Building Scalable Applications using Docker and Kubernetes

ΓΑΛΑΝΗΣ ΑΧΙΛΛΕΑΣ ΑΛΕΞΑΝΔΡΟΣ ΒΑΣΙΛΕΙΟΣ - 02941 and ΓΑΛΑΝΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΟΡΕΣΤΗΣ ΒΑΣΙΛΕΙΟΣ - 03074

Application
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας, Βόλος
{acgalanis, kogalanis}@e-ce.uth.gr

Περιεχόμενα

1 Application Overview

”SFM” is a web-based application developed using Node.js. It serves as a file manager, offering a user-friendly interface for file operations. The back-end, built on Express, manages file handling functionalities, like creating, deleting, or moving an object and managing files, while the front-end provides an interactive user interface. We used python 3.8.0. To run the program install all dependencies using:

```
npm install
```

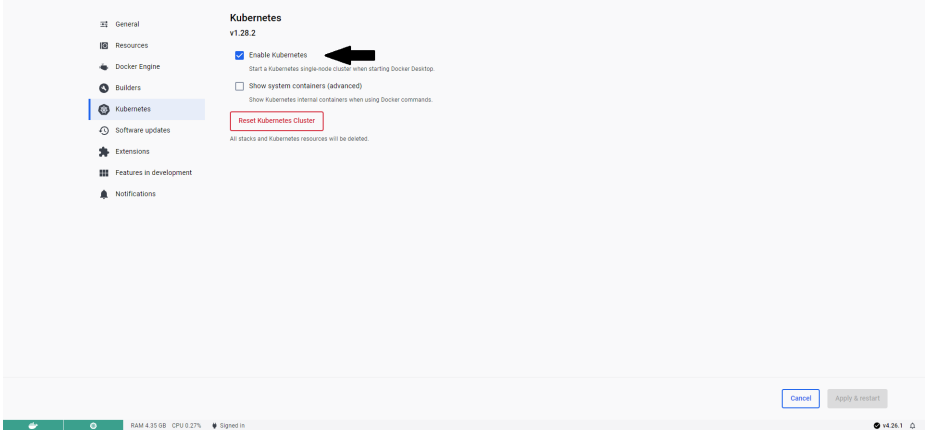
and execute it using:

```
npm start
```

2 Prerequisites

We followed this tutorial to install Docker and Kubernetes in our local machine. We used the Minikube local cluster. Make sure you have Docker Desktop running and

enable Kubernetes in the settings.



3 Containerization

We decided to split our application in two containers — one dedicated to the front-end and the other to the back-end. This approach enhances maintainability and scalability. To implement this, we began by constructing the Dockerfiles for each segment:

– **Front-end Dockerfile:**

```
# We use an official Nginx image as a base
FROM nginx:alpine

# Copy static files to Nginx server
COPY src/frontend/view /usr/share/nginx/html
COPY src/frontend/main.html /usr/share/nginx/html/index.html

# Expose port 80
EXPOSE 80

# Start Nginx when the container has provisioned
CMD ["nginx", "-g", "daemon off;"]
```

```
# Use an official Nginx image as a base
FROM nginx:alpine
```

This line specifies the base image for the Docker container. It uses an official Nginx image based on Alpine Linux, which is a lightweight and secure Linux distribution. The ‘nginx:alpine‘ image comes preconfigured with Nginx.

```
# Copy static files to Nginx server
COPY src/frontend/view /usr/share/nginx/html
```

This command copies static files from the local directory ‘src/frontend/view‘ into the container at the directory ‘/usr/share/nginx/html‘. This directory is the default location where Nginx serves static files.

```
COPY src/frontend/main.html /usr/share/nginx/html/index.html
```

This line copies the file ‘main.html‘ from the local ‘src/frontend‘ directory and places it as ‘index.html‘ in the ‘/usr/share/nginx/html‘ directory inside the container. This makes ‘main.html‘ the default page served by Nginx.

```
# Expose port 80
EXPOSE 80
```

This line indicates that the container listens on port 80. While this does not actually publish the port on the host machine, it serves as documentation and can be used by tools that run Docker containers to automatically expose the port.

```
# Start Nginx when the container has provisioned
CMD ["nginx", "-g", "daemon off;"]
```

The final line defines the default command that gets executed when the container starts. Here, it starts the Nginx server with the specified flag ‘-g ”daemon off;”‘. This flag tells Nginx to run in the foreground (not as a daemon), which is a common practice for Docker containers to keep the container running. Containers are designed to stop when their main process (in this case, Nginx) finishes running, so keeping Nginx in the foreground ensures that the container doesn’t immediately shut down.

– **Back-end Dockerfile:**

```
# Use an official Node.js image as a base
FROM node:14

# Create app directory inside the container
WORKDIR /app

# Install app dependencies
COPY package*.json ./

# Bundle app source inside /app
COPY src/backend/ src/backend/

RUN npm install

EXPOSE 3000
```

```
# Run the app when the container launches
CMD ["npm", "start"]
```

```
# Use an official Node.js image as a base
FROM node:14
```

This line sets the base image for the Docker container. It uses an official Node.js image, specifically version 14. The Node.js image contains the Node.js runtime and npm, which are necessary to run a Node.js application.

```
# Create app directory inside the container
WORKDIR /app
```

This command sets the working directory in the container to '/app'. All subsequent commands will be executed in this directory. It's like doing 'cd /app' in a terminal.

```
# Install app dependencies
COPY package*.json ./
```

This line copies 'package.json' and 'package-lock.json' from the project into the root of the working directory ('/app') in the container. These files define the project's dependencies.

```
# Bundle app source inside /app
COPY src/backend/ src/backend/
```

This command copies the source code of the back-end application from the local 'src/backend/' directory into the 'src/backend/' directory inside the container.

```
RUN npm install
```

This line executes the command 'npm install' inside the container, which installs all the dependencies specified in 'package.json'.

```
EXPOSE 3000
```

This instruction informs Docker that the container listens on port 3000. This is the port that the Node.js application will use. Note that this doesn't automatically expose the port to the host machine; it's more for documentation and must be published explicitly when running the container.

```
CMD ["npm", "start"]
```

The final line in the Dockerfile specifies the command to run when the container starts. In this case, it runs 'npm start', which starts the Node.js application.

After constructing our Dockerfiles and ensuring we are located in the root directory of our application, we proceed by executing the following commands:

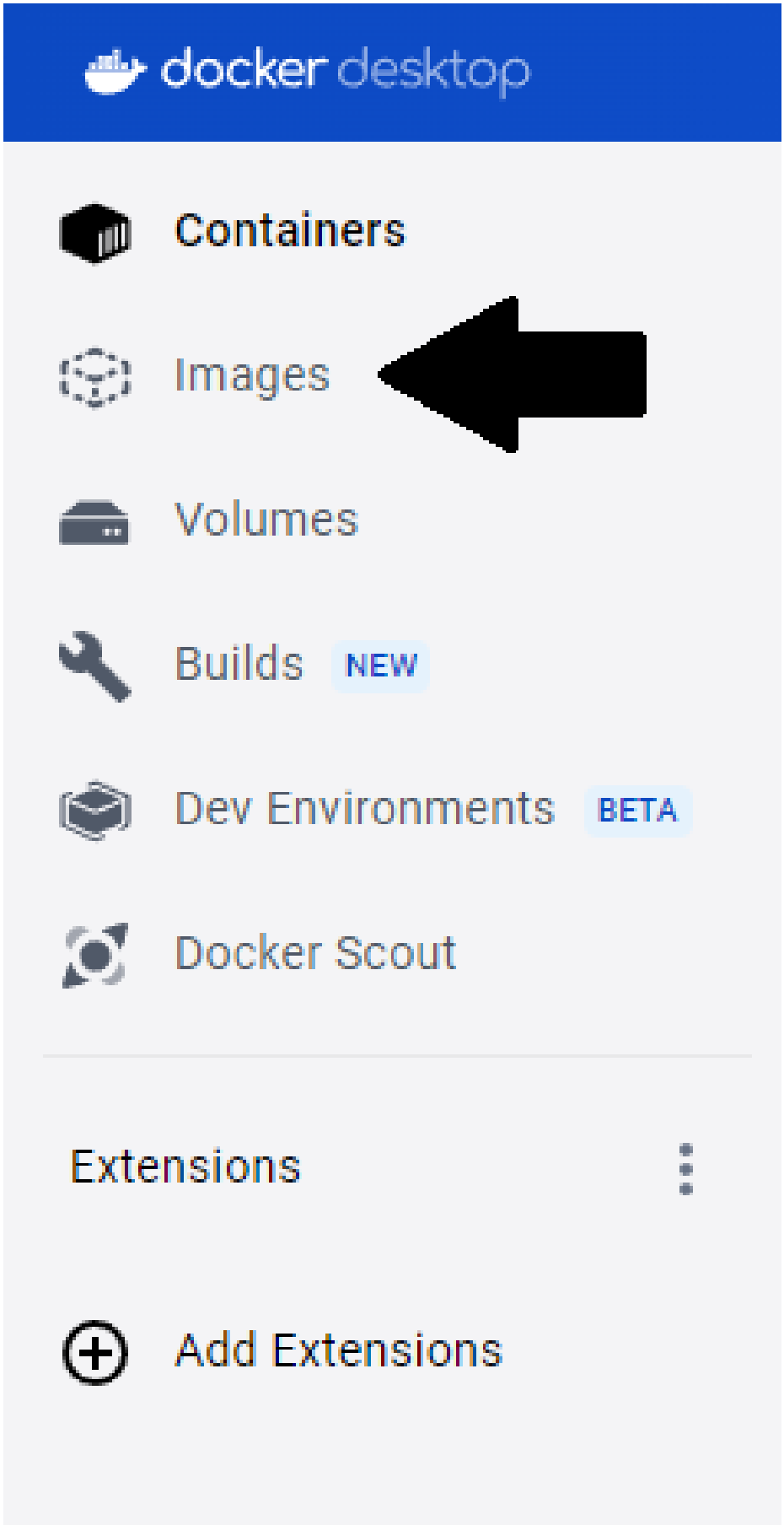
```
docker build -f src/backend/Dockerfile -t <username>/myapp-
    backend:1.0 .
docker build -f src/frontend/Dockerfile -t <username>/myapp-
    frontend:1.0 .
```

These two commands are Docker commands and are used to build Docker images from the Dockerfiles for the back-end and front-end component of our application. The **-f** flag specifies the path to the Dockerfile and the **-t** flag tags the resulting image. This tag consists of a username/repository (the username used in Docker Hub) and a version (1.0).

We can verify that the images are created by typing the command:

```
docker images
```

or navigating in the images menu in Docker Desktop:


docker desktop
Containers
Images
Volumes
Builds
NEW
Dev Environments
BETA
Docker Scout
Extensions
Add Extensions

Now we can login to Docker hub and push the images in a repository. This step is mandatory to pull the images in the Kubernetes cluster in case it can't find them. We login to Docker Hub using the command:

```
docker login --username <username> --password <password>
```

and we push the images into repositories by executing the commands:

```
docker push <username>/myapp-frontend:1.0
docker push <username>/myapp-backend:1.0
```

The last step is to make sure that our images run correctly. We run the back-end image:

```
docker run -p 3000:3000 <username>/myapp-backend:1.0
```

and we open **http://localhost:3000/** as we have exposed port 3000 for the back-end. We observe that the back-end image can't find the html files rendering the user interface which is correct because there is no communication between the front and back end yet.

```
Error: ENOENT: no such file or directory, stat '/app/src/frontend/main.html'
```

Additionally, we run the front-end image:

```
docker run -p 8080:80 <username>/myapp-frontend:1.0
```

and we open **http://localhost:8080/** as we have exposed port 80 for the front-end. We deduct that despite the user interface is rendered correctly, there are no utilities because there is no communication between the front and back end yet and the and the API routes are not yet configured or operational.

After executing the commands, two containers are created for the images within Docker Desktop, allowing us to manage their functionality both from there and the command line.

4 Deployment

We now want to enorchestrate the functionality of the containers using Kubernetes. We first the local launch cluster with:

```
minikube start
```

Then, we create two YAML configuration files one for the back-end and one for the front-end. These files typically define the configuration for containers, including which images to use, necessary environment variables, network settings, and volume configurations. They play a crucial role in managing how different components of an application, such as the front-end and back-end services, interact and connect with each other. In Kubernetes, they are used to set up various resources like pods, services, deployments, and ingress rules, facilitating the deployment process.

– **Front-end configuration file:**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: <username>/myapp-frontend:1.0
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  type: NodePort
  ports:
    - protocol: TCP
      port: 80
      nodePort: 30007
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: <username>/myapp-frontend:1.0
        ports:
        - containerPort: 80
```

This section defines a Kubernetes deployment for the front-end. It specifies that the deployment should have 2 replicas, meaning two instances of the front-end pod will be running. It uses the image ‘<username>/myapp-frontend:1.0‘ and exposes container port 80, the standard port for web servers. If it can’t find the image in the

local machine it pulls it from Docker Hub.

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  type: NodePort
  ports:
    - protocol: TCP
      port: 80
      nodePort: 30007
```

This section creates a Kubernetes service for the front-end deployment. It's a NodePort service, which makes the front-end accessible from outside the Kubernetes cluster. The service listens on port 80 and is available on a node's IP at port 30007.

– **Back-end configuration file:**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: <username>/myapp-backend:1.0
        ports:
        - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  type: ClusterIP
```

```
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: <username>/myapp-backend:1.0
        ports:
        - containerPort: 3000
```

This section outlines a Kubernetes deployment for the back-end. It specifies that the deployment should have 2 replicas, which means two instances of the back-end pod will be running concurrently. The deployment uses the image '<username>/myapp-backend:1.0' and opens container port 3000, typically used for back-end services.If it can't find the image in the local machine it pulls it from Docker Hub.

```
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
```

This section creates a Kubernetes service for the back-end deployment. It is defined as a ClusterIP service, which means it is only accessible within the Kubernetes cluster. The service listens on port 3000 and targets port 3000 on the back-end pods,

facilitating internal communication within the cluster.

After constructing our configuration files and ensuring we are located in the root directory of our application, we proceed by executing the following commands:

```
kubectl apply -f src/k8s/back-end.yml
kubectl apply -f src/k8s/front-end.yml
```

These commands are used in Kubernetes for deploying or updating resources in a cluster. kubectl is the command-line tool for interacting with a Kubernetes cluster. It allows you to run commands against the cluster for deploying applications, inspecting and managing cluster resources, and viewing logs.The **-f** flag specifies the path to the configuration file.

After we configure the pods containing the front and back end containers we want to ensure that they run correctly. We execute the following commands:

- **kubectl get services:** This command lists all the services in the current namespace. Services in Kubernetes are an abstraction that defines a logical set of pods and a policy by which to access them, often termed a micro-service. The output includes details like the name of the service, type (e.g., ClusterIP, NodePort, LoadBalancer), IP address, port(s), and age.

- **kubectl get deployments:** This command lists all the deployments in the current namespace. Deployments are higher-level abstractions that manage the deployment and scaling of a set of pods, along with updating the pods as specified. The output usually includes the name of the deployment, the number of desired replicas, the number of up-to-date replicas, the number of available replicas, and the age of the deployment.
- **kubectl get pods:** This command lists all the pods in the current namespace Pods are the smallest deployable units in Kubernetes and consist of one or more containers. The output includes information like the name of the pods, their status (running, pending, failed), how long they have been running, and on which node they are located.

- **kubectl get nodes:** This command lists all the nodes in the cluster. A node is a worker machine in Kubernetes, which can be either a virtual or a physical machine, depending on the cluster. The output shows the name of each node, its status (e.g., Ready, NotReady), how long it has been up, its version, and internal IP address.

Finally, we execute this command:

```
minikube service backend-service
```

to access the backend-service that's running in Minikube Kubernetes cluster. Essentialy, it launches the application using the front-end and back-end pods.

5 Sources

- Docker Documentation
- Kubernertes Documentation