DevOps Intern Assignment: Microservices with Nginx and Docker Compose

This document outlines a project demonstrating microservices architecture using Golang and Python Flask, orchestrated with Docker Compose and fronted by Nginx as a reverse proxy. It details the project structure, individual service implementations, Dockerfile configurations, Nginx setup, and instructions for running and accessing the services.

Project Structure and Components

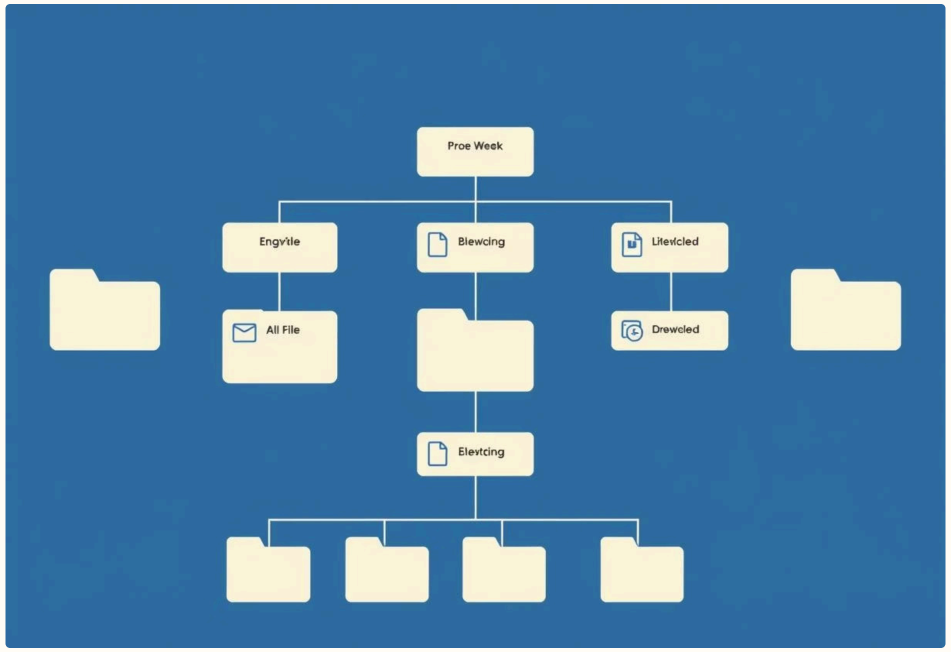
The project is organized into a clear folder structure, encapsulating each microservice and the Nginx configuration. This modular approach facilitates independent development and deployment of each component.

Overall Structure

```
devops-intern-assignment/
├── docker-compose.yml
├── service_1/
│   ├── main.go
│   └── Dockerfile
├── service_2/
│   ├── app.py
│   └── Dockerfile
└── nginx/
    ├── nginx.conf
    └── Dockerfile
```

Key Components

- **service_1:** A microservice built with Golang.
- **service_2:** A microservice built with Python Flask.
- **nginx:** Acts as a reverse proxy for routing requests to the services.
- **docker-compose.yml:** Defines and connects all services, enabling multi-container Docker applications.



Golang Microservice (service_1)

The Golang-based microservice, **service_1**, provides two basic API endpoints: /ping for health checks and /hello for a greeting message. It listens on port 8001 and returns JSON responses.

> **service_1/main.go**
>
> ```
> package main
> import (
>         "encoding/json"
>         "log"
>         "net/http"
> )
> func main() {
>         http.HandleFunc("/ping", func(w http.ResponseWriter, r *http.Request) {
>                 jsonResponse(w, map[string]string{"status": "ok", "service": "1"})
>         })
>         http.HandleFunc("/hello", func(w http.ResponseWriter, r *http.Request) {
>                 jsonResponse(w, map[string]string{"message": "Hello from Service 1"})
>         })
>         log.Println("Service 1 listening on port 8001...")
>         log.Fatal(http.ListenAndServe(":8001", nil))
> }
> func jsonResponse(w http.ResponseWriter, data map[string]string) {
>         w.Header().Set("Content-Type", "application/json")
>         json.NewEncoder(w).Encode(data)
> }
> ```

The Dockerfile for **service_1** is based on golang:1.21-alpine, copies the source code, builds the executable, and defines the command to run the service.

> **service_1/Dockerfile**
>
> ```
> FROM golang:1.21-alpine
> WORKDIR /app
> COPY . .
> RUN go build -o service1
> CMD ["./service1"]
> ```

Python Flask Microservice (service_2)

The Python Flask-based microservice, **service_2**, mirrors the functionality of **service_1** by offering `/ping` and `/hello` endpoints. It runs on port 8002 and returns JSON responses.

> **service_2/app.py**
>
> ```
> from flask import Flask, jsonify
> app = Flask(__name__)
> @app.route("/ping")
> def ping():
>     return jsonify({"status": "ok", "service": "2"})
> @app.route("/hello")
> def hello():
>     return jsonify({"message": "Hello from Service 2"})
> if __name__ == "__main__":
>     app.run(host="0.0.0.0", port=8002)
> ```

The `Dockerfile` for **service_2** uses `python:3.11-slim`, installs Flask, and sets up the command to execute the Python application.

> **service_2/Dockerfile**
>
> ```
> FROM python:3.11-slim
> WORKDIR /app
> COPY . .
> RUN pip install flask
> CMD ["python", "app.py"]
> ```

Nginx Reverse Proxy Configuration

Nginx acts as the central entry point for external requests, routing them to the appropriate microservice based on the URL path. This setup provides a single point of access and can handle load balancing and other proxy functionalities.

nginx/nginx.conf

```
events {}
http {
    server {
        listen 80;
        location /service1/ {
            proxy_pass http://service1:8001/;
        }
        location /service2/ {
            proxy_pass http://service2:8002/;
        }
    }
}
```

The `Dockerfile` for Nginx is straightforward, building upon the official `nginx:alpine` image and copying the custom configuration file into the container.

nginx/Dockerfile

```
FROM nginx:alpine
COPY nginx.conf /etc/nginx/nginx.conf
```

Docker Compose Orchestration

The docker-compose.yml file is the core of this project, defining all services, their build contexts, container names, and dependencies. It also configures health checks for the microservices and port mapping for Nginx.

docker-compose.yml

```
version: '3.8'
services:
  service1:
    build: ./service_1
    container_name: service1
    healthcheck:
      test: ["CMD", "curl", "-f", "http://service1:8001"]
      interval: 30s
      timeout: 10s
      retries: 3
  service2:
    build: ./service_2
    container_name: service2
    healthcheck:
      test: ["CMD", "curl", "-f", "http://service2:8002"]
      interval: 30s
      timeout: 10s
      retries: 3
  nginx:
    build: ./nginx
    container_name: nginx_proxy
    ports:
      - "9090:80"
    depends_on:
      - service1
      - service2
```

This configuration ensures that all services are built and started in the correct order, with Nginx depending on the microservices to be available.

Running the Project

To deploy and run the entire microservices stack, a few simple Docker Compose commands are all that's needed. This process handles building the Docker images and starting the containers as defined in the `docker-compose.yml` file.

Stop Existing Containers

Ensure no conflicting containers are running by stopping and removing any previous instances of the project.

```
$ docker-compose down
```

Build and Start Services

Build the Docker images for each service and then start all containers in detached mode.

```
$ docker-compose up --build
```

The `--build` flag ensures that the Docker images are rebuilt from their respective Dockerfiles, incorporating any recent code changes.

Accessing the Services

Once the Docker Compose project is up and running, you can access the microservices through the Nginx reverse proxy via your web browser. Nginx is configured to listen on port 9090 on your localhost and route requests to the appropriate backend service.

Service 1 Endpoints

- http://localhost:9090/service1/ping
- http://localhost:9090/service1/hello

Service 2 Endpoints

- http://localhost:9090/service2/ping
- http://localhost:9090/service2/hello

These URLs demonstrate how Nginx effectively proxies requests, making the individual microservices accessible under a unified entry point.