Milestone 2 - Willem De Bie - Linux Webservices

Table of Contents

1. Introduction

In this milestone I explored the power of Kubernetes, software that helps manage containers. I wanted to see how Kubernetes works by creating a trio of containers that work together like a dream: a front-end, an API and a database. The whole point was to make these containrs talk to each other smoothly. The front-end is an instance of lighttpd, which is a simple webserver with which I loved working. The API consists of Node.JS and the database was MongoDB, which is a "NoSQL" database, meaning it has flexible schemas to build modern applications.

I'll spill the beans about how I did all this - setting up the containers, tweaking configurations, and making them play nice within a Kubernetes cluster. And believe me, it wasn't all smooth sailing! I ran into some roadblocks, but I found ways around them.

This paper will explain everything I did to achieve this milestone, from how I containerized everything to how I managed to make them work together within a Kubernetes cluster. Plus I'll talk about why Kubernetes is such a big deal nowadayas and how it makes apps stronger, scalable, and easier to handle.

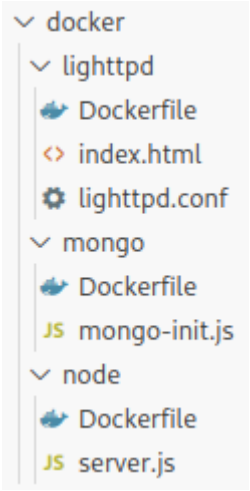
I apologize for the cut-offs on pages sometimes, I'm trying to fit a lot of codeblocks on this paper.

2. Making the container images

I realized that there were two ways to handle this.

- I could take an image for lighttpd, node and mongo from the Docker Hub and place them in Kubernetes configuration files and work with volumes, configmaps, etc. all in the Deployment to make them run the code that I wanted.
- Or I could create a "pre-built" image for each container **before** working with them in Kubernetes, so that I don't need to add volumes or configmaps because everything is already inside that image.

I chose the latter option because I wanted my Kubernetes configuration to be clean and easy to read, as well as easy to maintain and change things. So I went ahead and made three docker containers and I pushed them to Docker Hub. This is what my file structure looks like to build my containers:



I would go into the appropriate directory for every container, and then run the following commands to build them, and then push them to Docker Hub so they could later be pulled again by my Kubernetes cluster.

```
docker build -t willemdebie/lighttpd .
docker push willemdebie/lighttpd

docker build -t willemdebie/node .
docker push willemdebie/node

docker build -t willemdebie/mongo .
docker push willemdebie/mongo
```

willemdebie / **node**
Contains: Image | Last pushed: 2 days ago

willemdebie / **lighttpd**
Contains: Image | Last pushed: 2 days ago

willemdebie / **mongo**
Contains: Image | Last pushed: 2 days ago

2.1. My Lighttpd container

So for the first container, the front-end with Lighttpd, I created a Dockerfile with Ubuntu on which I installed Lighttpd and loaded my HTML page and configuration.

```
# Use Ubuntu 22.04 as the base image for the Lighttpd container
FROM ubuntu:22.04

# Set ENVVAR to avoid Ubuntu asking for user interaction
ARG DEBIAN_FRONTEND=noninteractive

# Install Lighttpd
RUN apt-get update && apt-get install -y lighttpd

# Copy the code for the HTML page to the location where Lighttpd can run it
COPY ./index.html /var/www/html/.

# Copy the configuration for lighttpd
COPY ./lighttpd.conf /etc/lighttpd/

# Expose port 80 for HTTP traffic
EXPOSE 80

# Start Lighttpd when the container starts
CMD ["lighttpd", "-D", "-f", "/etc/lighttpd/lighttpd.conf"]
```

The HTML code and lighttpd webserver configuration are located in `/var/www/html` and `/etc/lighttpd` respectively because these are the default locations according to the Lighttpd docs. The `index.html` file looks pretty familiar because the front-end code was provided to us, but I changed the endpoint and added a little bit of JS code to display the container ID of the API pod.

The `<script>` section in the HTML file has two "fetch" instructions, once from `/user` and another time from `/id`. This doesn't mean anything in itself, but I configured my lighttpd to have two proxies. What I'm doing here is I basically forward a request to my API container, which will resolve it and then reply in JSON format. `node-service` is the name of the service for the API, k8s resolves it so that it connects to that pod:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Milestone 2</title>
  </head>
  <body>
    <h1><span id="user">Loading...</span> has reached milestone 2!</h1>
    <hr>
    <h2>Container ID: <span id="containerid">loading...</span></h2>

    <script>
      // fetch user from API
      fetch("/user")
        .then((res) => res.json())
        .then((data) => {
          // get user name
          const user = data.name;
          // display user name
          document.getElementById("user").innerText = user;
        });

      fetch("/id")
        .then((res) => res.json())
        .then((data) => {
          // get container ID
          const containerid = data.id;
          // display container ID
          document.getElementById("containerid").innerText = containerid;
      });
    </script>
  </body>
</html>
```

```
# Load the appropriate modules for this configuration.
# mod_auth: Provides authentication capabilities for the server.
# mod_fastcgi: Implements FastCGI support, allowing external applications to
interface with the server for dynamic content generation.
# mod_proxy: Enables proxying requests to other servers, useful for load
balancing or forwarding requests to backend services.
server.modules += ("mod_auth", "mod_fastcgi", "mod_proxy")

# Proxy configuration for requests to URLs starting with '/user'
$HTTP["url"] =~ "^/user($|/)" {
    proxy.server = (
        "/user" => (
            (
                "host" => "node-service",  # Hostname of the backend service
                "port" => 3000             # Port number where the backend
service is running
            )
        )
    )
}

# Proxy configuration for requests to URLs starting with '/id'
$HTTP["url"] =~ "^/id($|/)" {
    proxy.server = (
        "/id" => (
            (
                "host" => "node-service",  # Hostname of the backend service
                "port" => 3000             # Port number where the backend
service is running
            )
        )
    )
}

# Set the document root for the server
server.document-root = "/var/www/html"

# Define the default index file names to be served
```

```
index-file.names = ("index.html")

# Enable directory listing when index file is not found
dir-listing.activate = "enable"

# Set the server port to listen on
server.port = 80

# Define MIME types for certain file extensions
mimetype.assign = (
    ".html" => "text/html",  # Assigns the MIME type 'text/html' to files with
a '.html' extension
)
```

I explained every line in a short comment above them, maybe I should add some clarification for the "MIME types." These types are crucial for web servers to understand what to do with certain file extentions, for example this HTML file should be read and displayed as a webpage, not just plain text or for example a download.

2.2. My NodeJS container

So the front-end is ready, I have built and pushed it to the Docker Hub. Now the next step is to make the API, which will serve as a middleman between the front-end and the database in the back-end. I chose to use NodeJS for this as I'm experience with it and I like to work with JavaScript to make APIs. The Dockerfile for the API is rather simple:

```
# Use Node v18.15.0 as the base image for this container
FROM node:18.15.0

# Set /home/node/app as the working directory for server scripts as per the
Docker Hub wiki about Node
WORKDIR /home/node/app

# Install express, cors and mongodb. These packages are required to run the
server.
RUN npm install express cors mongodb

# Copy the server.js file to the appropriate location
COPY ./server.js /home/node/app/.

# Open the default Node port (3000)
EXPOSE 3000

# Start the node server
CMD ["node", "server.js"]
```

I start from a Node image that I know for sure will work, it is `18.15.0` specifically because I know "MongoDB Compass" uses this version to connect to a mongo database and that's what I am about to do as well. One of the packages that I'm installing is express, which is a web framework that will handle the `fetch` requests from `index.html`. The cors package is CORS middleware, which I installed because my browser was giving some issues with CORS (Cross-Origin Resource Sharing) when I tried to access the API. And finally the mongodb package to connect to the database later.

The code in `server.js` is pretty long, I tried my best to explain everything in the comments:

```
// import all NPM packages that I've installed
// as well as "os" to get the container ID later
const express = require('express');
const cors = require('cors');
const { MongoClient } = require('mongodb');
const os = require('os');

// define "app" for the express framework
const app = express();
const PORT = 3000;

// Enable CORS for all routes
app.use(cors());

// This is the URL it will use to connect to my mongo database
// "mongo-service" will be the name of the Kubernetes service for my mongo
deployment
// The default port for mongo is 27017, so I used that.
const mongoURI = `mongodb://mongo-service:27017`;
let conn;

// Define a route for /user endpoint
app.get('/user', async (req, res) => {
  try {
    // Making a connection to the database
    const client = new MongoClient(mongoURI);
    conn = await client.connect();

    // Accessing the database and collection
    const db = conn.db('milestone2');
    const collection = db.collection('names');

    // Find the document with the name
    const result = await collection.findOne({});
    const name = result ? result.name : 'Name not found';

    // Respond with JSON containing the retrieved name
    res.json({ name });
```

```
    // Close the connection
    await client.close();
  } catch (error) {
    console.error('Error:', error);
    res.status(500).json({ error: "Internal server error" });
  }
});

// Define a route for /id endpoint
app.get('/id', async (req, res) => {
  try {

    // get the hostname of the container
    const id = os.hostname();

    // Respond with JSON containing the retrieved ID
    res.json({ id });

  } catch (error) {
    console.error('Error:', error);
    res.status(500).json({ error: "Internal server error" });
  }
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

2.3. My MongoDB container

The only image left to make now is the one for MongoDB, I didn't have to change much from the base mongo image here, except to add a database and collection that stores my name. Here is the Dockerfile:

```
# Start with mongo as the base image for this container
FROM mongo

# Copy the database initialization file to the appropriate location according
to Docker Hub
# For MongoDB, this is written in JavaScript
COPY ./mongo-init.js /docker-entrypoint-initdb.d/

# Open the default mongodb port
EXPOSE 27017

# Start the mongo daemon and make it accessible for all IPs
CMD ["mongod", "--bind_ip_all"]
```

The "initialization file" basically creates a database called "milestone2", a collection called "names" and it inserts my name into that collection:

```
db = db.getSiblingDB('milestone2');
db.createCollection('names');
db.names.insertOne({ name: 'Willem De Bie' });
```

3. Kubernetes

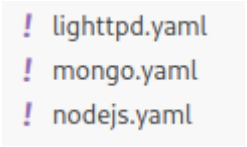
Now that I have the images for all three containers stored in Docker Hub, I have to set up a Kubernetes configuration that runs these containers on a worker node and allows them to connect to one another. The first thing I did was installing kind and kubectl on my Linux laptop and setting up a configuration file for the cluster. I have to admit that this was really easy because the course material provided a cluser configuration for two worker nodes, so I copied it and removed a single worker because I only need one.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
    kubeadmConfigPatches:
      - |
        kind: InitConfiguration
        nodeRegistration:
          kubeletExtraArgs:
            node-labels: "ingress-ready=true"
    extraPortMappings:
      - containerPort: 80
        hostPort: 80
        protocol: TCP
      - containerPort: 443
        hostPort: 443
        protocol: TCP
  - role: worker
```

Then I ran the command to start this cluster in Docker:

```
kind create cluster --config=kindconfig.yaml
```

When this finishes I can now use `kubectl` to communicate with this cluster and apply configurations. The configurations that I made for k8s can be found in three files. One file for each pod in the worker node, the file structure for these looks as follows:



Each of these YAML files includes a Deployment section and a Service section. the Deployment is to create the pod and configure it and the services exist to open it up for connection to my host machine and/or to other pods inside of the node. Just like I did with my Docker containers, I'll showcase every configuration in order from front-end to back-end.

3.1. K8s Lighttpd, NodeJS and MongoDB

The three pages that follow contain the kubernetes configurations for all three deployments and services, first lighttpd, then nodejs and finally mongodb.

Before I wrote these YAML files, I thought about what I needed for each pod.

- I have to open the Lighttpd pod up for connection from my host machine, so that I can open it from a browser so I need a "NodePort". This NodePort builds on top of the ClusterIP and exposes the pod (or group of pods) to the outside world.
- The Node pod does not need a NodePort because it only needs to be accessed from another pod in the cluster.
- The MongoDB doesn't **need** a NodePort because the API can access the node without one, but I made one anyway so I can connect to the database from my host machine. This way I can change the collection values for demonstration purposes.

I explained the function of all lines in this YAML files with comments.

```
# Deployment configuration for the lighttpd application
apiVersion: apps/v1
kind: Deployment
metadata:
  name: lighttpd-deployment  # Name of the deployment
spec:
  replicas: 1  # Number of desired replicas for the deployment
  selector:
    matchLabels:
      app: lighttpd  # Label selector for pods associated with this deployment
  template:
    metadata:
      labels:
        app: lighttpd  # Label assigned to pods created from this template
    spec:
      containers:
        - name: lighttpd  # Name of the container
          image: willemdebie/lighttpd  # Docker image used for this container
          ports:
            - containerPort: 80  # Port number that the container exposes

---
# Service configuration for the lighttpd application
apiVersion: v1
kind: Service
metadata:
  name: lighttpd-service  # Name of the service
spec:
  type: NodePort  # Type of service (exposes the Service on a port on each
node)
  selector:
    app: lighttpd  # Label selector to route traffic to pods with this label
  ports:
  - port: 80  # Port on which the service is exposed within the cluster
    nodePort: 30100  # Port accessible externally on each node for the service
```

```
# Deployment configuration for the Node.js application
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-deployment  # Name of the deployment
spec:
  replicas: 1  # Number of desired replicas for the deployment
  selector:
    matchLabels:
      app: node  # Label selector for pods associated with this deployment
  template:
    metadata:
      labels:
        app: node  # Label assigned to pods created from this template
    spec:
      containers:
        - name: node  # Name of the container
          image: willemdebie/node  # Docker image used for this container
          ports:
            - containerPort: 3000  # Port number that the container exposes

---
# Service configuration for the Node.js application
apiVersion: v1
kind: Service
metadata:
  name: node-service  # Name of the service
spec:
  selector:
    app: node  # Label selector to route traffic to pods with this label
  ports:
  - protocol: TCP  # Protocol used for the service
    port: 3000  # Port on which the service is exposed within the cluster
    targetPort: 3000  # Port to which the service sends traffic inside the pods
```

```
# Deployment configuration for the MongoDB instance
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-deployment  # Name of the deployment
  labels:
    app: mongo  # Label assigned to the deployment
spec:
  replicas: 1  # Number of desired replicas for the deployment
  selector:
    matchLabels:
      app: mongo  # Label selector for pods associated with this deployment
  template:
    metadata:
      labels:
        app: mongo  # Label assigned to pods created from this template
    spec:
      containers:
      - name: mongodb  # Name of the container
        image: willemdebie/mongo  # Docker image used for this container
        ports:
        - containerPort: 27017  # Port number that the container exposes

---
# Service configuration for the MongoDB instance
apiVersion: v1
kind: Service
metadata:
  name: mongo-service  # Name of the service
spec:
  type: NodePort  # Type of service (exposes the Service on a port on each
node)
  selector:
    app: mongo  # Label selector to route traffic to pods with this label
  ports:
  - port: 27017  # Port on which the service is exposed within the cluster
    targetPort: 27017  # Port to which the service sends traffic inside the
```

```
pods
    nodePort: 30200  # Port accessible externally on each node for the service
```

3.2. Running the final result

Once I made all these YAML files, I can apply the configuration for each deployment to create a ReplicaSet of pods, in this case they all have one replica so I just make three pods. Before I apply the front-end, I have to make sure that the API and database are running so that it can successfully connect to the backend. This is done with the following commands:

```
kubectl apply -f nodejs.yaml
kubectl apply -f mongo.yaml
```

The `-f` flag here is to specify a filename, in this case `nodejs.yaml` and `mongo.yaml`.

When those two are up and running, I can apply the front-end as well.

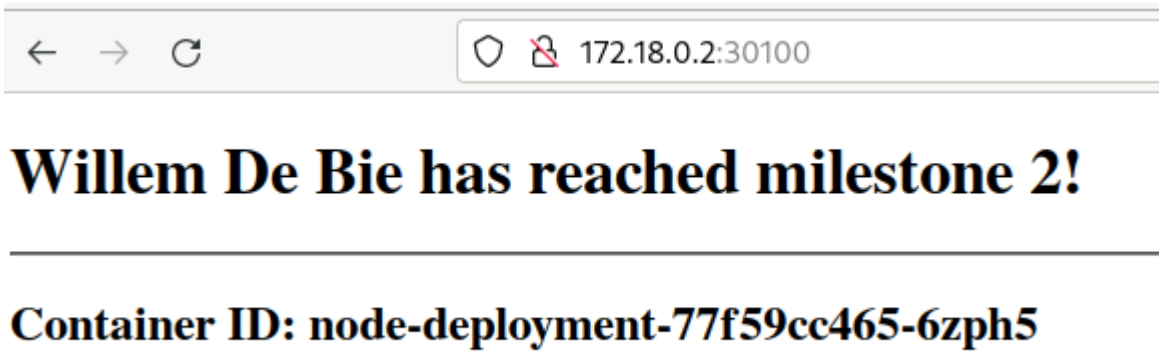
```
kubectl apply -f lighttpd.yaml
```

And finally I can find the IP to access the website by seeing the worker node IP.

```
kubectl get node -o wide
```

```
archlinux% kubectl get node -o wide
NAME                 STATUS   ROLES           AGE    VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE                         KERNEL-VERSION   CONTAINER-RUNTIME
kind-control-plane   Ready    control-plane   2d17h  v1.27.3   172.18.0.3    <none>        Debian GNU/Linux 11 (bullseye)   6.6.4-arch1-1    containerd://1.7.1
kind-worker          Ready    <none>          2d17h  v1.27.3   172.18.0.2    <none>        Debian GNU/Linux 11 (bullseye)   6.6.4-arch1-1    containerd://1.7.1
```

When I use that IP and the port I assigned for the NodePort (30100), the final result will look like this:



Additionally I can also see all my pods, deployments, services, etc. with a simple command to see what's happening in the background:

```
kubectl get all
```

```
archlinux% kubectl get all
NAME                                      READY   STATUS    RESTARTS       AGE
pod/lighttpd-deployment-6864bc9578-9kpdr  1/1     Running   3 (105m ago)   2d1h
pod/mongo-deployment-6864666cc5-jnghp     1/1     Running   1 (105m ago)   2d1h
pod/node-deployment-77f59cc465-6zph5      1/1     Running   1 (105m ago)   2d1h

NAME                       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/kubernetes         ClusterIP   10.96.0.1       <none>        443/TCP        2d1h
service/lighttpd-service   NodePort    10.96.23.16     <none>        80:30100/TCP   2d1h
service/mongo-service      ClusterIP   10.96.249.108   <none>        27017/TCP      2d1h
service/node-service       ClusterIP   10.96.195.234   <none>        3000/TCP       2d1h

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/lighttpd-deployment   1/1     1            1           2d1h
deployment.apps/mongo-deployment      1/1     1            1           2d1h
deployment.apps/node-deployment       1/1     1            1           2d1h

NAME                                             DESIRED   CURRENT   READY   AGE
replicaset.apps/lighttpd-deployment-6864bc9578   1         1         1       2d1h
replicaset.apps/mongo-deployment-6864666cc5      1         1         1       2d1h
replicaset.apps/node-deployment-77f59cc465       1         1         1       2d1h
```

4. Conclusion

The experience of setting up a Kubernetes cluster for our milestone—integrating a front-end, an API, and a backend database—has been really fun. This milestone taught me a lot, especially compared to my past attempts at working with containerized systems in milestone 1.

This time around, things went smoother. Kubernetes made the whole process more manageable and efficient. It helped me understand how to better organize and handle complex stacks, making deployment a lot simpler than before.

Building each part of the system within Kubernetes was a great learning experience. Seeing how the front-end, API, and database interacted seamlessly within containers was nice. It showed me the incredible power of containerization in making development and deployment more efficient.

This project reinforced the idea that learning is a process, especially in the fast-changing world of technology. I'm excited to use this new knowledge and efficiency for future projects. I'm feeling more confident in dealing with complex containerized systems.

In summary, this project was not just about meeting our deadline—it was about gaining valuable skills and confidence in handling containerized setups. I've learned a lot, and I'm looking forward to applying this knowledge in the future.