ACIT 3495

Technical report:

Containerized

Video Streaming System

By Garen Pham

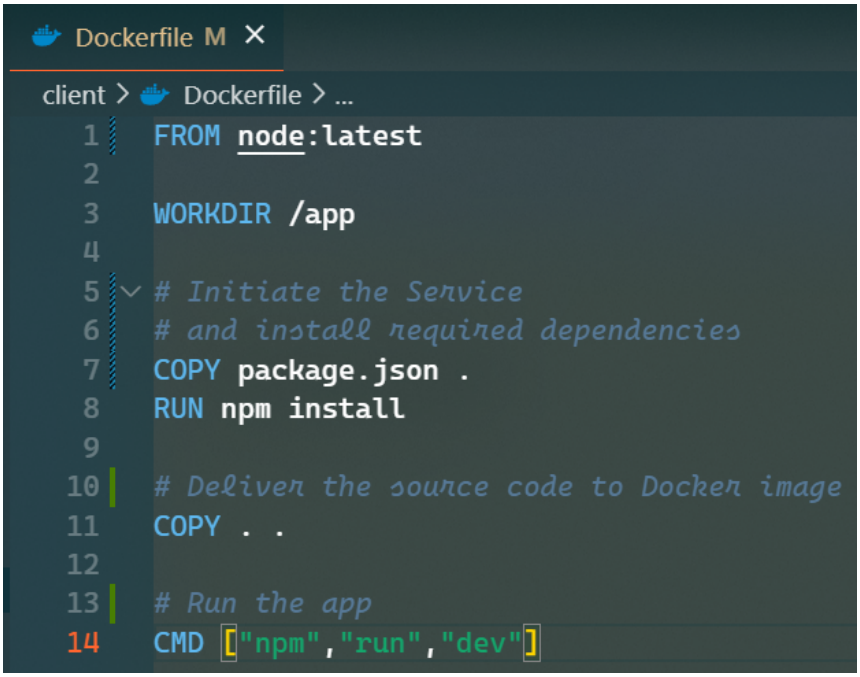
Submitted to Motasem Aldiab

Introduction:

The Video Streaming System that I constructed for project 1 in ACIT 3495 is a web application, using Next.js in the Video Streaming and Upload Video (frontend), combined with MySQL DB and lastly Node.js for the Authentication and File System services. They are deployed as docker containers by using docker-compose. The services communicate with each other via REST API. The front end uses Axios to communicate with the authentication and file system services. The backend servers are using Express Node.js to operate the servers to listen and process the request APIs from the frontend app.

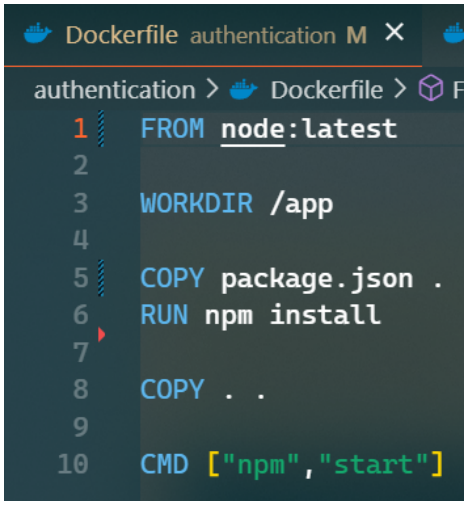
Dockerfile:

The Dockerfile plays an essential role in configuring the services into containers for deployment on Docker. Let's begin with the Dockerfile configured for the front-end services, which include Video Streaming and the Upload Video web:

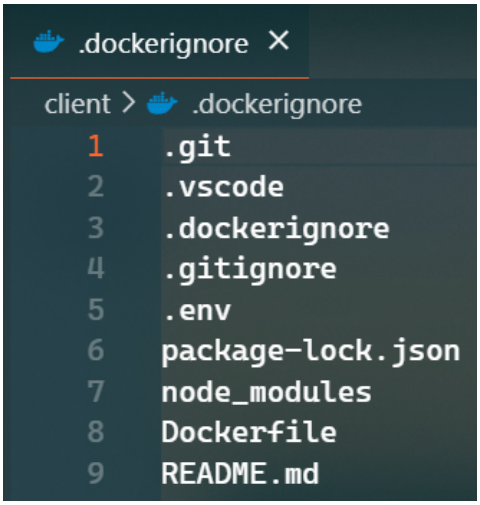


As shown in the figure above, the two front-end services are using the latest node.js image pulled from the Docker Hub. Their working directory is set to /app, as it is important for later on when retrieving user-upload videos to view on the main page. Then, the package.json - which carries important information about the dependencies that need to be installed for the service - is copied into the image's directory and set up the service with

the command RUN npm install. After the image has been initiated with Node.js and its required dependencies, the source code from the host is copied into the working directory /app (with exclusions written in .dockerignore), and then the app is started by running npm run dev command. Let's take a look at the Dockerfile for the back-end services,



Compared to the front-end services that are using Next.js framework, the back-end applications, including authentication and file_system services, are using Node.js, which is a framework that Next.js is based on. The process is quite similar, with the only difference on the command to initiate the app, which is npm start. While delivering the source codes from the host to the image, we have to exclude unnecessary folders and files that are not helpful in initiating the services inside Docker. The exclusions are described in a .dockerignore file:



Docker container architecture:

This part contains the analysis of the docker-compose.yml that is responsible for automatically deploying services. The Video Streaming (Web) is named as client in the docker-compose.

```yaml
client:
  depends_on:
    - authentication
    - file_system
  image: client:latest
  restart: always
  build: ../client
  ports:
    - 4000:3000
  networks:
    - 'project1-apiNetwork'
  volumes:
    - project1-uploads:/app/public/videos
```

The depends_on the authentication and file_system means that it waits for that 2 services to start first in the Docker. The container binds with the image named client, with the tag latest. The restart: always make sure that the service restarts every time when the container is stopped. The build: ../client makes the docker look for the Dockerfile in the client folder from the host machine, to later build them into a Docker container. The ports: - 4000:3000 means that the Docker binds the client's port 3000 to port 4000 on the host machine so that when accessing http://localhost:4000, it will direct to the localhost port 3000 on the client container. 'project1-apiNetwork' is a Docker bridge network that is used to communicate between services. The client container access the user uploads video via a shared Docker volume, project1-uploads. Because of how Next.js access the static storage through the /public/videos, the directory /app/public/videos stores the videos to be displayed in the client service.

```yaml
upload_video:
  depends_on:
    - authentication
  image: upload_video:latest
  restart: always
  build: ../upload_video
  ports:
    - 4004:3000
  networks:
    - 'project1-apiNetwork'
```

```yaml
authentication:
  depends_on:
    - db
  image: authentication:latest
  restart: always
  build: ../authentication
  ports:
    - 3004:3004
  networks:
    - 'project1-apiNetwork'
```

```yaml
file_system:
  depends_on:
    - db
  image: file_system:latest
  restart: always
  build: ../file_system
  ports:
    - 3010:3010
  networks:
    - 'project1-apiNetwork'
  volumes:
    - project1-uploads:/app/public/videos
```

These 3 services are similar to the client, except they build on their own directory and bind with different ports to the host machine. The MySQL DB, on the other hand, has a different docker build structure:

```yaml
db:
  image: mysql:5.7
  restart: always
  networks:
    - 'project1-apiNetwork'
  environment:
    MYSQL_ROOT_PASSWORD: project1
    MYSQL_DATABASE: project1
    MYSQL_USER: project1
    MYSQL_PASSWORD: project1
  command: --default-authentication-plugin=mysql_native_password
  volumes:
    - project1-db:/var/lib/mysql
```

The environment is set for the MySQL login credentials, with a pre-created project1 database. It makes us able to access the MySQL server with the variables set up. Because the backend services communicate to the database internally, it is unnecessary to create a binding port with the host machine like other containers.

Authentication Functionality (front-end):

First, let's take a look at the Video Streaming (Web) on the user authentication route:

```
const [userName, setUserName] = useState('');
const [password, setPassword] = useState('');
```

```
<h5 className="mb-[5px]">User Name:</h5>
<input
  className={style.container__input}
  type="text"
  value={userName}
  onChange={(e) => setUserName(e.target.value)}
/>
<h5 className="mb-[5px]">Password:</h5>
<input
  className={style.container__input}
  type="password"
  value={password}
  onChange={(e) => setPassword(e.target.value)}
/>
```

The username and password input is attached with the userName and password variable, by using useState. Here is what happens when the user clicks the sign-in or register button:

```
const signIn = async () => {
  await Axios.post('http://localhost:3004/login', {
    username: userName,
    password: password,
  }).then((response) => {
    console.log(response);
    if (response.data.message) {
      // wrong username/password
      setLoginState('');
      // status display
      setInvalid(true);
      setCreated(false);
    } else {
      // User info received
      setLoginState(response.data[0].username);
      setInvalid(false);
      setCreated(false);
      window.location.href = '/';
    }
  });
};
```

```
const register = async () => {
  await Axios.post('http://localhost:3004/register', {
    username: userName,
    password: password,
  }).then((response) => {
    setCreated(true);
    setLoginState('');
    setInvalid(false);
  });
};
```

By using Axios, the Video Streaming Service sends the request to the Authentication Server, including the username and password inputs from the user as HTTP Requests.

Authentication Functionality (back-end):

Here is how the authentication service is set up. It uses the express to process the requests from the front end, CORS to allow the requests and cookies to store the user login session. Then, it proceeds to connect with the MySQL database and create a table storing user accounts.

```
/**
 * Initialize connection with mysql-db service
 */
const db = mysql.createConnection({
  user: 'project1',
  host: 'db',
  port: '3306',
  password: 'project1',
  database: 'project1',
  multipleStatements: true,
});
```

```
app.listen(PORT, HOST, () => {
  /**
   * Create table storing users
   * while initializing the service
   */
  let createUsers = `
  CREATE TABLE IF NOT EXISTS users
  (id INT NOT NULL AUTO_INCREMENT,
  username VARCHAR(250) NOT NULL,
  password VARCHAR(250) NOT NULL,
  CONSTRAINT login_pk PRIMARY KEY (id))`;

  db.query(createUsers, (err, result) => {
    console.log(err);
  });
  console.log(`Running Server on http://${HOST}:${PORT}`);
});
```

Authentication service - POST and GET request processing:

```
app.post('/register', (req, res) => {
  const username = req.body.username;
  const password = req.body.password;

  db.query(
    'SELECT * FROM users WHERE username = ? AND password = ?',
    [username, password],
    (err, result) => {
      if (err) {
        res.send({ err: err });
      }
      if (result.length > 0) {
        res.send({ message: 'User Exist' });
      } else {
        db.query(
          'INSERT INTO users (username,password) VALUES (?,?)',
          [username, password],
          (err, result) => {
            if (err) {
              res.send({ err: err });
            }
            if (result) {
              res.status(201).send(result);
            }
          },
        );
      }
    },
  );
});
```

With the register procedure, first thing first, the server access the MySQL database with the username and password variables from the request to check if there is an existing user in the database. If the user exists, it will send back the response to the web with the message ‘User Exist’. On the other hand, if the user does not exist, the server will insert the credentials into the table users, and send back the success response 201 back to the web.

The login route is similar to the register:

```
100  app.post('/login', (req, res) => {
101    const username = req.body.username;
102    const password = req.body.password;
103
104    db.query(
105      'SELECT * FROM users WHERE username = ? AND password = ?',
106      [username, password],
107      (err, result) => {
108        if (err) {
109          res.send({ err: err });
110        }
111        if (result.length > 0) {
112          req.session.user = result;
113          res.send(result);
114        } else {
115          res.send({ message: 'Wrong username/password combination!' });
116        }
117      },
118    );
119  });
```

By querying into the MySQL database with the username and password sent by the Video Streaming Web, it will determine if there is username and password existed in the database. If there is, it will save the result into a cookie session, which will be important for processing the GET requests, as shown in the figure below:

```
app.get('/login', (req, res) => {
  if (req.session.user) {
    res.send({ loggedIn: true, user: req.session.user });
  } else {
    res.send({ loggedIn: false });
  }
});

app.get('/logout', (req, res) => {
  req.session.user = null;
  res.send({ loggedIn: false });
});
```

File System Service:

```
const express = require('express');
const mysql = require('mysql');
const cors = require('cors');

const fileUpload = require('express-fileupload');

// Constants
const PORT = 3010;
const HOST = '0.0.0.0';

// App
const app = express();
app.use(express.json());
app.use(
  cors({
    origin: [
      'http://localhost:4000',
      'http://localhost:4004',
      'http://localhost:3000',
    ],
    methods: ['GET', 'POST'],
    credentials: true,
  }),
);
app.use(fileUpload());
```

Similar to the authentication service, the file system also uses express and cors, with the differences is it does not use cookie, and instead, it uses the express-fileupload.

Connecting with MySQL database is also the same with authentication service:

```
const db = mysql.createConnection({
  user: 'project1',
  host: 'db',
  port: '3306',
  password: 'project1',
  database: 'project1',
  multipleStatements: true,
});
```

```
app.listen(PORT, HOST, () => {
  let createFileSystem = `
CREATE TABLE IF NOT EXISTS uploads
(id INT NOT NULL AUTO_INCREMENT,
filename VARCHAR(250) NOT NULL,
path VARCHAR(250) NOT NULL,
CONSTRAINT path_pk PRIMARY KEY (id))`;

  db.query(createFileSystem, (err, result) => {
    console.log(err);
  });
  console.log(`Running Server on http://${HOST}:${PORT}`);
});
```

It creates the table uploads to store the video name and path information from the user video uploads from the Upload Video (Web). Let's inspect how the Upload Video (Web) sends the video upload request to the server:

```
const [file, setFile] = useState('');
const [filename, setFilename] = useState('Choose File');
const [src, setSrc] = useState('');

/**
 * Display selected file name
 */
const videoSelected = (e: any) => {
  setFile(e.target.files[0]);
  setFilename(e.target.files[0].name);
  setSrc(URL.createObjectURL(e.target.files[0]));
};
```

```
/**
 * Upload video to file system
 */
Axios.defaults.withCredentials = true;
const submit = async (e: any) => {
  setFilename('Posting to database ... ');
  const formData = new FormData();
  formData.append('file', file);
  try {
    await Axios.post('http://localhost:3010/upload', formData, {
      headers: { 'Content-Type': 'multipart/form-data' },
    }).then((res) => {
      console.log(res);
    });
    setFilename('Upload Successfully!');
  } catch (err: any) {
    if (err.status === 500) {
      console.log('Server Error');
    } else {
      console.log(err);
    }
  }
};
```

I have 3 useState variables for tracking the video from user input. The file is for sending as the request to the server, wrapped in the function FormData(). The filename is for displaying the selected file, and the src variable is for previewing the inputted video from the user. The file system service then handles the request:

```
41  /**
42   * POST request /upload
43   * Move video file into shared volume - project1-uploads
44   * Save the file path for accessing videos to shared volume.
45   * Then access it from video streaming web
46   */
47  app.post('/upload', (req, res) => {
48    if (req.files === null) {
49      return res.status(400).json({ msg: 'No file selected' });
50    }
51    console.log(req.files);
52    const file = req.files.file;
53
54    file.mv(`${__dirname}/public/videos/${file.name}`, (err) => {
55      if (err) {
56        console.log(err);
57        return res.status(500).send(err);
58      }
59
60      const filename = file.name;
61      const filePath = `./videos/${filename}`;
62
63      db.query(
64        'INSERT INTO uploads (filename,path) VALUES (?,?)',
65        [filename, filePath],
66        (err, result) => {
67          if (err) {
68            res.send({ err: err });
69          }
70          if (result) {
71            res.status(201).send(file);
72          }
73        },
74      );
75    });
76  });
```

After checking in the request if it has any files attached, it will create a variable file containing the file from the request. Then, using the file.mv, it moves the file into the Docker shared folder, in the /app/public/videos directory (__dirname of the file system server is /app, as set before in the Dockerfile). If no errors occurred, it will query to MySQL database to store the video file name and path, for later retrieving them to view the videos.

```
const [videos, setVideos] = useState([]);
useEffect(() => {
  try {
    const getVideos = async () => {
      await Axios.get('http://localhost:3010/view').then((response) => {
        if (response) {
          console.log(response.data);
          setVideos(response.data);
        }
      });
    };
    getVideos();
  } catch (err) {
    console.log(err);
  }
}, []);
console.log(videos);
```

The figure above shows how the Video Streaming Service sends the GET request, which then gets processed by the File System Service, as shown in the figure below:

```
/**
 * Retrieve list of videos from mysql-db
 */
app.get('/view', (req, res) => {
  db.query('SELECT * FROM uploads', (err, result) => {
    if (err) {
      return res.send({ err: err });
    }
    if (result.length > 0) {
      res.send(result);
    } else {
      res.send(null);
    }
  });
});
```

Using the SELECT * FROM uploads, the server queries it to the MySQL database to retrieve the video information. The Video Streaming Web then uses them to determine the video paths in the Docker shared volume, which is then able to link it to display on the front-end application.

Conclusion:

The project has taught me important factors on how to create automatically built Docker containers. I have learnt to use the correct resource and options to connect them together and make them communicate seamlessly. The project makes me understand more about the Next.js framework, how to use its API Request, and how to retrieve and display static files, such as videos or images. It gives me a huge time in inspecting and interacting with Docker containers and learning how to control and configure them effectively.