the browser window open, we will use them later. The `namespace` parameter, in particular will be used repeatedly.

Installing and configuring `kubectl`

If you are running on a Linux lab workstation on campus or via Thinlinc/SSH, should not install kubectl as we have already installed it for you. Instead, use the following command - `module add courses/TDDC88` in a terminal to add it to your environment. If that module can't load, try `module add prog/kubectl` and report the error to the course staff.

To use `kubectl` we first need to provide the necessary data to connect to the cluster. Let's configure it. Kubectl stores its config in `~/.kube`, so create that first:

```
mkdir -p ~/.kube
```

Next, we create the cluster configuration. Start by creating the configuration file in a text editor. We have used `Kate`, but you can use whichever editor you are comfortable with:

```
kate ~/.kube/config
```

Paste what you copied from the Gitlab CI/CD Jobs Console Output into this text editor. It should look something like this (below). Once done, save the file. DO NOT copy the same file as below!

```
apiVersion: v1
clusters:
- cluster:
```

```
    server: https://10.216.0.8
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURBRENDQWVpZ0F3SUJBZ0lVV3N0
Q01oU0QvZ2FJQ0F4bWtPazVIMDc0cXRrd0RRWUpLb1pJaHZjTkFRUwKQlFBd0dERVdN
QlFHQTFVRUF4TU5hM1ZpWlhKdVpYUmxjeTFqWVRBZUZ3MHlNekE0TWpJd056TTBNREJh
RncwegpNekE0TVRrd056TTBNREJhTUJneEZqQVVCZ05WQkFNVERXdDFZbVZ5Ym1WMFpY
TXRZMkV3Z2dFaU1BMEdDU3FHClNJYjNEUUVCQVFVQUE0SUJED0F3Z2dFS0FvSUJBUUMy
bDN0VWZ1VElCQXRrb2VvTzh2VEQ4YktiLzRjSW41QnoKSjRFbjNFcVJ3NUxNVE03Z3lW
cnJkeHNVNlZCdm9XUzNNamRKbDFJQnlzQWk5K05XaE15RDJPZUJCMDhUenZLVApIZ0lV
WWRGM3VYb0ZFN2lPOWxXWXNsNk03dlk2RGxpeXdaL0NSdUY3RmtDR0ZEMnZTNFM2QmRw
ZlJ0cGlxaXkrCmdiaWcweGNUd0xwaFJxdGZrT3NuNExOU3lXWkN5aHU0YW1HQ3d1dmxH
QjFXTHVvOFJwVUxSTEI3NDFCdDhCd0MKMWRoVEhPM1FUNTlQS0t5SDZOemV2dytRUFFU
R245M2VtT0w1SXlOYUJvSHFsV3NPZDFEdkVhdHdNWUNtUnRPZgorZ1Z0eG1sNnU2T3VH
cGxmMFlkSGp1TGUrNHpHUkNPN0QxZVl0NDRaNnROTlVmK2ZkdlduQWdNQkFBR2pRakJB
Ck1BNEdBMVVkRHdFQi93UUVBd0lCQmpBUEJnTlZIUk1CQWY4RUJUQURBUUgvTUIwR0Ex
VWREZ1FXQkJSQTlHVUwKUnA2ck42M29lU3ZURjZUR1FIV2FzVEFOQmdrcWhraUc5dzBC
QVFzRkFBT0NBUUVBcWIvQzk1emVBUzQyR1BsMgpLRW1RdGltb2tkUTBVMmdlNCtPakRn
U0JueDRLK3pQdVlRVHBaaWxCYXh0UFZrQVA4Rnp1TFNOLzhiVjltdHV5CitjTEpyeFND
YWZDWFFmLzFYdmhWYlI1eFNJZjJudEc1ZFFFS2VoNlBKYXNtbEk3SEltWFBtSWYvemEy
dFRvUmgKb1dmcFd3cXhiUEI5OUg5RkhiQlJJcVQ0UG5hZVlpTkFrZ2FzWmRsVW1FRzhN
clBlYVlWcWRNdCtrQVJGUWNjdQpWTW9GWFA5aVlrTXFNNG9SQnBlY3hqSGluVCtYMDRK
Z2dRY1dFSVE0R0hveUhXZjRvc3hxQk9UWmNtN2REc2xUClFJU0pDRGZieWVzcFNMcEFu
TUZmY3dMM21VWHBMaldZMnQ3ckRoRmRRZHRpdjJDOS9SWlVHelVEN3lSNitsQ2IKY3dI
eUhBPT0KLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
  name: tddc88
contexts:
- context:
    cluster: tddc88
    namespace: tddc88-ht23-test-group-0
    user: tddc88
  name: tddc88
current-context: tddc88
kind: Config
preferences: {}
users:
- name: tddc88
  user:
    token:
eyJhbGciOiJSUzI1NiIsImtpZCI6IkRnQzdhbUFPbnZZclN2bWdqU09CZVY2VDh2enBk
```

```
elprRUp2TFRmZUZvbmsifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50I
iwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJ0ZGRjODgta
HQyMy1OZXN0LWdyb3VwLTAiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlY
3JldC5uYW1lIjoiYWRtaW4tdXNlci10b2tlbiIsImt1YmVybmV0ZXMuaW8vc2VydmljZ
WFjY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJhZG1pbiIsImt1YmVybmV0ZXMua
W8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjU3MzMxYWJjLTM3M
WMtNDJhNi04NTM3LTA2Nzc3MDgxMzE2YyIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb
```

Make sure it's working

We can list the resources we use to make sure everything is working so far:

```
$ kubectl get all

No resources found in tddc88-2023-group7 namespace.
```

We can also check what resources we have available:

```
$ kubectl describe quota

Name:                   resource-quota
Namespace:              tddc88-2023-group7
Resource                Used  Hard
--------                ----  ----
configmaps              0     5
count/deployments.apps  1     5
cpu                     10m   1
memory                  42Mi  1Gi
persistentvolumeclaims  0     2
pods                    1     10
replicationcontrollers  0     0
requests.storage        0     4Gi
resourcequotas          1     1
secrets                 2     5
services.loadbalancers  0     0
services.nodeports      0     0
```

Deploy a simple *Hello World* application

This is a loose adoption of the HelloMinikube tutorial from Kubernetes documentation, which you probably should read. Let's deploy a simple *Hello World* application to the cluster and see if we can access it from our web browser. In this Lab, we'll use `.yaml` files to specify our deployment particulars.

Create the deployment

You can create and manage a Deployment by using the Kubernetes command line interface, **kubectl**. Kubectl uses the Kubernetes API to interact with the cluster. In this section, you'll learn the most common kubectl commands needed to create Deployments that run your applications on a Kubernetes cluster.

The common format of a kubectl command is: `kubectl action resource`

This performs the specified *action* (like `create`, `describe`, or `delete`) on the specified *resource* (like `node` or `deployment`). You can use `--help` after the subcommand to get additional info about possible parameters (for example: `kubectl get nodes --help`).

Check that `kubectl` is configured to talk to your cluster, by running the `kubectl version` command.

Now, we need to create a file, `hello-deployment.yaml` that will hold the details of our deployment specifications. Since the formatting of this file is important as Kubernetes validates JSON/YAML files before deployment, we will run the following command to directly fetch the file,

```
wget https://www.ida.liu.se/~TDDC88/labs/Labs2023/Lab2/hello-deployment.yaml
```

Once you fetch this file, open it using your desired text editor and analyze all the parameters that are involved in the deployment. This will be useful for the examination tasks at the end. The file contents should look something like this (If you get any validation errors, delete the file, and fetch it again using the command above).

(Do not copy this file. For understanding purposes only)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deployment
  labels:
    app: hello
spec:
  replicas: 1 # How many replicas we want
  selector:   # How do we indentify pods we should manage?
    matchLabels:
      app: hello # Match all nodes with this label
  template: # How should pods within this deployment be created?
    metadata:
      labels:
        app: hello # Specify a label, used in the above selector
    spec:
      containers:
        - name: hello
          image: crccheck/hello-world #Specify what image we want to
use
          ports:
          - containerPort: 8000 # Specify what port our container is
listening on.
            protocol: TCP
          resources: # Specify what resources we need...
            limits: # ...both absolute limits...
              cpu: 100m
              memory: 80Mi
            requests: # ...and what we probably will get by with.
              cpu: 10m
              memory: 42Mi
```

Now that we have a specification for our deployment, let's apply it!

```
$ kubectl apply -f hello-deployment.yaml

deployment.apps/hello-deployment created
```

Now, Kubernetes will pull the container specified in your deployment, and start it according to our specifications. We can check the state using:

```
$ kubectl get all

NAME                                     READY   STATUS    RESTARTS   AGE
pod/hello-deployment-69f546bc9c-jctpw    1/1     Running   0          2m8s

NAME                                READY  UP-TO-DATE   AVAILABLE   AGE
deployment.apps/hello-deployment    1/1        1            1        6m42s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/hello-deployment-69f546bc9c    1         1         1       2m8s
```

This command shows you a lot of information about the current state of your work. Note that if you run it directly after your `kubectl apply`, your `STATUS` might be something other than `Running`, like for example `ContainerCreating`. If something looks weird, start by checking the logs given by

```
$ kubectl get events
```

Create the service

A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service is defined using YAML or JSON, like all Kubernetes object manifests. The set of Pods targeted by a Service is usually determined by a *label selector.*

Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic. Services can be exposed in different ways by specifying a `type` in the spec of the Service.

- *ClusterIP (default)* - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.
- *NodePort* - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using `<NodeIP>:<NodePort>`. Superset of ClusterIP.
- *LoadBalancer* - Creates an external load balancer in the current cloud (if supported) and assigns a fixed, external IP to the Service. Superset of NodePort.
- ExternalName - Maps the Service to the contents of the `externalName` field (e.g. `foo.bar.example.com`), by returning a `CNAME` record with its value. No

proxying of any kind is set up. This type requires v1.7 or higher of `kube-dns`, or CoreDNS version 0.0.8 or higher.

Now, on top of our `Deployment`, we want a `Service` to encapsulate it. Similarly, as before we need a file, `hello-service.yaml` that will hold the details of our service specification. Since the formatting of this file is important as Kubernetes validates JSON/YAML files before deployment, we will run the following command to directly fetch the file,

```
wget https://www.ida.liu.se/~TDDC88/labs/Labs2023/Lab2/hello-service.yaml
```

Once you fetch this file, open it using your desired text editor and analyze all the parameters that are involved in the service formation. The file contents should look something like this (If you get any validation errors, delete the file, and fetch it again using the command above).

(Do not copy this file. For understanding purposes only)

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hello
  name: hello-service
spec:
  ports:
  - name: web # This port is named `web`...
    port: 80  # ... and should be exposed on port 80, ...
    protocol: TCP # ... using TCP ...
    targetPort: 8000 # ... to expose port 8000 of internal pods ...
  selector: # ... designated by the selector 'hello' ...
    app: hello
  type: ClusterIP # ... to an internal cluster IP address
```

Let's apply this service definition:

```
$ kubectl apply -f hello-service.yaml

service/hello-service created

$ kubectl get all

NAME                                     READY   STATUS    RESTARTS   AGE
pod/hello-deployment-69f546bc9c-jctpw    1/1     Running   0          12m

NAME                    TYPE  CLUSTER-IP  EXTERNAL-IP   PORT(S)   AGE
service/hello-service   ClusterIP 10.64.215.97  <none>  80/TCP    4s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/hello-deployment    1/1     1            1           17m

NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/hello-deployment-69f546bc9c   1     1         1       12m
```

Try opening the `CLUSTER-IP` given to the service in a web browser. It won't work!

Let's fix that - To reach the service from our web browser, we need to configure ingress, telling the Kubernetes cluster to redirect traffic to your service. For this, we need a file, `hello-ingress.yaml` that holds the details of our ingress specifications. Again, since the formatting of this file is important as Kubernetes validates JSON/YAML files before deployment, we will run the following command to directly fetch the file,

```
wget https://www.ida.liu.se/~TDDC88/labs/Labs2023/Lab2/hello-
ingress.yaml
```

Once you fetch this file, open it using your desired text editor and analyze all the parameters that are involved in the ingress formation. The file contents should look something like this

(Do not copy this file. For understanding purposes only)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx-public
  name: hello-public
  labels:
    app: hello
spec:
```

```
  rules:
    - host: namespace.kubernetes-public.it.liu.se
      http:
        paths:
          - pathType: Prefix
            path: /
            backend:
              service:
                name: hello-service
                port:
                  number: 80
```

Pods that are running inside Kubernetes are running on a private, isolated network. By default, they are visible from other pods and services within the same Kubernetes cluster, but not outside that network. When we use `kubectl`, we're interacting through an API endpoint to communicate with our application.

IMPORTANT:

Note the `host` parameter in the `hello-ingress.yaml` file. You need to change this parameter as per the instruction below! Use a text editor of your choice to do so. The easiest way would be to use the Kate text editor as `kate hello-ingress.yaml`

You can get the YOUR_NAMESPACE parameter from the configuration file we created at the beginning of this lab. If your namespace was `tddc88-ht23-test-group-0`, then the `host` parameter will become

`tddc88-ht23-test-group-0.kubernetes-public.it.liu.se`

In accordance with your personal configuration file, change the `host` parameter to YOUR_NAMESPACE`.kubernetes-public.it.liu.se` and then save the file. Note that Kate does not auto-save the file like Visual Studio.

Let's apply our configuration!

```
$ kubectl apply -f hello-ingress.yaml
```

Now, open our ingress in a web browser using the `host` parameter. We now have a complete web application deployed via Kubernetes! This link should be accessible from anywhere in the world!

Easy scaling

When you need to scale horizontally, it's as easy as the following (scale down the same way)

```
$ kubectl scale deploy hello-deployment --replicas=3
```

The load balancer will then manage everything for you. You can see the replicas using:

```
$ kubectl get all
$ kubectl get pods
```

Task: You are a software developer working for SJ (Statens Järnvägar). Management has decided to create a new ticket booking system capable of dynamically handling peak time network traffic. The system should also automatically use lesser resources when the network traffic decreases. You've read about Kubernetes and it's ability to automatically scale applications (along with it's load balancing capabilities). You make this suggestion to your team, and they are impressed with your initiative – The only problem is that sometimes, the applications running on the pods for long periods of time eventually transition to broken states (due to memory leaks, resource corruption, concurrency issues, and so on). The only way these applications can recover is if they are restarted. At the peak, you have 1000 Kubernetes pods running, and you cannot manually restart all of them. How would you fix this problem? Also, implement your application recovery mechanism on the current LiU Kubernetes cluster. (Please note that you don't have to deploy a booking system. Find what mechanism would work in this situation along with it's corresponding YAML file and deploy it to the cluster).

Once done with the above task, you are ready for the demonstration. You may also be asked a few questions to test your knowledge of Kubernetes (sample questions are listed at the end of this document).

I've finished demonstrating, now what?

(Only do this step when you have finished demonstrating to a Lab Assistant) To ensure that other students in the lab can successfully deploy their applications to LiU's Kubernetes Cluster, you are required to delete your deployments, services, and ingresses. Run the following commands (in order):

```
$ kubectl delete -f hello-ingress.yaml

$ kubectl delete -f hello-service.yaml

$ kubectl delete -f hello-deployment.yaml

$ kubectl get all
No resources found in tddc88-2023-group7 namespace.
```

(Also delete all deployments in connection with the task in a similar manner)

That's it. You're done with an introduction to Kubernetes, the world's most popular and widely used container/cluster orchestrator. We just fetched a container from the Docker Hub, deployed it on LiU's Kubernetes Instance and set up web ingress for worldwide access to our application. We also scaled our application, which is normally done in the real world to handle peak demand. Finally, we implemented a recovery mechanism in case something goes wrong with our application during deployment.

The next step could be complete automated image building (using docker), automated testing, and deployment using Gitlab CI/CD. While the basics on Docker were covered in the first lab, the remaining aspects will be covered in the upcoming labs in the TDDC88 Series.

In case you need help with K8s parameters:

If you want more information on a specific parameter, try using `explain`:

```
$ kubectl explain deploy.spec.replicas

KIND:     Deployment
VERSION:  extensions/v1beta1
FIELD:    replicas <integer>
DESCRIPTION:
     Number of desired pods. This is a pointer to distinguish
between explicit
     zero and not specified. Defaults to 1.
```

Examination

You should understand everything in each part deeply to be able to answer the questions that assistants ask you. Furthermore, you must answer all questions. Contact your assistant during a lab session and be ready to answer questions concerning what you did when demonstrating. You don't need to hand in anything.

Questions you must be prepared to answer:

- What is container/cluster orchestration in terms of Kubernetes (K8s)? Why is this so important when it comes to DevOps?
- How are Kubernetes and Docker related?
- What is the difference between K8s nodes and pods?
- Can a single K8s cluster have more than one control plane?
- What is the function of a load balancer in K8s? Is this function activated automatically when you scale your deployment?
- What is the difference between stateless and stateful applications in K8s?