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/

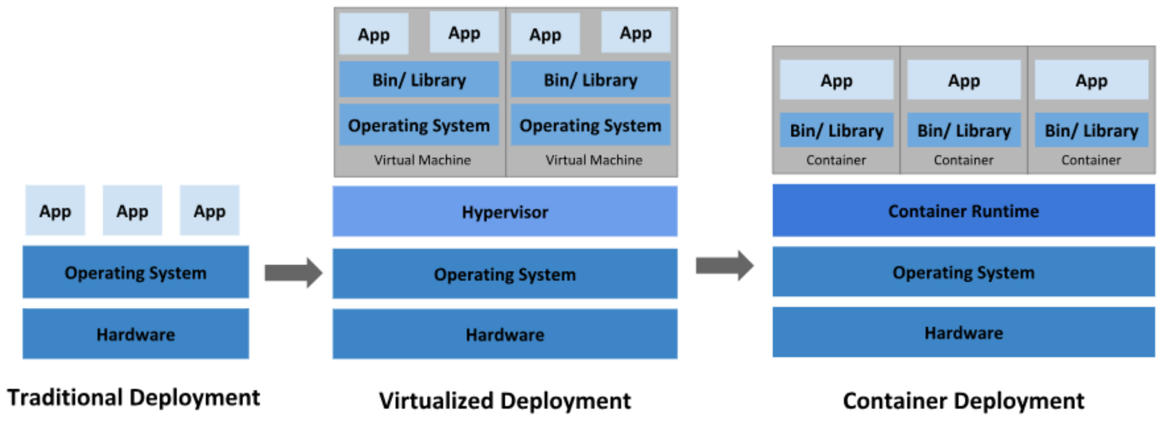
What is Kubernetes (K8s)?

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.

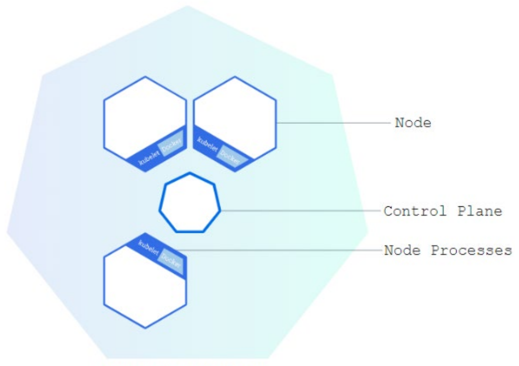
These components work together to create a robust and scalable platform for managing containerized applications in a Kubernetes cluster.

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.



Kubernetes Cluster

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 to read all the theory and definitions in this document.

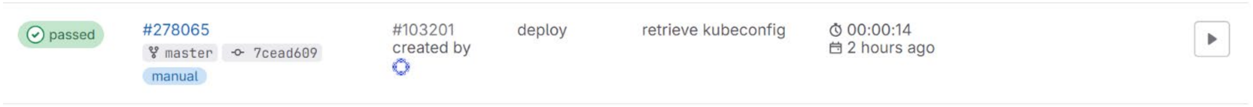
Find the needed information to connect

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 2024 as group A, subgroup 12, go to https://gitlab.liu.se/tddc88-ht24/a-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` ▶


passed | #278065 master 7cead609 manual | #103201 created by | deploy | retrieve kubeconfig | 00:00:14 2 hours ago | ▶

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 EMACS, but you can use whichever editor you are comfortable with:

```
touch ~/.kube/config && emacs ~/.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 this file below!

```
apiVersion: v1
clusters:
- cluster:
    server: https://tddc88-ht24-test.course.kubernetes.it.liu.se
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURCVENDQWUyZ0F3SUJBZ0lJV1M0Yjgra1JNYTB3RFFZ
SktvWklodmNOQVFFTEJRQXdGVEVUTUJFR0ExVUUKQXhNS2EzVmlaWEp1WlhSbGN6QWVGdzB5TkRBM01UVXdP
VEV3TVROYUZ3MHpOREEzTVRNd09URXdNVE5hTUJVeApFekFSQmdOVkJBTVRDbXQxWW1WeWJtVjBaWE13Z2dF
aU1BMEdDU3FHU0liM0RRRUJBUVVBQTRJQkR3QXdnZ0VLCkFvSUJBUURLWXY4OUkxaDBWM1pzZ050YUtyTjlB
TUwvZXRBQWRqSHVyVjU2VXE5cmdPenhHUDVYMVJ2czNqZ2UKOXpRNlZjaXRWTEkwSXRUL3hzRUsycGYzRzhH
a1VVM3FOOXdFblRTOW5yazlVRFVxcGdKQWMzM2dMVHZWaGs5cgppTDl3YTViZUYvZUFvVTZrSGgxNzBacFR3
V2NSY0pnUWNBRFJUUk5tZGGtuK0J3NTRhZ1Zsa3VTalBGUGlhQ3pNCkUydTVoa3pFc1RWWHl1ZDVwckNON29G
dmpuV0RwcWlraStrYnUvQUhHZDNxYUp3QmhrbDZ4YWQzS0FYWVdmQXcKRWJjc1lJQ2s3d0RiSUNYalV0MUVI
OGthaVZrSzVQTzNpaGgxR3d0Y1Z6dlBhMG5SUGdZQUJDbGlmNzBWclVFYQpaVzF3Q1dKMWIxZDRPdk4rdDFP
RGpMWlY1L1V4QWdNQkFBR2pXVEJYTUE0R0ExVWREd0VCL3dRRUF3SUNwREFQCkJnTlZIUk1CQWY4RUJUQURB
UUgvTUIwR0ExVWREZ1FXQkJTM01iMFV5cytNdE1WemNadTQvV0d4Vm00N1ZUQVYKQmdOVkhSRUVEakFNZ2dw
cmRXSmxjbTVsZEdWek1BMEdDU3FHU0liM0RRRUJDd1VBQTRJQkFRQzlSYjJ6V3JaMAoyVGdKVnJWL215MHI0
bmNRYzduMFFmTWZtdUJSNktVY005c2dFYnpVRUs5NDlHQnVwZ1M4b1ZoblBwV3d4bHpFCkpydzVpKzBGc29s
RDFaMWxNL0JBeWNRRlZZbmZNNXFwNzJXcGl2ZUlicmJpeDN1SWVabU91d2hacEZzVVY5VXAKRTVxa3lSRkpi
NzhmM2hIazd0aElrNGF3aS9lTkpnR1RnQitGaGhLVU9Sd3VJU0gwY2JkeEtNQlZpZ0c4V1I3UwpCVFFnb0ox
Z3NKRWpwZVhGYThiKzlOcDREeHk3S1hoUVNFdzFkaFJlNzN3KzVIaXlOUXlXTndTaXpQV1FiOG0vCjlEckxM
aVVUZUlYaGM3L2NYZndUMmE5Mmd4eUVWbFdqSy9lU1dnRGdXQkI5QWdzM0VzcTE3Z1ZPMzdIalVCckYKdWNI
Zk44VnRKR1pBCi0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K
```

```
  name: tddc88
contexts:
- context:
    cluster: tddc88
    namespace: tddc88-ht24-test-group-0
    user: tddc88
  name: tddc88
current-context: tddc88
kind: Config
preferences: {}
users:
- name: tddc88
  user:
    token:
eyJhbGciOiJSUzI1NiIsImtpZCI6InJrNTBUc2RMWUdzVElXdFlscEhKYjMyb3FBem9iZU9pLVNMekZFSnZ4
LWMifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlY
WNjb3VudC9uYW1lc3BhY2UiOiJ0ZGRjODgtaHQyNC10ZXN0LWdyb3VwLTAiLCJrdWJlcm5ldGVzLmlvL3Nlc
nZpY2VhY2NvdW50L3NlY3JldC5uYW1lIjoiYWRtaW4tdXNlci10b2tlbiIsImt1YmVybmV0ZXMuaW8vc2Vyd
mljZWFjY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJhZG1pbiIsImt1YmVybmV0ZXMuaW8vc2VydmljZ
WFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6ImVlODNjNjQwLTdlNjEtNDZkZi1iM2YyLWJkZjhhODU4M
zRkNCIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDp0ZGRjODgtaHQyNC10ZXN0LWdyb3VwLTA6YWRta
W4ifQ.IAC5bTc5RTbELgITOwHTyBlvD7LdlUagCMgheKSndT8WKEJGDqA1UyDHWLGtkzwGTdE1hqZFQmKqP8
WaOx2G35Y8abXo75TorpopF84rWnapvFk0EinymlD9aVC5hE1Yp2fuEIUNEiZ1L6V6C34O5MmuMi3gZaWmKU
JVit3GwinZXHu-iPwmIcXagt36I4nEER93-
fany4u_Qs_1lWB018NU374QUa44tVuyo4yuIytk3iVMQvXWX32k8WYZoA15c3Bj6xCUkSwjzquQvhp9iTTAc
DI8JG5FA1MW640kc6pZEf7eNh2ie4Uptm2A5lgzxairAHQJ5OSmYwSZ76UWXQ
```

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-ht24-test-group-0 namespace.
```

If you face indentation or validation errors, use the tool at https://bit.ly/tddc88-yaml-tool and follow the instructions at the end of the page (after the file has been generated).

We can also check what resources we have available:

```
$ kubectl describe quota

Name:                   resource-quota
Namespace:              tddc88-ht24-test-group-0
Resource                Used  Hard
--------                ----  ----
configmaps              1     20
count/deployments.apps  0     10
cpu                     0     300m
memory                  0     256Mi
persistentvolumeclaims  0     10
pods                    0     30
replicationcontrollers  0     0
requests.storage        0     1Gi
secrets                 2     20
services.loadbalancers  0     0
services.nodeports      0     0
```

LAB: Deploy a Dockerized Application to a Kubernetes Cluster

***Your mission, should you choose to accept it**, is as follows:*

You are Agent Andersson, a covert operative working for the Swedish Security Service (SÄPO). Your mission has brought you deep into the heart of Moscow, Russia where you successfully infiltrated the Kremlin under the guise of a high-level diplomat. After months of painstaking surveillance and intelligence gathering, you have discovered something that could alter the balance of power in Northern Europe.

In a dimly lit, secure briefing room, you unearth a chilling operational document marked with the ominous classification: Secret/NOFORN (No Foreign Distribution). The document outlines Russia's plan to deploy ELSA, a sophisticated malware component originally developed by the CIA, to track and monitor high-profile targets within Sweden. The implications are staggering – a direct threat to Swedish national security, the privacy of its citizens, and the integrity of its government.

The Russians have reverse-engineered ELSA to enhance its capabilities, allowing them to geolocate targets with unprecedented precision. The plan is to infiltrate critical infrastructure, gather sensitive data, and destabilize Sweden from within. As you digest the gravity of this information, you must act swiftly to prevent this digital invasion. Returning to Sweden with the document is not an option; the risk of exposure is too great. Instead, you must establish an online dead-drop – a secure, anonymous digital channel to transmit the critical intelligence to your analysts back home.

During your SÄPO training, you learned how to deploy applications in real-time using Docker and Kubernetes. Your handler at SÄPO explained to you how the application, called **Dead-Drop**, could be used to securely upload files protected with a username and password in a temporary SQLite database. The following file (`app.py`) contains the core functionality for the Dead-Drop Application and can be found at https://gitlab.liu.se/valco79/dead-drop

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

```
import os
from flask import Flask, request, redirect, url_for, session, render_template,
make_response, send_file
from flask_sqlalchemy import SQLAlchemy
from werkzeug.utils import secure_filename
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__)
app.secret_key = 'your_secret_key'

# Configure the SQLite database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
app.config['UPLOAD_FOLDER'] = 'uploads'  # Folder to store uploaded files
# Ensure the upload directory exists
upload_folder = app.config['UPLOAD_FOLDER']
if not os.path.exists(upload_folder):
    os.makedirs(upload_folder)

db = SQLAlchemy(app)

# User model
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password = db.Column(db.String(120), nullable=False)
    filename = db.Column(db.String(255))  # Store the filename for the uploaded file

# Helper function to check allowed file extensions
def allowed_file(filename):
    allowed_extensions = {'txt', 'pdf', 'doc', 'docx', 'zip'}
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in
allowed_extensions

@app.route('/style.css')
def serve_css():
    return send_file('style.css', mimetype='text/css')

@app.route('/')
def home():
    if 'username' in session:
        user = User.query.filter_by(username=session['username']).first()
        if user:
            if user.filename:
                return render_template('home.html', username=session['username'],
filename=user.filename)
            else:
                session.clear()
                return redirect(url_for('login'))
    return redirect(url_for('login'))

@app.route('/login', methods=['GET', 'POST'])
def login():
    message = None
    message_type = None
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        user = User.query.filter_by(username=username).first()

        if user and check_password_hash(user.password, password):
            session['username'] = user.username
            message = "Login Successful. Redirecting..."
            message_type = "success"
        else:
            message = "Login Failed. Please try again."
            message_type = "error"

    return render_template('login.html', message=message, message_type=message_type)

@app.route('/signup', methods=['GET', 'POST'])
def signup():
    message = None
    message_type = None
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        hashed_password = generate_password_hash(password, method='pbkdf2:sha256')

        file = request.files['file']
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            new_user = User(username=username, password=hashed_password,
filename=filename)
            db.session.add(new_user)
            db.session.commit()
            message = "Secure File Upload Successful!"
            message_type = "success"

        else:
            message = "Secure File Upload Failed!"
            message_type = "error"

    return render_template('signup.html', message=message,
message_type=message_type)

@app.route('/logout')
def logout():
    session.pop('username', None)
    resp = make_response(redirect(url_for('home')))
    resp.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate'
    resp.headers['Vary'] = '*'
    return resp

@app.route('/download')
def download():
    if 'username' in session:
        user = User.query.filter_by(username=session['username']).first()
        if user:
            if user.filename:
                return send_file(os.path.join(app.config['UPLOAD_FOLDER'],
user.filename), as_attachment=True)
        else:
            session.clear()
            redirect(url_for('login'))
    return redirect(url_for('login'))

if __name__ == '__main__':
    with app.app_context():
        db.create_all()
    app.run(host='0.0.0.0', port=5000, debug=False)
```

The Dockerfile for this application is as follows. Again, this file is provided for understanding purposes only. There's no need to copy this file.

```
FROM python:3.8-slim
WORKDIR /app
COPY . /app
RUN pip install Flask SQLAlchemy Flask-SQLAlchemy Werkzeug
EXPOSE 5000
CMD ["python", "app.py"]
```

This way, the Dead-Drop application provides a secure, temporary, and traceless method for transmitting critical intelligence. For ease of convenience, your handler published this application to DockerHub at https://hub.docker.com/r/valencycolaco/dead-drop. Note that all links to SÄPO are removed to maintain plausible deniability. Instead, the application carries the emblem/seal of the International Intelligence Agency, a fictional organization 'hosted' by the United Nations International Command.

Now that we have understood how Dead-Drop works, we need to fetch the ELSA dossier as:

```
wget https://gitlab.liu.se/valco79/dead-drop/-/raw/main/Elsa_User_Manual.pdf
```

Now that we have the dossier, we need to deploy the Dead-Drop Application to a Kubernetes Cluster which the analysts in Sweden can securely access.

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 an ***action****:* create, describe, or delete, on the ***resource***: 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 `kubectl version`

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

```
wget https://gitlab.liu.se/valco79/dead-drop/-/raw/main/K8s%20Files/dead-drop-
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: dead-drop-deployment
  labels:
    app: dead-drop
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dead-drop
  template:
    metadata:
      labels:
        app: dead-drop
        group: group_name # Enter your group name here, like e-7
    spec:
      containers:
        - name: dead-drop-group_name # Replace group_name like dead-drop-e-7
          image: valencycolaco/dead-drop
          ports:
            - containerPort: 8000
              protocol: TCP
          resources:
            limits:
              cpu: 100m
              memory: 80Mi
            requests:
              cpu: 10m
              memory: 42Mi
```

This file fetches the application from DockerHub and prepares it for deployment to the cluster. Now that we have a specification for our deployment, let's apply it!

Before you can apply the file, you must change the group_name parameter in the deployment YAML file using a text editor of your choice.

```
$ kubectl apply -f dead-drop-deployment.yaml
deployment.apps/dead-drop-deployment created
```

Now, Kubernetes will pull the container specified in your deployment and start it according to our specifications. The process may take 60 seconds. We can check the state using:

```
$ kubectl get all

NAME                                        READY   STATUS    RESTARTS   AGE
pod/dead-drop-deployment-5b4c6947b4-w6f78   1/1     Running   0          36s

NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/dead-drop-deployment   1/1     1            1           37s

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/dead-drop-deployment-5b4c6947b4   1         1         1       36s
```

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, `dead-drop-service.yaml` that will hold the details of our service specification. Since the formatting of this file is important as Kubernetes validates YAML files before deployment, we will run the following command to directly fetch the file,

```
wget https://gitlab.liu.se/valco79/dead-drop/-/raw/main/K8s%20Files/dead-drop-
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)

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: dead-drop
  name: dead-drop-service
spec:
  ports:
  - name: web
    port: 80
    protocol: TCP
    targetPort: 5000  # Update to match the port exposed by your Flask app
  selector:
    app: dead-drop
  type: ClusterIP
```

Let's apply this service definition:

```
$ kubectl apply -f dead-drop-service.yaml

service/dead-drop-service created

$ kubectl get all

NAME                                        READY   STATUS    RESTARTS   AGE
pod/dead-drop-deployment-5b4c6947b4-w6f78   1/1     Running   0          7m34s

NAME                        TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/dead-drop-service   ClusterIP   10.64.95.76    <none>        80/TCP    37s

NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/dead-drop-deployment   1/1     1            1           7m35s

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/dead-drop-deployment-5b4c6947b4   1         1         1       7m34s
```

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 any web browser, we need to configure ingress, telling the Kubernetes cluster to redirect traffic to your service. For this, we need a file, `dead-drop-ingress.yaml` that holds the details of our ingress specifications. Again, since the formatting of this file is important as Kubernetes validates YAML files before deployment, we will run the following command to directly fetch the file,

```
wget https://gitlab.liu.se/valco79/dead-drop/-/raw/main/K8s%20Files/dead-drop-
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)

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: dead-drop-public
  labels:
    app: dead-drop
spec:
  ingressClassName: nginx-public
  rules:
    - host: namespace.kubernetes-public.it.liu.se # Replace with your namespace
      http:
        paths:
          - pathType: Prefix
            path: /
            backend:
              service:
                name: dead-drop-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 `dead-drop-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 EMACS text editor as `emacs dead-drop-ingress.yaml`

You can get the `namespace` parameter from the configuration file we created at the beginning of this lab. If your namespace was `tddc88-ht24-test-group-0`, then the `host` parameter will become

`tddc88-ht24-test-group-0.kubernetes-public.it.liu.se`

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

Let's apply our configuration!

```
$ kubectl apply -f dead-drop-ingress.yaml
```

Now, open the ingress in a browser using the `host` parameter. We now have a complete application deployed via K8s! This link should be accessible from anywhere in the world!

You must now upload the ELSA dossier to the Dead-Drop Application hosted on the Kubernetes Cluster. Secure the file using a username and password of your choice. Once the file has been uploaded, the analysts in Sweden will have all the information required to neutralize the threat.

In the covert world of espionage, victories are often silent, and heroes remain unsung. Yet, as you resume your day-to-day life in the Kremlin, you know that your actions today have set in motion a chain of events that will safeguard your nation. The shadows may be your realm, but the light of security and freedom for your people is your ultimate prize!

Well done, Agent Andersson!

Examination Task: Your intelligence on Moscow's plans for Sweden swiftly rippled across global intelligence networks. Analysts from allied nations, recognizing the severity of the threat, urgently expressed the need to access the ELSA dossier you found. Now, while SÄPO cannot share the document directly with these analysts (as doing so would create a traceable document trail, potentially exposing Sweden's involvement in foreign espionage), they can share the login credentials. However, your deployment will not be able to handle this increased demand. To address this issue, you must scale up the deployment in line with your available cluster quota or resources. Find the appropriate command to meet this increased demand write it in the box down below. Do not apply the command to the cluster before showing it to a lab assistant.

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?

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 dead-drop-ingress.yaml

$ kubectl delete -f dead-drop-service.yaml

$ kubectl delete -f dead-drop-deployment.yaml

$ kubectl get all
No resources found in tddc88-ht24-test-group-0 namespace.
```

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. Finally, we also scaled our application, which is normally done in the real world to handle peak demand. 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?