

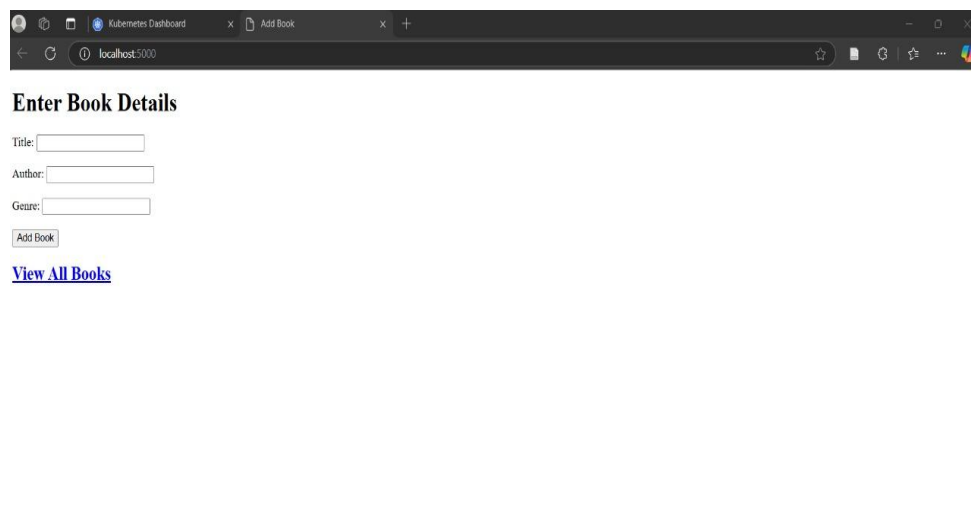
PA2577: Assignment: Build Something

Name: SumaVendrapu

Email: suve22@student.bth.se

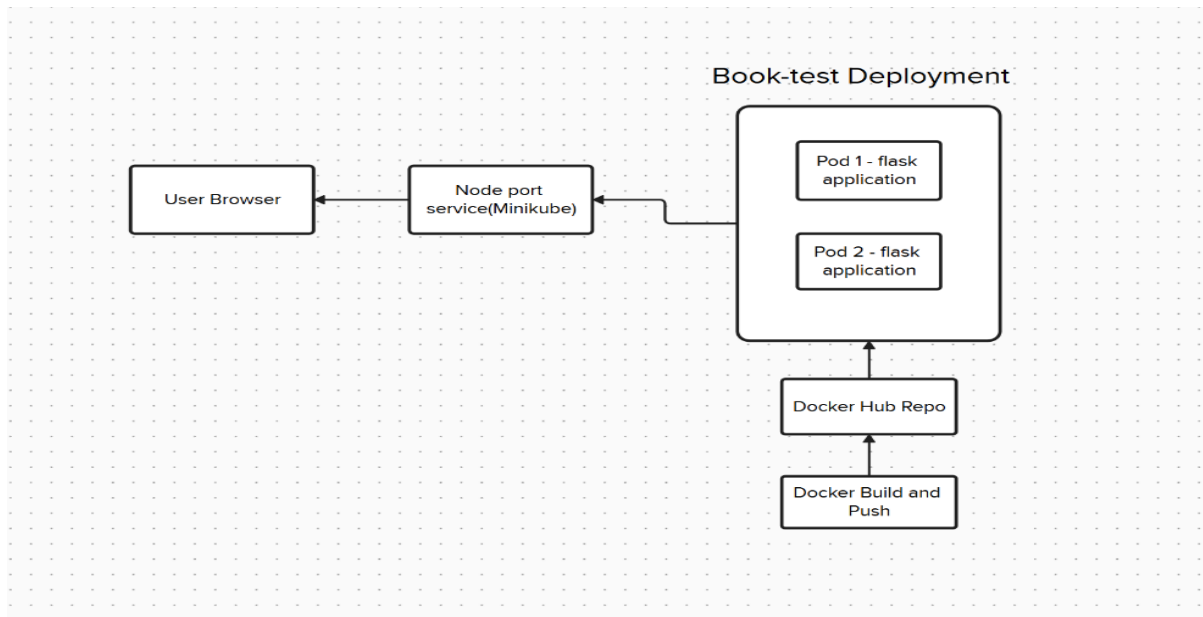
Application Overview

The **Book Service Application** is a Flask-based web app designed to manage a collection of books by allowing users to add and view details like title, author, and genre. The application is containerized using Docker and deployed on a Kubernetes cluster. Once deployed, users can access the application through their web browser. Requests are first routed to a Kubernetes Load Balancer Service, which exposes the application externally and forwards these requests to the Pods running the Flask application. Each Pod is part of a Deployment that ensures multiple replicas are running, providing fault tolerance and scalability. The Flask application in each Pod interacts with an SQLite database to store and retrieve book details,



maintaining data locally within the Pod's filesystem.

In this Kubernetes deployment, the Load Balancer Service ensures smooth communication by routing user requests to the appropriate Pods based on availability and load. The Deployment manages the Pods' lifecycle, including scaling replicas up or down to handle varying traffic levels. The application image, hosted on Docker Hub, is pulled during Deployment creation, ensuring consistent environments across all replicas. This architecture leverages Kubernetes' built-in capabilities for automatic scaling, high availability, and load balancing, ensuring a seamless user experience with a reliable and scalable backend infrastructure.



Prerequisites

1. Docker:

- To build the application image and push it to Docker Hub.
- Install Docker and ensure it is configured correctly.

2. Docker Hub Account:

- To host the Docker image for the application.
- Ensure you have an active account and are logged in to Docker CLI.

3. Minikube:

- To set up and manage a local Kubernetes cluster.
- Install Minikube and verify it is working by starting a cluster using minikube start.

4. Kubectl:

- To interact with the Kubernetes cluster and manage resources.
- Install Kubectl and configure it to work with Minikube.

5. Python and Flask Knowledge:

- Basic understanding of Python and Flask, as the application is written in Flask.

6. YAML Files for Kubernetes Configurations:

- Ensure you have the deployment.yaml and service.yaml files to define the Deployment and Service resources for Kubernetes.

7. SQLite (Optional for Local Testing):

- Required if you are testing the application locally before deployment.

Docker

The **Dockerfile** is a script used to build a container image for the Book Service Application. It automates the setup of the application environment, including installing dependencies and configuring the Flask application to run inside the container.

The Dockerfile begins by specifying a lightweight Python image as the base for the container using the line `FROM python:3.8-slim`. This establishes a minimal environment suitable for running Python applications. Next, the `WORKDIR /app` instruction sets the working directory inside the container to `/app`, ensuring that all subsequent commands are executed from this location. The `COPY . /app` command then copies all files from the current directory, where the Docker build command is executed, into the `/app` directory inside the container. To install the necessary dependencies, the `RUN pip install --no-cache-dir -r requirements.txt` command is used to install all Python packages listed in the requirements.txt file, ensuring that the environment is properly configured. The `EXPOSE 5000` directive informs Docker that the container will listen on port 5000, typically used for Flask applications. Finally, the `CMD ["python", "book_service.py"]` line specifies the command that will be run when the container starts, in this case, launching the Flask application by running the `book_service.py` script.

Docker build command is used to create an image of the application and below is the proof

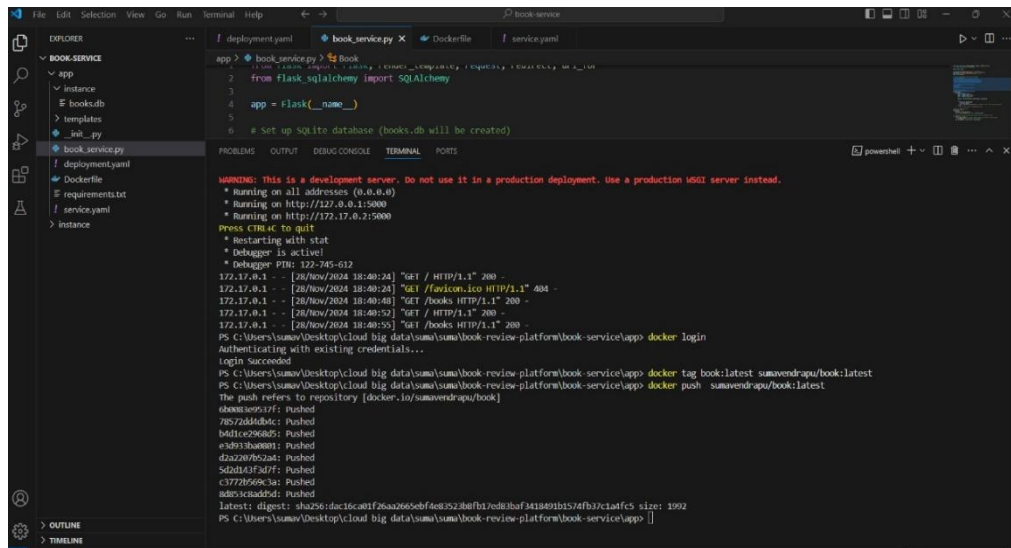
The screenshot shows a VS Code editor with the following components:

- EXPLORER:** Shows the project structure with files like `deployment.yaml`, `book_service.py`, `Dockerfile`, `requirements.txt`, and `service.yaml`.
- EDITOR:** Displays the `book_service.py` file, which includes imports for `flask` and `flask_sqlalchemy`, and a `main` function that sets up the application and database.
- TERMINAL:** Shows the output of the `docker build -t book:latest .` command. The output indicates that the build was successful, with the image named `book:latest` pushed to the Docker registry.

```

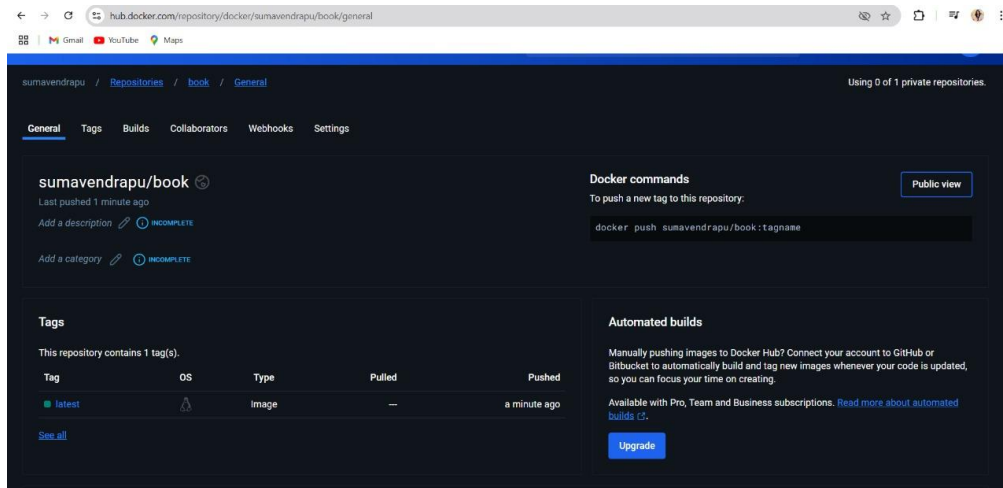
PS C:\Users\sumu\Desktop\cloud big data\sumu\book-review-platform\book-service\app> docker build -t book:latest .
[+] Building 2.6s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load metadata for docker.io/library/python:3.8-slim
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> [internal] transfer context: .
=> [1/5] FROM docker.io/library/python:3.8-slim@sha256:1052836a6602b65d611b613ab64d073288665ea79e7f6e42382f25c5d613f
=> [internal] load build context
=> [internal] transfer context: .
=> [2/5] WORKDIR /app
=> [3/5] COPY requirements.txt .
=> [4/5] RUN pip install -r requirements.txt
=> [5/5] COPY .
=> exporting layers
=> writing image sha256:b0731ba1fced2a261a6d8b9f573f28f8c79e16518897ed354ac316f562
=> naming to docker.io/library/book:latest
  
```

After the image creation the image is needed to be pushed to docker hub and below



```
book-service
File Edit Selection View Go Run Terminal Help
EXPLORER
BOOK-SERVICE
  app
  instance
  books.db
  templates
  _init_.py
  book-service.py
  deployment.yaml
  Dockerfile
  requirements.txt
  service.yaml
  instance
book-service.py
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3
4 app = Flask(__name__)
5
6 # Set up SQLite database (books.db will be created)
7 db = SQLAlchemy(app)
8
9 if __name__ == '__main__':
10     app.run(host='0.0.0.0', port=5000)
11
Terminal
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://172.17.0.2:5000
* Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 123-456-789
172.17.0.1 - - [28/Nov/2024 18:40:24] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [28/Nov/2024 18:40:24] "GET /favicon.ico HTTP/1.1" 404 -
172.17.0.1 - - [28/Nov/2024 18:40:48] "GET /books HTTP/1.1" 200 -
172.17.0.1 - - [28/Nov/2024 18:40:53] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [28/Nov/2024 18:40:55] "GET /books HTTP/1.1" 200 -
PS C:\Users\sumav\Desktop\cloud big data\sumav\book-review-platform\book-service\app> docker login
Authenticating with existing credentials...
Login Succeeded
PS C:\Users\sumav\Desktop\cloud big data\sumav\book-review-platform\book-service\app> docker tag book:latest sumavendrappu/book:latest
The push refers to repository [docker.io/sumavendrappu/book]
d8a5e9517f: pushed
7857add8db: pushed
b6d1ce968d5: pushed
e3d913ba0001: pushed
d0a2207652a2: pushed
5d0b43f3d0f: pushed
c37726969ca: pushed
ad8c5caddfd: pushed
latest: digest: sha256:d016c401f5b0d960d4f0e52d0f1a2d0b0f41a091b157d837c1aefc3 size: 1992
PS C:\Users\sumav\Desktop\cloud big data\sumav\book-review-platform\book-service\app>
```

The push images of my application is seen in the docker hub and below is the proof.

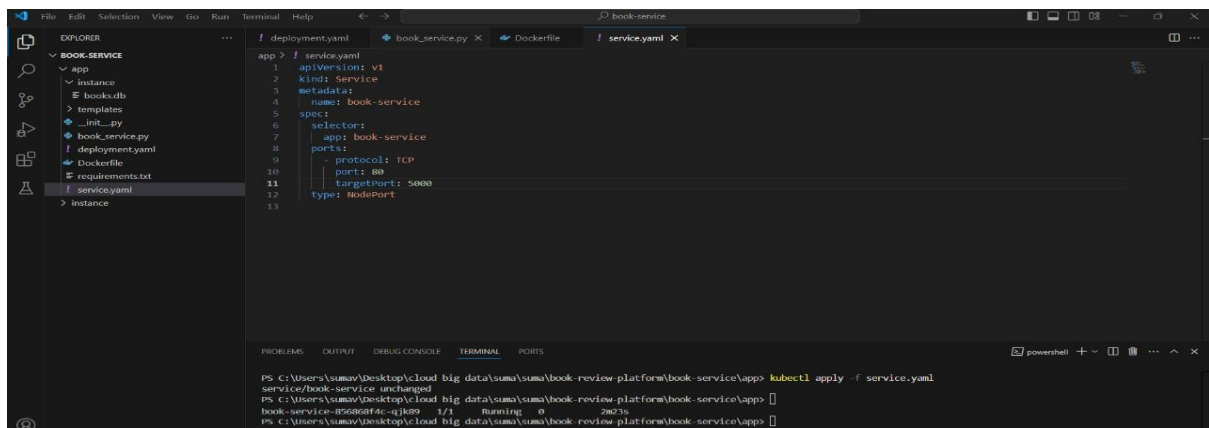


Kubernetes

First, start the Minikube service by running the necessary command. After that, check the status of the Kubernetes cluster using `kubectl` to ensure that everything is set up correctly. Once confirmed, create the `deployment.yaml` and `service.yaml` files, which contain configuration details about the service, including the number of replicas and the image names to be used. These image names should correspond to the Docker images that have been pushed to Docker Hub.

After deploying the configuration files, use the `kubectl get pods` command to verify if the pods are running correctly and whether the image has been pulled successfully

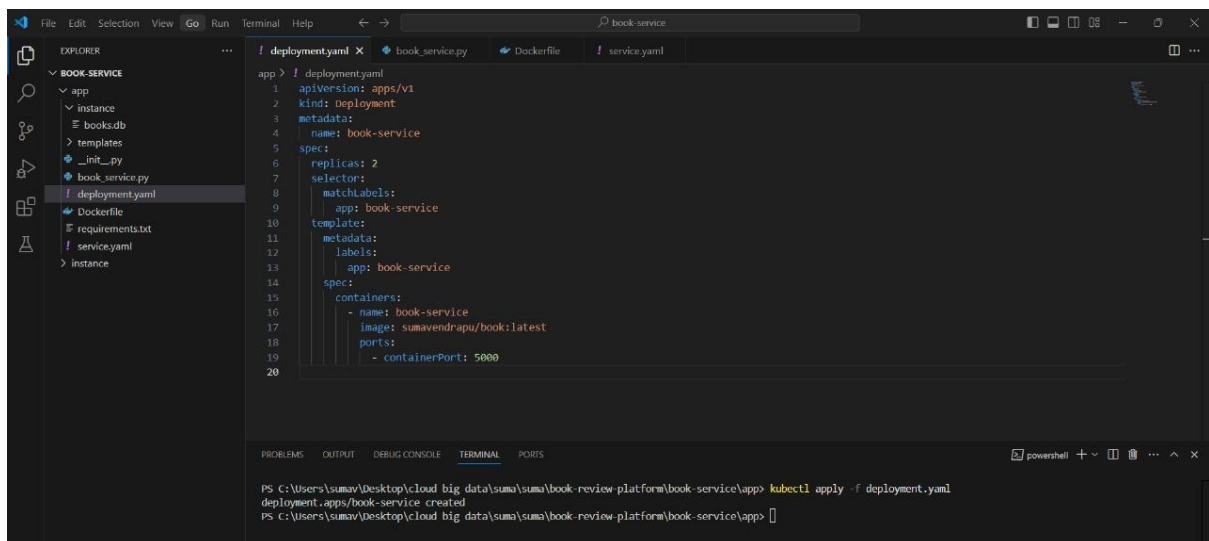
Below are the proofs of executions



The screenshot shows the Visual Studio Code editor with the Explorer sidebar on the left displaying the project structure for 'BOOK-SERVICE'. The main editor area shows the 'service.yaml' file with the following content:

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: book-service
5 spec:
6   selector:
7     app: book-service
8   ports:
9     - protocol: TCP
10       port: 80
11       targetPort: 5000
12   type: NodePort
```

The terminal at the bottom shows the command `kubectl apply -f service.yaml` being executed, resulting in the service being created.



The screenshot shows the Visual Studio Code editor with the Explorer sidebar on the left displaying the project structure for 'BOOK-SERVICE'. The main editor area shows the 'deployment.yaml' file with the following content:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: book-service
5 spec:
6   replicas: 2
7   selector:
8     matchLabels:
9       app: book-service
10   template:
11     metadata:
12       labels:
13         app: book-service
14     spec:
15       containers:
16         - name: book-service
17           image: sumavendrapu/book:latest
18           ports:
19             - containerPort: 5000
```

The terminal at the bottom shows the command `kubectl apply -f deployment.yaml` being executed, resulting in the deployment being created.

Details of Your Deployment

The application is deployed as follows:

Containerization:

The application being developed is containerized with Docker and a Docker file to build and package Flask application, and its dependencies.

Deployment on Kubernetes:

- By means of a ReplicaSet, a Kubernetes Deployment manages three replica Pods to guarantee high availability. Every Pod executes the Flask web application and is linked to an SQLite database where book information will be stored.

Service Configuration:

A LoadBalancer Service helps put the application outside the cluster for users to access while distributing traffic to the Pods.

Image Hosting:

The Docker image is maintained on Docker Hub to ensure the stable image download during Kubernetes configuration.

Access:

End users might access the application, through a public IP address that is created by the Load Balancer. Security Problems Found and/or Addressed

Securing Communication:

Next versions can utilize HTTPS for safe connection between the client and the application.

Container Security:

The base image used (python:3.8-slim) is as listed below in order to minimize the area of vulnerabilities: Necessary packages are not installed which means that unnecessary services which can be utilized by a hacker are also not installed.

Data Security:

In case Pods are deleted, which is a local database, data stored in SQLite can also be lost. This can be solved by using persistent storage or moving to a cloud database solution as a resolver for the issue.

Image Integrity:

Images are based on Docker Hub where version control allows only relevant images to be used for a given application.

Pod Security:

Kubernetes Pod Security Policies in the production environment can be used to limit capability and also set isolation between the Pods.

These problems allow the application to be protected against possible threats, and the data as well as the operations of the user to remain secure.

Repository Link:

The repository for the project has been updated and is now accessible at the following link:

<https://github.com/Sumavendrapu/buildsomething.git>