Ultimate Guideline

Getting Started

- The Below Section serves as a Guideline for someone who's willing to learn and actually run the code to see how the Application works.
- I started with getting the backend API running since the frontend web application depends on the API.

- `Prerequisites 1 (Description)` :
  - The depends on the Node Package Manager (NPM). You will need to download and install Node from https://nodejs.com/en/download. This will allow you to be able to run `npm` commands.
  - Environment variables will need to be set. These environment variables include database connection details that should not be hard-coded into the application code.

- `Prerequisites 2 (Setup)` :
  - Environment Script:
    - I used the file named `set_env.sh` to configure my variables on my local development environment.
    - I do *not* want your credentials to be stored in git. After pulling this `starter` project, I run the following command to tell git to stop tracking the script in git but keep it stored locally. This way, I can use the script for your convenience and reduce risk of exposing your credentials. `git rm --cached set_env.sh`
    - Afterwards, I can prevent the file from being included in your solution by adding the file to our `.gitignore` file.
  - Database:
    - Create a `PostgreSQL` database either locally or on `AWS RDS` . The database is used to store the application's metadata.
    - We will need to use password authentication for this project. This means that a username and password is needed to authenticate and access the database.
    - The port number will need to be set as `5432` . This is the typical port that is used by PostgreSQL so it is usually set to this port by default.
    - Once your database is set up, set the config values for environment variables prefixed with `POSTGRES_` in `set_env.sh` .
    - If you set up a local database, your `POSTGRES_HOST` is most likely `localhost`
    - If you set up an RDS database, your `POSTGRES_HOST` is most likely in the following format: `***.****.us-west-1.rds.amazonaws.com` . You can find this value in the `AWS console's` RDS dashboard.
  - S3:
    - Create an `AWS` S3 `bucket` . The S3 bucket is used to `store` images that are displayed in Udagram.
    - Set the `config` values for environment variables prefixed with `AWS_` in `set_env.sh` .
  - Backend API:
    - Launch the `Backend API` locally. The API is the application's `interface` to S3 and the database.
    - To download all the `package` dependencies, run the `command` from the directory `udagram-api/` : `bash npm cache clear --force npm install .`
    - To run the application locally, run: `bash npm run dev`
  - You can visit `http://localhost:8080/api/v0/feed` in your web browser to verify that the application is running. You should see a `JSON` payload. Feel free to play around with `Postman` to test the `API` .
  - Frontend App:
    - Launch the `Frontend` app locally.
    - To download all the `package` dependencies, run the `command` from the directory `udagram-frontend/` : `bash npm install .`
    - Install `Ionic Framework's Command Line` tools for us to `build` and run the application:
      - The `package` name has changed from ionic to `@ionic/cli!`
        - To update, run: npm uninstall -g ionic `bash npm i -g @ionic/cli`
        - Then run: `bash npm i -g @ionic/cli`
    - Prepare your application by compiling them into static files. `bash ionic build`
    - Run the application locally using files created from the `ionic build` command. `bash ionic serve`
    - You can visit `http://localhost:8100` in your web browser to verify that the application is running. You should see a web interface.
  - Tips
    - Take a look at `udagram-api` -- does it look like we can divide it into two modules to be deployed as separate microservices?
    - The `.dockerignore` file is included for your convenience to not copy `node_modules` . Copying this over into a `Docker container` might cause issues if your local environment is a different operating system than the `Docker image` (ex. `Windows` or `MacOS` vs. `Linux`).
    - It's useful to `lint` your code so that changes in the codebase adhere to a coding standard. This helps alleviate issues when developers use different styles of coding. `eslint` has been set up for `TypeScript` in the codebase for you.
      - To lint your code, run the following: `bash npx eslint --ext .js,.ts src/`
      - To have your code fixed automatically, run `bash npx eslint --ext .js,.ts src/ --fix`
    - `set_env.sh` is really for your backend application. Frontend applications have a different notion of how to store configurations.

Configurations for the application endpoints can be configured inside of the `environments/environment.*ts` files.

- In `set_env.sh`, environment variables are set with `export $VAR=value`. Setting it this way is not permanent; every time you open a new terminal, you will have to run `set_env.sh` to reconfigure your environment variables. To verify if your environment variable is set, you can check the variable with a command like `echo $POSTGRES_USERNAME`.

Running the project locally in a Multi-Container environment

- The objective of this part of the project is to:
  - `Refactor` the `monolith` application to `microservices`
  - Set up each `microservice` to be run in its own `Docker` container
- Once you refactor the Udagram application, it will have the following services running internally:
  - `Backend /user/ service`: allows users to register and log into a web client.
  - `Backend /feed/ service`: allows users to post photos, and process photos using image filtering.
  - `Frontend`: It is a basic Ionic client web application that acts as an interface between the user and the backend services.
  - `Nginx as a reverse proxy server`: for resolving multiple services running on the same port in `separate containers`. When different `backend services` are running on the same `port`, then a `reverse proxy` server directs client requests to the appropriate backend server and retrieves resources on behalf of the `client`.
- Navigate to the project directory, and set up the environment variables again

```
source set_env.sh
```

- `Docker Containers:`
  - Use `Docker compose` to build and run `multiple` Docker containers
  - `Create images:`
    - In the project's parent directory, create a `docker-compose-build.yaml file`.
    - It will create an image for each individual service. Then, you can run the following command to create images `locally` then run the images.
  - Make sure the Docker `services` are running in your `local` machine.
  - Remove unused and dangling images `docker image prune --all`
  - Run this command from the directory where you have the `docker-compose-build.yaml` file present: `bash docker-compose -f docker-compose-build.yaml build --parallel`
  - Docker images running
  - Run the container `bash docker-compose up`
  - Visit http://localhost:8100 in your web browser to verify that the application is running.

Backend api feed

- Local host server running
- The containerized application running
- Images of a succesful build and deploy of docker to dockerhub using Gitlab
- `Creating the HorizontalPodAutoscaler:`
  - Installation Command:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

  - Create the HorizontalPodAutoscaler:(Do this for all deployment)

```
 kubectl autoscale deployment backend-feed --cpu-percent=70 --min=3 --max=5
kubectl autoscale deployment backend-user --cpu-percent=70 --min=3 --max=5
kubectl autoscale deployment frontend --cpu-percent=70 --min=3 --max=5
kubectl autoscale deployment reverseproxy --cpu-percent=70 --min=3 --max=5
```

  - You can check the current status of the newly-made HorizontalPodAutoscaler, by running:

```
kubectl get hpa
```
