

Fusion DevOps Challenge

Standard Operating Procedure

Rajat Karad

rajatkarad007@gmail.com — 7620779767

Level 1: Full Stack Deployment on AWS

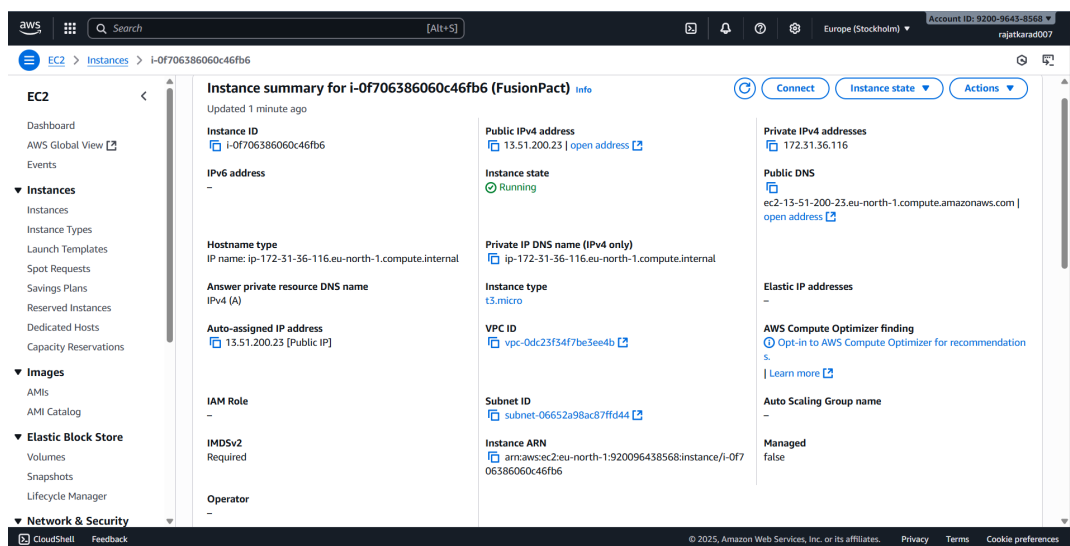
Step 1: Launch EC2 Instance and Connect

Configure Security Groups with the following inbound rules:

- SSH: 22 → your IP
- HTTP: 80 → anywhere
- Backend API port: 8000 → anywhere

Connect to the EC2 instance using SSH with your private key:

```
ssh -i "keypair.pem" ubuntu@<EC2-IP>
```



EC2 instance connected via SSH with security groups configured

Step 2: Clone Repository and Install Docker

After successfully accessing the instance, update system packages and install Docker and Docker Compose:

```
git clone <repository-url>
cd <project-directory>

sudo apt update

sudo apt install -y docker.io

sudo systemctl enable docker

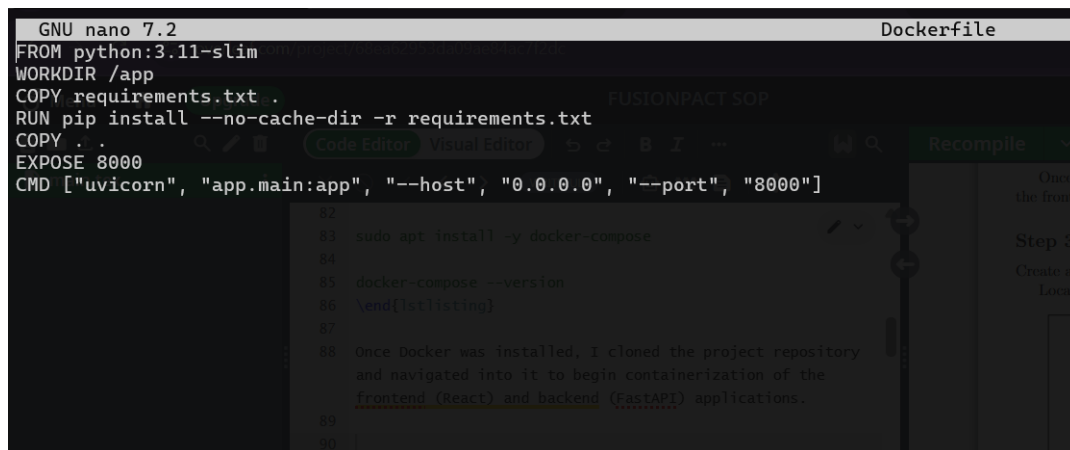
sudo systemctl start docker

sudo usermod -aG docker ubuntu

sudo apt install -y docker-compose

docker-compose --version
```

Once Docker was installed, I cloned the project repository and navigated into it to begin containerization of the frontend (React) and backend (FastAPI) applications.



```
GNU nano 7.2 Dockerfile
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

82
83 sudo apt install -y docker-compose
84
85 docker-compose --version
86 \end{listlisting}
87
88 Once Docker was installed, I cloned the project repository
and navigated into it to begin containerization of the
frontend (React) and backend (FastAPI) applications.
89
90
```

Repository cloned and Docker/Docker Compose installed successfully

Step 3: Create Backend Dockerfile

Create a Dockerfile in the backend directory with the appropriate Python configuration.

Location: backend/Dockerfile

```
GNU nano 7.2 Dockerfile
FROM nginx:alpine
WORKDIR /usr/share/nginx/html
RUN rm -rf /*
COPY . .
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]

82
83 sudo apt install -y docker-compose
84
85 docker-compose --version
86 \end{lstlisting}
87
88 Once Docker was installed, I cloned the project repository
89 and navigated into it to begin containerization of the
90 frontend (React) and backend (FastAPI) applications.
91
92 \subsection*{Step 3: Create Backend Dockerfile}
93
94 Create a Dockerfile in the backend directory with the
95 appropriate Python configuration.
96 Location: \texttt{backend/Dockerfile}
```

Backend Dockerfile with Python dependencies and FastAPI setup

Step 4: Create Frontend Dockerfile

Create a Dockerfile in the frontend directory with Node.js and Nginx configuration for React application.

Location: frontend/Dockerfile

```

GNU nano 7.2 docker-compose.yml
version: '3'

services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    volumes:
      - ./backend/app/app/app
      - ./backend/db/app/db

  frontend:
    build: ./frontend
    ports:
      - "80:80"

  prometheus:
    image: prom/prometheus
    container_name: prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus

  grafana:
    image: grafana/grafana
    container_name: grafana
    volumes:
      - grafana_data:/var/lib/grafana
    ports:
      - "3000:3000"

  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    container_name: cadvisor
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro

```

Frontend Dockerfile with Node.js build and Nginx reverse proxy

Step 5: Create docker-compose.yml

Create the docker-compose configuration file at the root directory. Define a named Docker volume called `sqlite_data` and mount it in the backend container at `/app/db`. This ensures that even if containers are stopped or recreated, the `users.db` file remains intact and persists across restarts.

Location: `docker-compose.yml`

```

ubuntu@ip-172-31-36-116:~/fusionpack-devops-challenge$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
c2809a8e4e5d   fusionpack-devops-challenge_frontend "/docker-entrypoint..." 4 hours ago   Up 4 hours   0.0.0.0:80->80/tcp
c8aee49e5d44   fusionpack-devops-challenge_frontend_1 "/bin/prometheus --c..." 4 hours ago   Up 4 hours   0.0.0.0:9090->9090/tcp
7c3fa5a28aba   gcr.io/cadvisor/cadvisor:latest    "/usr/bin/cadvisor -..." 4 hours ago   Up 4 hours (healthy) 0.0.0.0:8080->8080/tcp
7bf27ab0d95    grafana/grafana                    "/run.sh"                4 hours ago   Up 4 hours   0.0.0.0:3000->3000/tcp

```

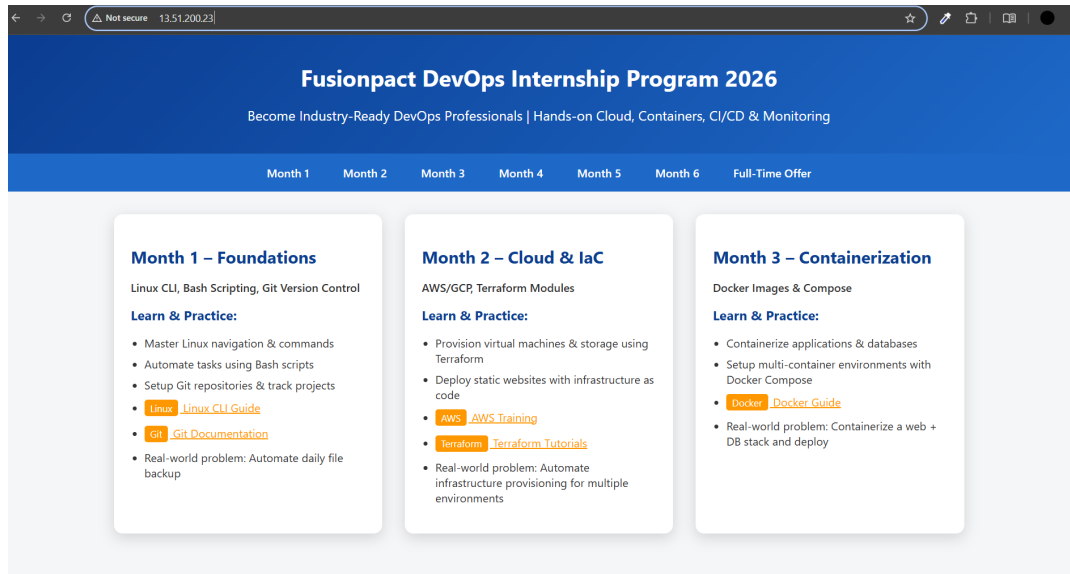
docker-compose.yml with frontend, backend, prometheus, grafana, cadvisor services, and sqlite_data volume

Step 6: Deploy Application

Build and start all application containers using Docker Compose:

```
docker-compose up -d
```

```
docker ps
```

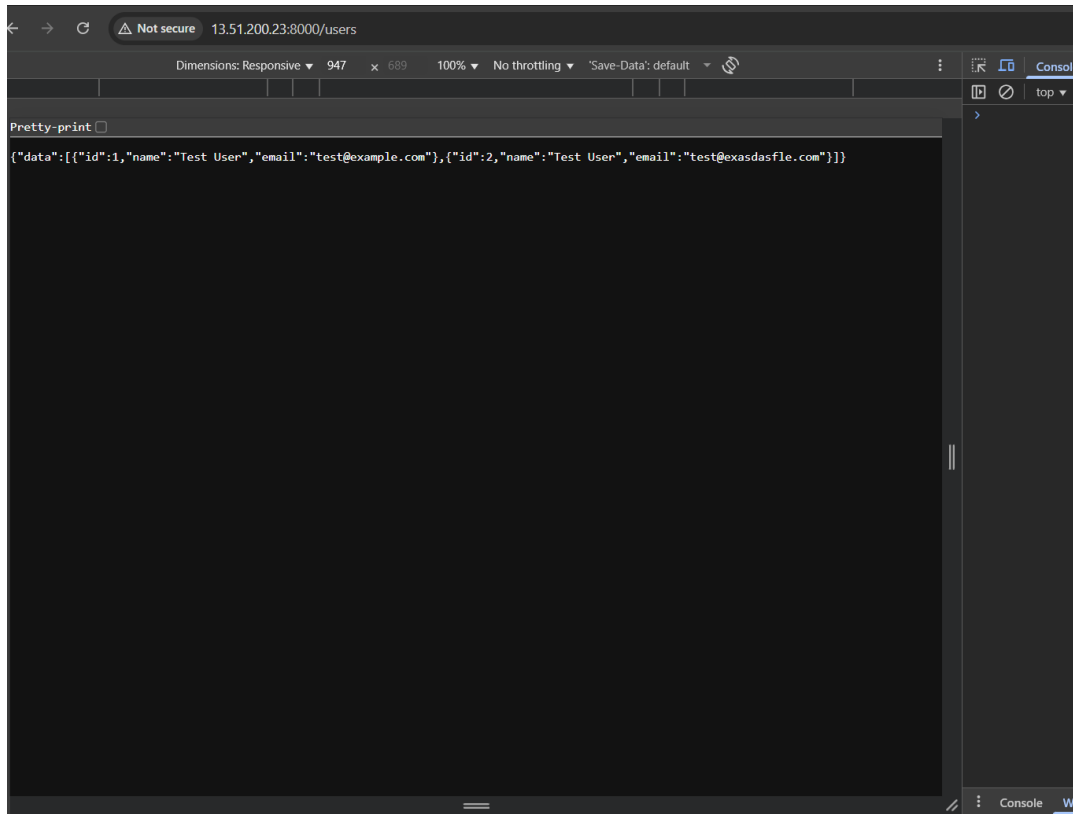


docker ps output showing all containers running successfully

Step 7: Test Public Access

Verify that the application is accessible from your local browser:

- Frontend: `http://<EC2_PUBLIC_IP>`
- Backend API: `http://<EC2_PUBLIC_IP>:8000`



Frontend and Backend API responding and accessible

Level 2: Observability - Prometheus & Grafana

Step 1: Install Prometheus Client and Dependencies

Activate the backend virtual environment and install the Prometheus client library along with required dependencies:

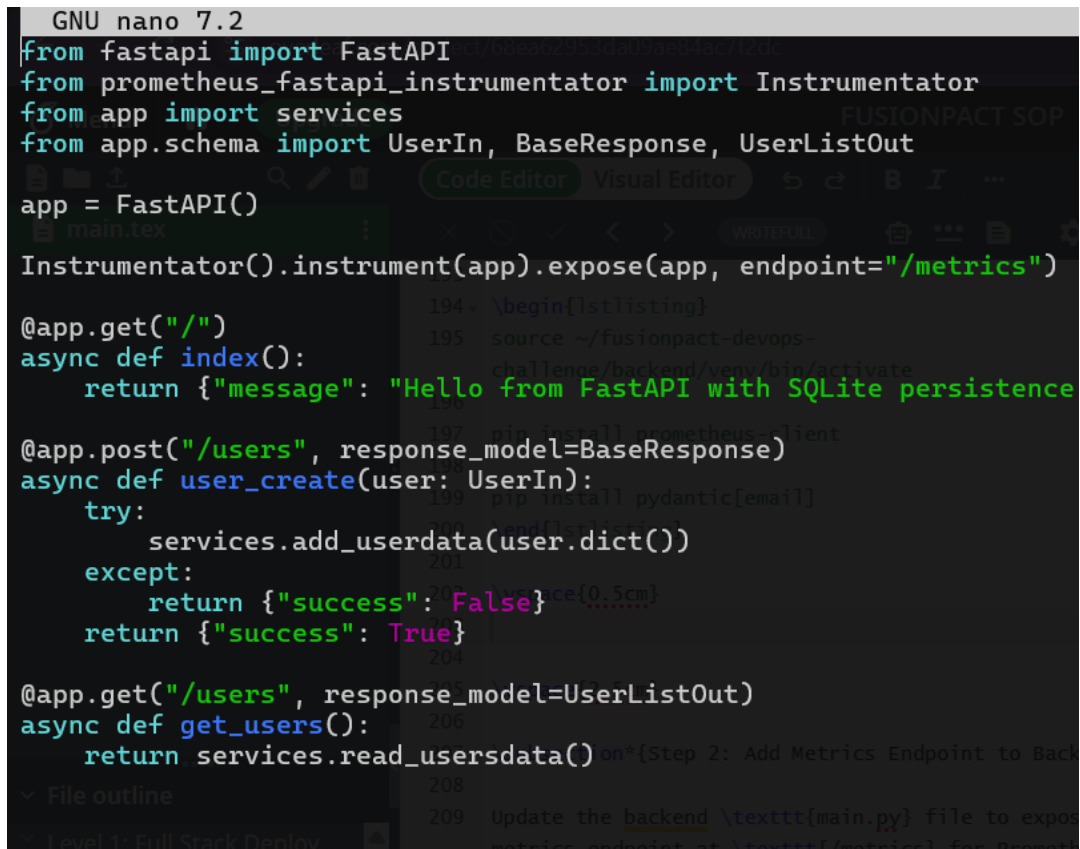
```
source ~/fusionpact-devops-challenge/backend/venv/bin/activate
```

```
pip install prometheus-client
```

```
pip install pydantic[email]
```

Step 2: Add Metrics Endpoint to Backend

Update the backend `main.py` file to expose the metrics endpoint at `/metrics` for Prometheus to scrape application metrics.



```
GNU nano 7.2
from fastapi import FastAPI
from prometheus_fastapi_instrumentator import Instrumentator
from app import services
from app.schema import UserIn, BaseResponse, UserListOut

app = FastAPI()
Instrumentator().instrument(app).expose(app, endpoint="/metrics")

@app.get("/")
async def index():
    return {"message": "Hello from FastAPI with SQLite persistence"}

@app.post("/users", response_model=BaseResponse)
async def user_create(user: UserIn):
    try:
        services.add_userdata(user.dict())
    except:
        return {"success": False}
    return {"success": True}

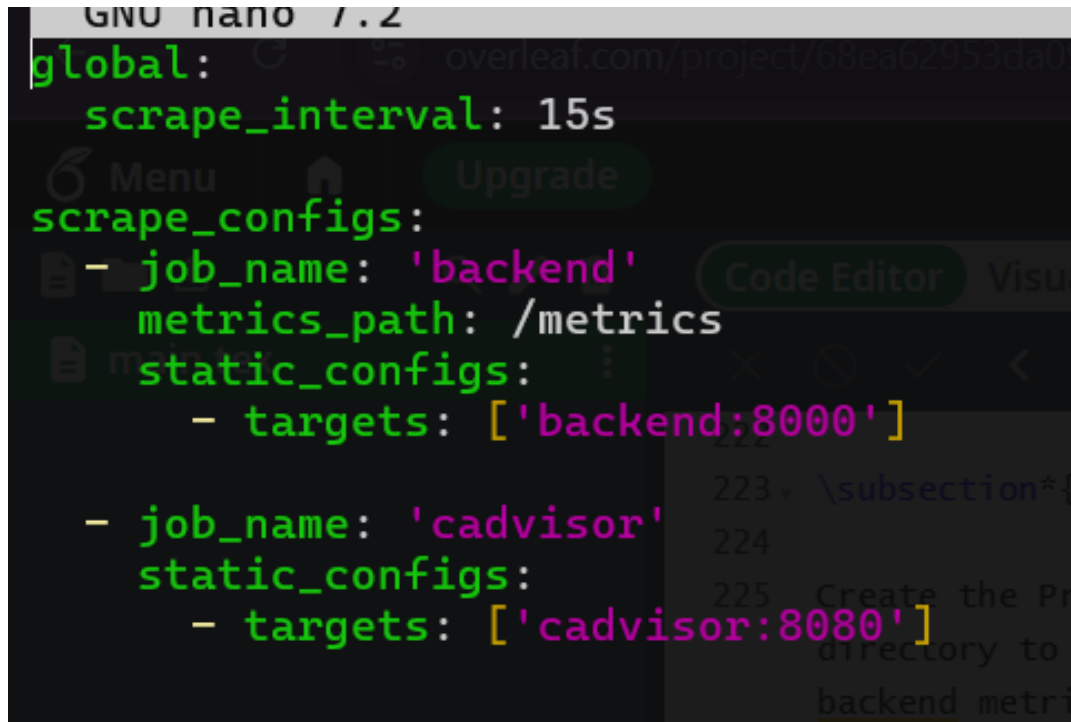
@app.get("/users", response_model=UserListOut)
async def get_users():
    return services.read_usersdata()
```

Backend main.py with /metrics endpoint integrated

Step 3: Create prometheus.yml Configuration

Create the Prometheus configuration file at the root directory to define scrape targets and intervals for backend metrics and cAdvisor.

Location: prometheus.yml



```
GNU nano 7.2
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'backend'
    metrics_path: /metrics
    static_configs:
      - targets: ['backend:8000']
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']
```

prometheus.yml with backend and cAdvisor scrape configurations

Step 4: Update docker-compose.yml with Monitoring Services

Add Prometheus, Grafana, and cAdvisor services to the existing docker-compose.yml file. This enables full observability with metrics collection, visualization, and infrastructure monitoring.

Step 5: Redeploy with Monitoring Stack

Tear down existing containers and redeploy with the complete monitoring services:

```
docker-compose down
```

```
docker-compose up --build -d
```

```
docker ps
```

Step 6: Access Prometheus Dashboard

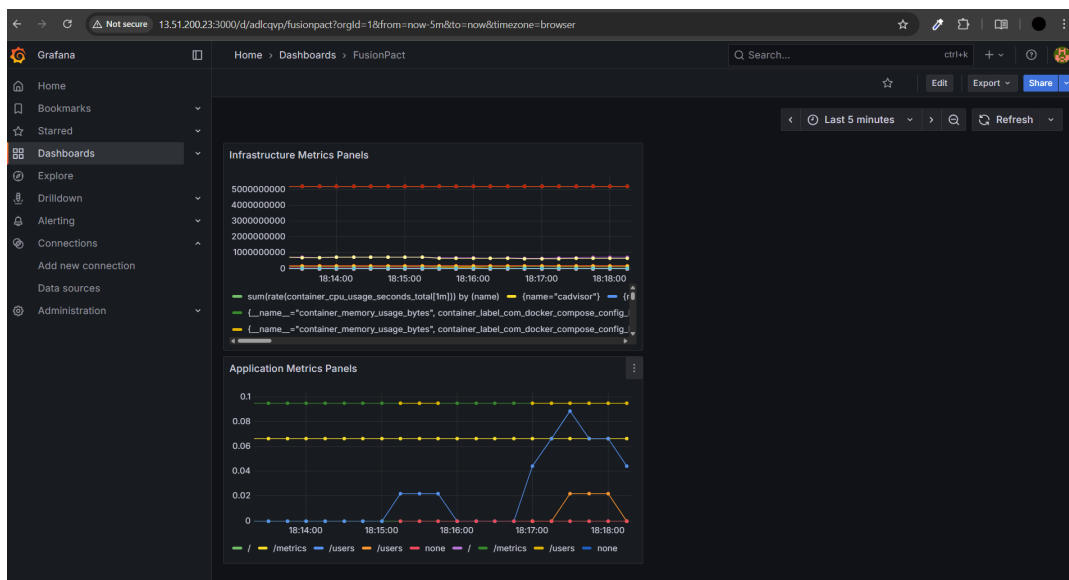
Open Prometheus dashboard in your browser to verify metrics collection:

- URL: `http://<EC2_PUBLIC_IP>:9090`
- Query available metrics to verify data collection from backend

Step 7: Configure Grafana Dashboards

Access Grafana and set up dashboards for application and infrastructure metrics:

- URL: `http://<EC2_PUBLIC_IP>:3000`
- Default login: admin / admin
- Add Prometheus as data source with URL: `http://prometheus:9090`



Grafana login and Prometheus data source successfully added

Level 3: CI/CD Pipeline with GitHub Actions

Step 1: Create GitHub Actions Workflow Directory

Create the necessary directory structure for GitHub Actions workflows:

```
mkdir -p .github/workflows
```

Step 2: Create main.yml Workflow File

Create the CI/CD workflow configuration file that defines the build, test, and deployment pipeline.

Location: `.github/workflows/main.yml`

```
name: Fusionpack CI/CD (EC2 Direct Deploy)

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      #1 Checkout your code
      - name: Checkout code
        uses: actions/checkout@v3

      #2 Set up Python for backend tests
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: 3.11

      #3 Install backend dependencies
      - name: Install Backend Dependencies
        run: |
          cd backend
          python -m venv venv
          source venv/bin/activate
          pip install --upgrade pip
          pip install -r requirements.txt

      #4 Run backend tests
      - name: Run Backend Tests
        run: |
          cd backend
          source venv/bin/activate
          # Example test command (replace with your actual test command)
          python -m pytest tests
```

main.yml workflow file with build, test, and deployment stages

Step 3: Configure GitHub Secrets

If deploying to AWS, add the following secrets in your GitHub repository settings (Settings → Secrets and variables → Actions):

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `EC2_INSTANCE_IP`
- `SSH_PRIVATE_KEY`

Step 4: Commit and Push Workflow

Add the workflow file to your repository and push to main branch:

```
git add .github/workflows/main.yml
```

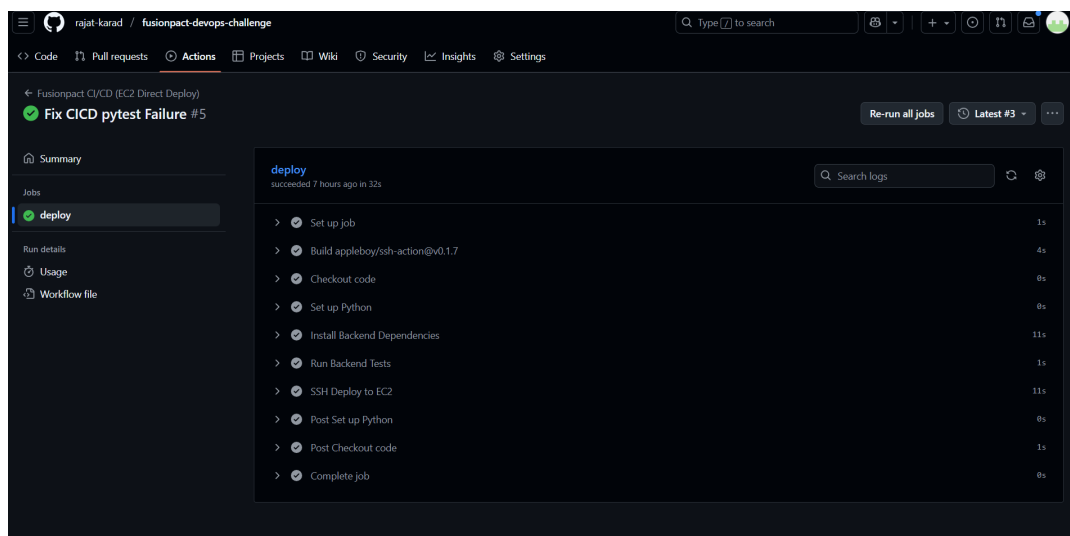
```
git commit -m "Add CI/CD workflow"
```

```
git push origin main
```

Step 5: Verify CI/CD Pipeline Execution

Navigate to your GitHub repository and verify the workflow runs successfully:

- Go to Actions tab in your GitHub repository
- Observe the workflow run for your latest commit
- Verify all jobs complete without errors



Workflow build completed without errors, all jobs passed

Data Persistence with SQLite

Problem Statement

The initial implementation of the full-stack application used file-based storage where user data was saved in temporary JSON files within the Docker container. This approach had a critical flaw: whenever containers

were restarted or rebuilt, all user data would be permanently lost due to the ephemeral nature of container filesystems.

Solution Implementation

I addressed the data persistence problem by replacing the earlier file-based storage with a proper SQLite database managed through SQLAlchemy. Key implementation steps included:

- Replaced JSON file-based storage with SQLite database for structured data management
- Created a dedicated `backend/db` folder on the host machine and mounted it to `/app/db` inside the backend container via `docker-compose.yml`
- Ensured the SQLite file (`users.db`) remains persistent across container restarts
- Updated core backend files — specifically `database.py`, `models.py`, and `services.py` — to integrate SQLite using SQLAlchemy ORM
- Modified the database connection string to use an absolute path (`sqlite:///app/db/users.db`)
- Changed all CRUD operations to use SQLAlchemy instead of direct file I/O
- Installed the required `pydantic[email]` dependency for email validation
- Added startup commands to create and grant permissions to the `/app/db` folder automatically

This approach effectively solved the issue of data loss after container restarts by ensuring the database file lives outside the container's ephemeral filesystem while also making the backend more robust, secure, and scalable for future improvements.

Verification

I verified data persistence by first creating a dedicated directory on the host (`backend/db`) to store the SQLite database and giving it full permissions with `chmod 777 backend/db`. I then ensured that this directory was mounted as a Docker volume in the `docker-compose.yml` so that the backend container would read from and write to it. After starting the containers with `docker-compose up -d`, I added test data by sending requests to the backend API endpoints—using tools like `curl` to create new user entries. Then, I restarted the Docker containers using `docker-compose down` followed by `docker-compose up -d`. Upon checking the database again, I confirmed that the previously added data was still present, proving that the data persisted independently of the container lifecycle.

Error Resolution

During the deployment process, several critical errors were encountered and systematically resolved:

Error 1: cadvisor Regex Mismatch

Error Message: "cadvisor does not match any of the regexes: `'x_'`"

Solution: Updated `docker-compose.yml` to correct the version specification and fixed YAML syntax according to Docker Compose standards.

Error 2: ContainerConfig Error

Error Message: "ContainerConfig error"

Solution: Rebuilt all images without cache to ensure a clean build environment

```
docker-compose down
```

```
docker-compose build --no-cache
```

```
docker-compose up -d
```

Error 3: No Space Left on Device

Error Message: "No space left on device"

Solution: Cleaned up all unused Docker resources including dangling images, stopped containers, and orphaned volumes:

```
docker system prune -a --volumes -f
```

Final Verification

Container Verification

Verified all containers running successfully:

```
docker ps
```

System Restart Testing

To ensure complete reliability, restarted the Docker daemon and verified all services come back online with data intact:

```
sudo systemctl restart docker
```

```
docker-compose up -d
```

Troubleshooting

Windows SSH Key Permissions Issue

If you encounter permission issues with your SSH key on Windows:

```
icacls "E:\Downloads\keypair.pem" /inheritance:r
```

```
icacls "E:\Downloads\keypair.pem" /grant:r "%username%:R"
```

Container Configuration Errors

If containers fail to start or have configuration issues:

```
docker-compose down
```

```
docker system prune -f
```

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