Universidad Distrital Francisco Jose de Caldas

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Workshop No. 4 — Application Design and UI Progress

Anderson Jefrey López Jiménez - 20162020424
Juan Esteban Oviedo Sandoval - 20192020064
Andrés Felipe Mateus Saavedra - 20201020119

Software Engineering Seminar
School of Engineering
Bogotá DC. October 2025

1 Dockerfiles and Containerization

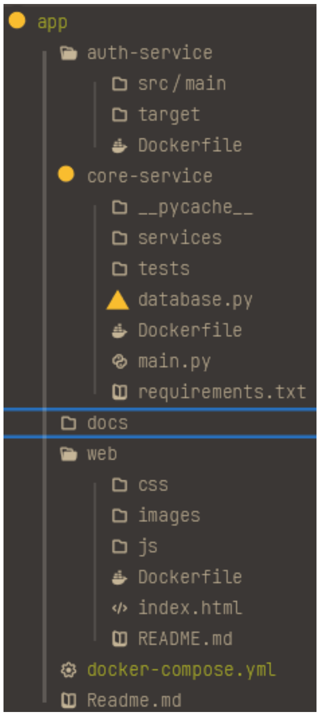
This project includes three independent components: a Java authentication service, a Python core service, and a web frontend built with HTML5. To containerize the entire application, each service is packaged into its own Docker image and orchestrated through a central `docker-compose.yml` file.

1.1 Folder Structure

The application is located in:

`Docs/Workshops/Workshop-4/app/`

The relevant structure of the project is:



```
app
    auth-service
        src/main
        target
        Dockerfile
    core-service
        __pycache__
        services
        tests
        database.py
        Dockerfile
        main.py
        requirements.txt
    docs
    web
        css
        images
        js
        Dockerfile
        index.html
        README.md
    docker-compose.yml
    Readme.md
```

Each service is isolated inside its own directory, which allows it to be built and executed independently while maintaining clear separation of concerns.

1.2 Dockerfile for the Java Authentication Service

The Java service is built with Maven and produces a JAR file inside the `target/` directory. A multi-stage Dockerfile is used to compile the application and run it using a lightweight JRE image:

```
FROM maven:3.9.6-eclipse-temurin-17 AS build
WORKDIR /app
COPY . .
RUN mvn -q -DskipTests package

FROM eclipse-temurin:17-jre
```

```
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
CMD ["java", "-jar", "app.jar"]
```

1.3 Dockerfile for the Python Core Service

The Python service exposes an API and includes a standard `requirements.txt`. Its Dockerfile installs dependencies and runs the main script:

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000
CMD ["python", "main.py"]
```

1.4 Dockerfile for the Web Frontend

If the frontend consists of static HTML, CSS, and JavaScript files, it can be served using NGINX:

```
FROM nginx:latest
COPY . /usr/share/nginx/html
EXPOSE 80
```

This creates a lightweight web server that serves the static site directly.

1.5 Docker Compose Orchestration

All services are orchestrated using `docker-compose.yml`, which builds and runs each container and connects them through a shared network:

```
version: "3.9"

services:

  auth-service:
    build: ./auth-service
    ports:
      - "8080:8080"
    networks:
      - appnet

  core-service:
    build: ./core-service
    ports:
      - "8000:8000"
    networks:
      - appnet
    depends_on:
      - auth-service

  web:
    build: ./web
    ports:
      - "3000:80"
    networks:
      - appnet
    depends_on:
      - core-service
      - auth-service

networks:
  appnet:
    driver: bridge
```

1.6 Running the Application

To build and run the entire application, execute:

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

This command builds each image, creates the network, and launches all services. The system becomes accessible on the following endpoints:

- Web frontend: `http://localhost:3000`
- Python core service: `http://localhost:8000`
- Java authentication service: `http://localhost:8080`

To stop the system:

```
docker compose down
```

This workflow provides a clean and reproducible environment for running the entire multi-service application.

2 Cucumber Feature Files and Test Results

3 JMeter Test Plans and Results

4 GitHub Actions Workflow