Assignment 2

Group: Youngjae, Lukas

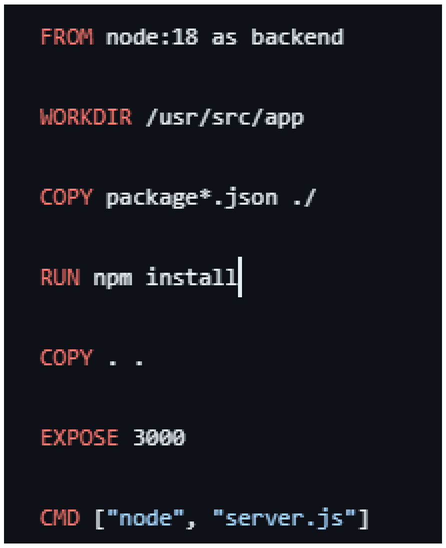
System Design:

- Use of 2 docker images, one for the frontend and one for backend.
- Our custom designed Workflows are triggered by GitActions on push to our main development branch.
- Docker images are built using newly committed code, tested, and then pushed to docker hub.
- Deployment to Droplet utilizing the newly pushed images from docker hub.

Part 1: CI Pipeline

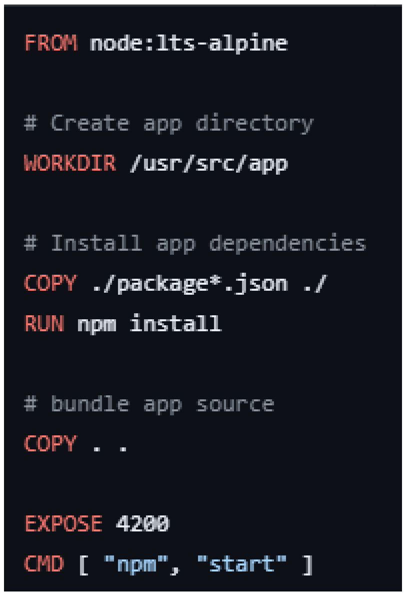
We created a separate docker image for both front end and backend. They are as follows:

Backend:



```
FROM node:18 as backend

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "server.js"]
```

Frontend:



```
FROM node:lts-alpine

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
COPY ./package*.json ./
RUN npm install

# bundle app source
COPY . .

EXPOSE 4200
CMD [ "npm", "start" ]
```

The docker files have similar commands. They create a directory, install all library dependencies, and copy in the source code of the application. It then exposes the respective ports that the backend and front end run on and then begin running.

Using these docker files we can then build the image using the docker build command in the respective directories (frontend and backend).

```
docker build -t frontend:latest
```

```
docker build -t backend:latest
```

We then tagged and pushed these images to a Docker registry using the following commands.

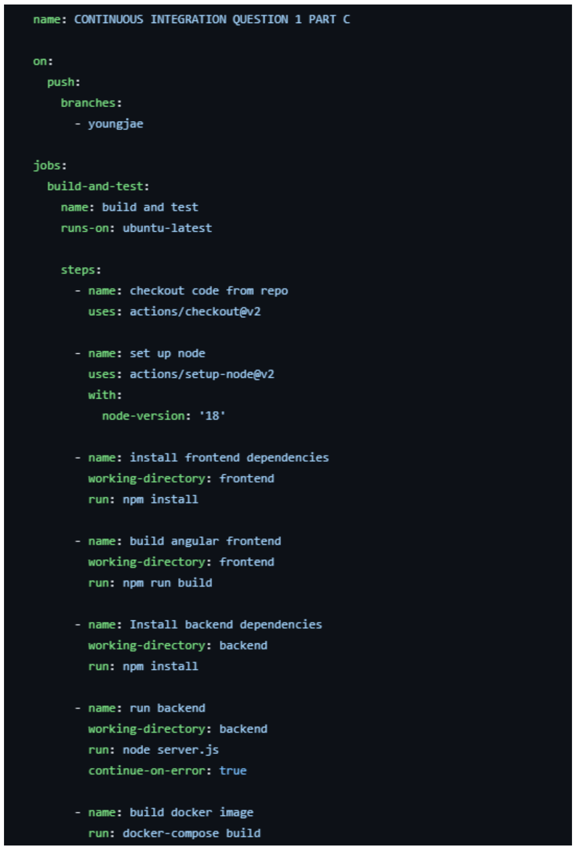
```
docker tag frontend:latest Youngjaeheo2002/frontend
```

```
docker tag backend:latest Youngjaeheo2002/backend
```

```
docker push Youngjaeheo2002/frontend
```

```
docker push Youngjaeheo2002/backend
```

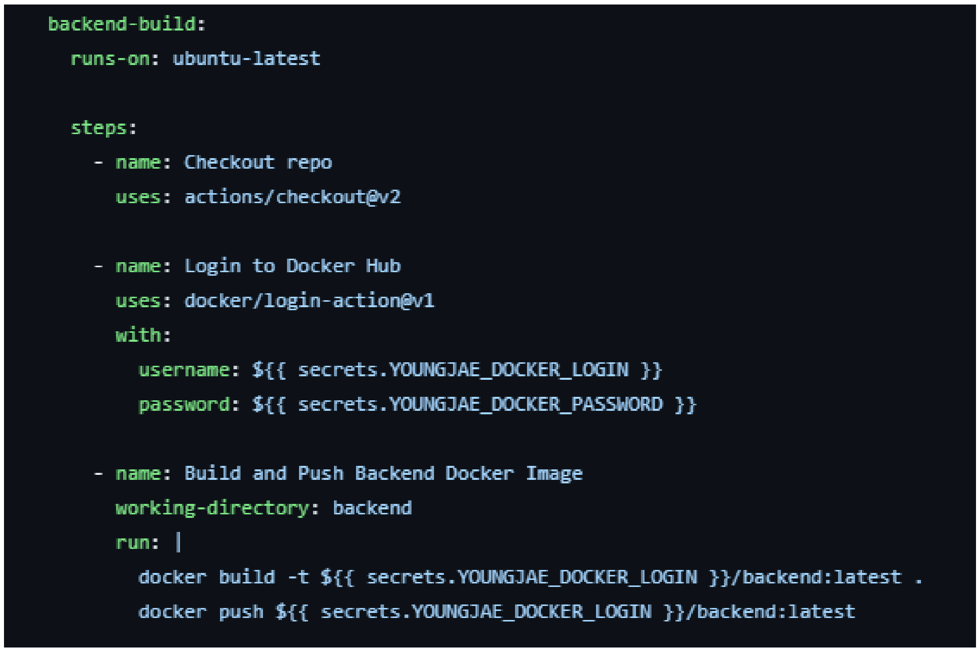
For the GitActions CI workflow we needed to define a YAML file for the backend and frontend to define what our workflow is. We define our workflow to automatically trigger on a push to our frontend or backend directories. The workflow consists of checking out the repository for git actions to have access to the code and then rebuilding the docker images using the build command defined in our docker-compose YAML file.



```yaml
name: CONTINUOUS INTEGRATION QUESTION 1 PART C

on:
  push:
    branches:
      - youngjae

jobs:
  build-and-test:
    name: build and test
    runs-on: ubuntu-latest

    steps:
      - name: checkout code from repo
        uses: actions/checkout@v2

      - name: set up node
        uses: actions/setup-node@v2
        with:
          node-version: '18'

      - name: install frontend dependencies
        working-directory: frontend
        run: npm install

      - name: build angular frontend
        working-directory: frontend
        run: npm run build

      - name: Install backend dependencies
        working-directory: backend
        run: npm install

      - name: run backend
        working-directory: backend
        run: node server.js
        continue-on-error: true

      - name: build docker image
        run: docker-compose build
```

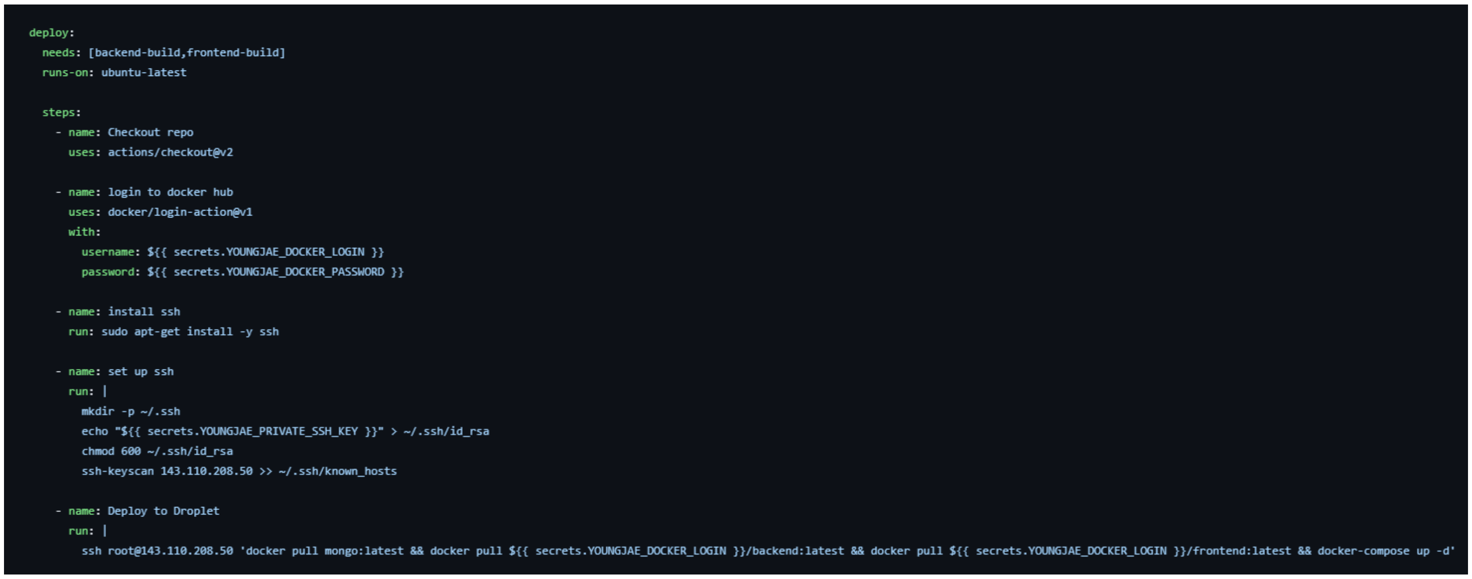
Part 2: CD Pipeline

In this section we first edited our workflow to push our newly built docker images to dockerhub. This is done using the following steps for both backend and frontend:



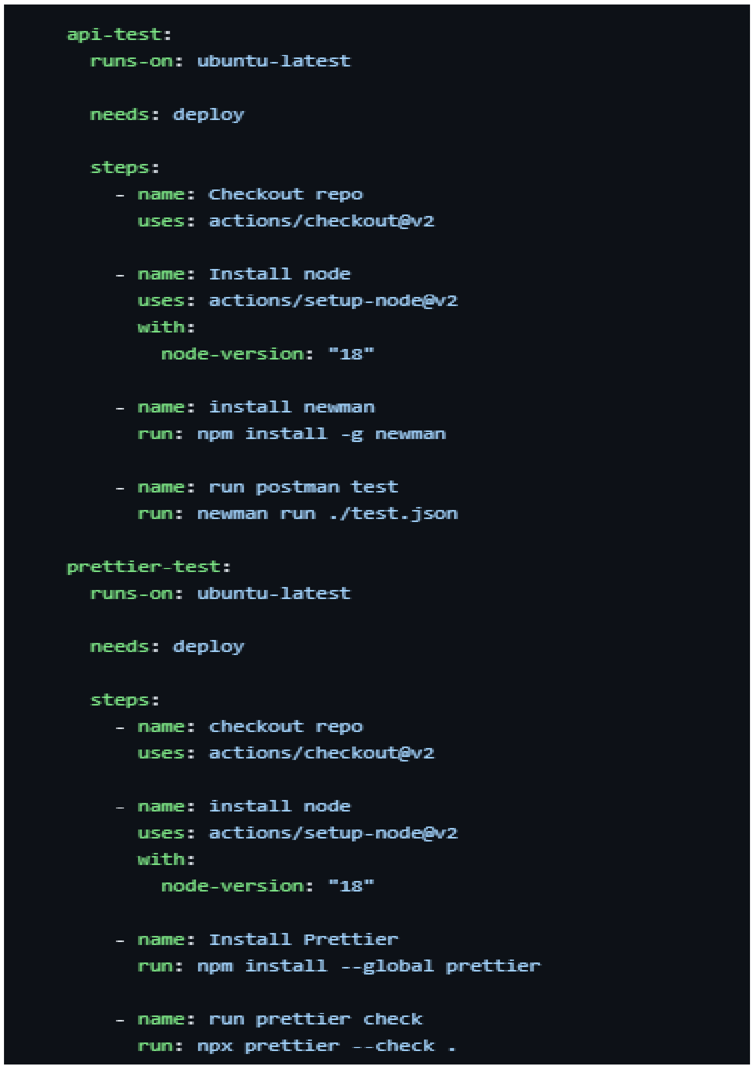
```
  backend-build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repo
        uses: actions/checkout@v2

      - name: Login to Docker Hub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.YOUNGJAE_DOCKER_LOGIN }}
          password: ${{ secrets.YOUNGJAE_DOCKER_PASSWORD }}

      - name: Build and Push Backend Docker Image
        working-directory: backend
        run: |
          docker build -t ${{ secrets.YOUNGJAE_DOCKER_LOGIN }}/backend:latest .
          docker push ${{ secrets.YOUNGJAE_DOCKER_LOGIN }}/backend:latest
```

We decided to do our deployment to Droplet. To do this we need to access our droplet service through SSH and then pull the docker images for the front end and back end and then start the images in a container through our docker-compose file. We now have successfully deployed the frontend and backend of our application on the Droplet VM. These commands are demonstrated below:



```
deploy:
  needs: [backend-build,frontend-build]
  runs-on: ubuntu-latest

  steps:
    - name: Checkout repo
      uses: actions/checkout@v2

    - name: login to docker hub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.YOUNGJAE_DOCKER_LOGIN }}
        password: ${{ secrets.YOUNGJAE_DOCKER_PASSWORD }}

    - name: install ssh
      run: sudo apt-get install -y ssh

    - name: set up ssh
      run: |
        mkdir -p ~/.ssh
        echo "${{ secrets.YOUNGJAE_PRIVATE_SSH_KEY }}" > ~/.ssh/id_rsa
        chmod 600 ~/.ssh/id_rsa
        ssh-keyscan 143.110.208.50 >> ~/.ssh/known_hosts

    - name: Deploy to Droplet
      run: |
        ssh root@143.110.208.50 'docker pull mongo:latest && docker pull ${{ secrets.YOUNGJAE_DOCKER_LOGIN }}/backend:latest && docker pull ${{ secrets.YOUNGJAE_DOCKER_LOGIN }}/frontend:latest && docker-compose up -d'
```

We then set up automated testing for our deployed container to ensure functionality and performance. We did this by running a set of automated tests to run at the deployment of the containers. In our YAML file we added *api-test* and *prettier-test* which runs tests defined in the root directory in a test.json file. The YAML is seen below:



```yaml
api-test:
  runs-on: ubuntu-latest

  needs: deploy

  steps:
    - name: Checkout repo
      uses: actions/checkout@v2

    - name: Install node
      uses: actions/setup-node@v2
      with:
        node-version: "18"

    - name: install newman
      run: npm install -g newman

    - name: run postman test
      run: newman run ./test.json

prettier-test:
  runs-on: ubuntu-latest

  needs: deploy

  steps:
    - name: checkout repo
      uses: actions/checkout@v2

    - name: install node
      uses: actions/setup-node@v2
      with:
        node-version: "18"

    - name: Install Prettier
      run: npm install --global prettier

    - name: run prettier check
      run: npx prettier --check .
```

As seen in YAML, we use keyword *needs* to ensure that the tests are run on the container after the deployment is finished. These tests target the backend to ensure our APIs are running correctly. These functionality checks include login tests, logout tests, delete users' tests, etc.