InstantChat

Our final project in the module Containerization Technologies

Loévan LE QUERNEC, Ahmed MAALOUL, Raphael VERON
CDOF2, group 8

Cooperation

- Each of us commited on GitLab
- WhatsApp group for the project
- Meetings in order to be aware of what the others are doing

We divided the work into 3 parts:

frontend-service

Raphael

chat-service

Loévan

user-service

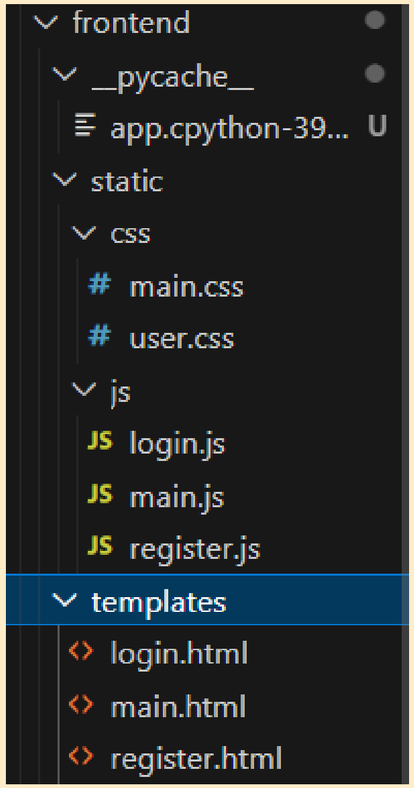
Ahmed

The structure of our project

- project
  - backend
    - chat-service
      - Dockerfile.app | Dockerfile.db
      - app.py | requirements.txt
      - init-db.sql
    - user-service
      - Dockerfile.app | Dockerfile.db
      - app.py | requirements.txt
      - init-db.sql
  - frontend
    - static
    - templates
    - Dockerfile
    - app.py | requirements.txt
  - .gitlab-ci.yml
  - README.md | AUTHORS.md
  - docker-compose.yaml

frontend

Developed with flask.



JavaScript is used to communicate with the server without having to reload the page (via AJAX), thus creating a dynamic, responsive web application.

```dockerfile
# use an official python runtime
FROM python:3.9-slim

# create a group and user to run our app
RUN groupadd -r appgroup && useradd -r -g appgroup appuser

# set the working directory in the container
WORKDIR /app

# copy the dependencies file to the working directory
# and run the command to install the dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# copy the application files to the working directory
COPY app.py /app
COPY templates /app/templates
COPY static /app/static

# expose the port the app runs on
EXPOSE 5000

# define environment variables
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_RUN_PORT=5000

# use HEALTHCHECK to allow Docker to verify the service is running
HEALTHCHECK --interval=30s --timeout=30s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:5000/ || exit 1

# switch to the user
USER appuser

# command to run on container start
CMD ["flask", "run"]
```

Dockerfile

chat-service

Developed with flask.

```dockerfile
# use an official python runtime
FROM python:3.9-slim

# set the working directory in the container
WORKDIR /app

# copy the dependencies file to the working directory
# and run the command to install the dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# copy the 'app.py' file to the working directory
COPY app.py .

# expose the port 5001
EXPOSE 5001

# define the environment variables
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_RUN_PORT=5001

# command to run on container start
CMD ["flask", "run"]
```

Dockerfile.app

- Text file for librairies we need to install using pip
- Expose the port 5001 and set up env variables

```dockerfile
# use an official postgres runtime
FROM postgres:14-alpine

# copy the init-db.sql file to the docker-entrypoint-initdb.d directory
COPY init-db.sql /docker-entrypoint-initdb.d/

# define the environment variables
ENV POSTGRES_DB=postgres-db
ENV POSTGRES_USER=normaluser
ENV POSTGRES_PASSWORD=user
```

Dockerfile.db

- Using simple PostgreSQL database
- Initialization file for the table messages

user-service

Developed with flask.

```dockerfile
# use an official python runtime
FROM python:3.9-slim

# set the working directory in the container
WORKDIR /app

# copy the dependencies file to the working directory
# and run the command to install the dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# copy the 'app.py' file to the working directory
COPY app.py .

# expose the port 5002
EXPOSE 5002

# define the environment variables
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_RUN_PORT=5002

# command to run on container start
CMD ["flask", "run"]
```

Dockerfile.app

```dockerfile
# use an official postgres runtime
FROM postgres:14-alpine

# copy the init-db.sql file to the docker-entrypoint-initdb.d directory
COPY init-db.sql /docker-entrypoint-initdb.d/

# define the environment variables
ENV POSTGRES_DB=postgres-db
ENV POSTGRES_USER=normaluser
ENV POSTGRES_PASSWORD=user
```

Dockerfile.db

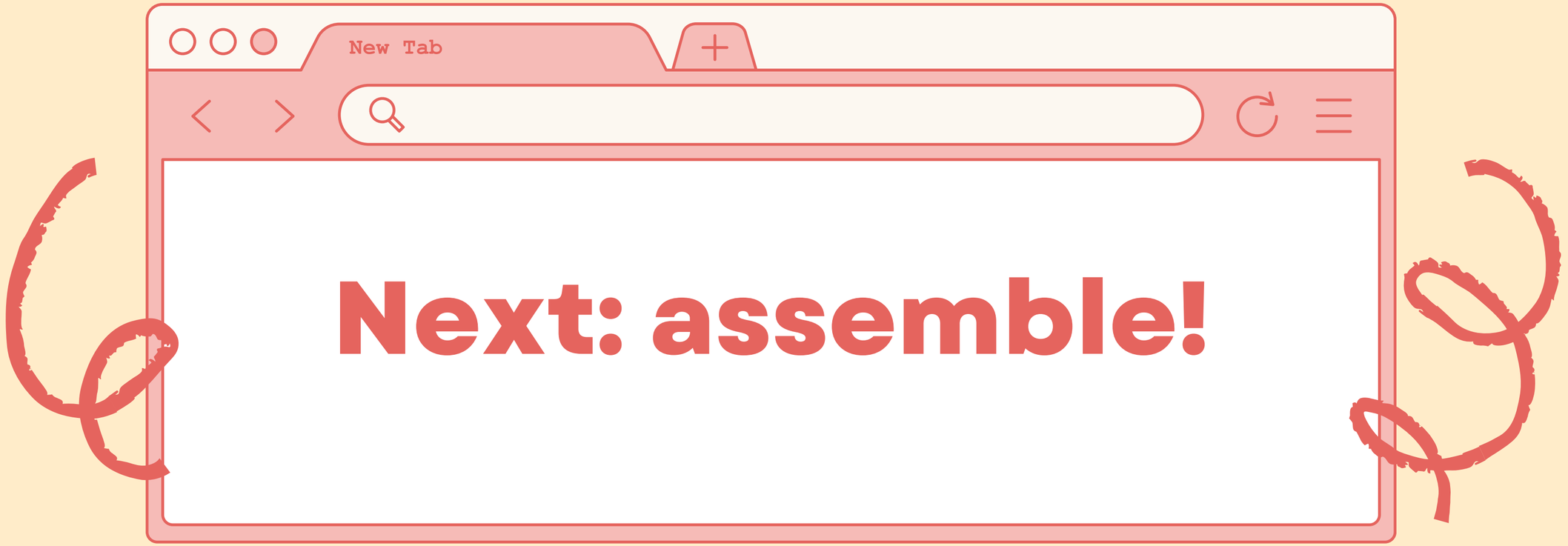
- Hashing Password using Bcrypt
- Implementation of JW Token

user-service

Developed with flask.

- Hashing Password using Bcrypt
- Implementation of JW Token

```python
# create the route /login
@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')

    user = User.query.filter_by(username=username).first()
    if user and bcrypt.checkpw(password.encode('utf-8'), user.password.encode('utf-8')):
        access_token = create_access_token(identity=username)
        return jsonify(access_token=access_token), 200

    return jsonify({"warning": "bad username or password"}), 401
```



Next: assemble!

Services

We build 5 containers, meaning 5 services on our ‘docker-compose.yml’.

Networks

1 network for our front and micro-services and 2 networks between each micro-services and their db.

Volumes

2 volumes for each db and 1 for our front (hot-reloading).

docker-compose

```yaml
version: '3'

services:
  frontend:
    container_name: frontend
    build: ./frontend
    ports:
      - 5000:5000
    volumes:
      - ./frontend:/app
    networks:
      - default

  chat-service:
    container_name: chat-service
    build:
      context: ./backend/chat-service
      dockerfile: Dockerfile.app
    ports:
      - 5001:5001
    depends_on:
      - chat-service-db
    networks:
      - chat-service-network
      - default

  chat-service-db:
    container_name: chat-service-db
    build:
      context: ./backend/chat-service
      dockerfile: Dockerfile.db
    volumes:
      - chat-service-db-data:/var/lib/postgresql/data
    networks:
      - chat-service-network
```

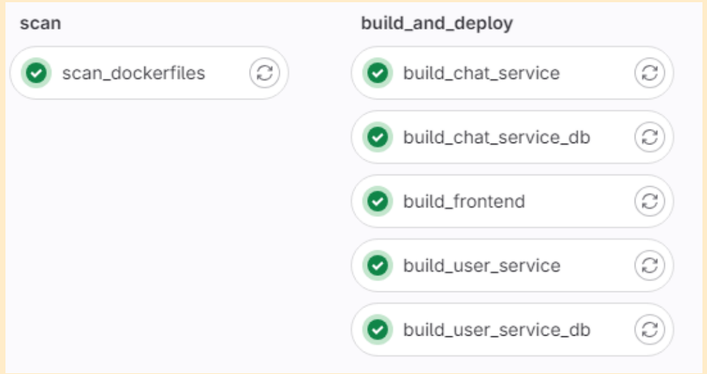
```yaml
  user-service:
    container_name: user-service
    build:
      context: ./backend/user-service
      dockerfile: Dockerfile.app
    ports:
      - 5002:5002
    networks:
      - user-service-network
      - default

  user-service-db:
    container_name: user-service-db
    build:
      context: ./backend/user-service
      dockerfile: Dockerfile.db
    volumes:
      - user-service-db-data:/var/lib/postgresql/data
    networks:
      - user-service-network

volumes:
  frontend:
  chat-service-db-data:
  user-service-db-data:

networks:
  chat-service-network:
    driver: bridge
  user-service-network:
    driver: bridge
  default:
    driver: bridge
```

Technologies

CI/CD

We implemented a pipeline that checks our Dockerfiles with Kics;

Then build and deploy our images on Docker hub.



Data persistence

We implemented 2 volumes for each database.

Hot reload

Our application is reloaded each time we change something in the frontend (thanks to the volume we created).

```yaml
frontend:
  container_name: frontend
  build: ./frontend
  ports:
    - 5000:5000
  volumes:
    - ./frontend:/app
  networks:
    - default
```

Challenges

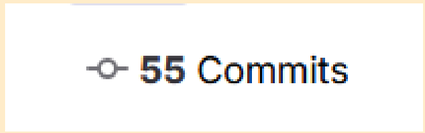
- Assemble

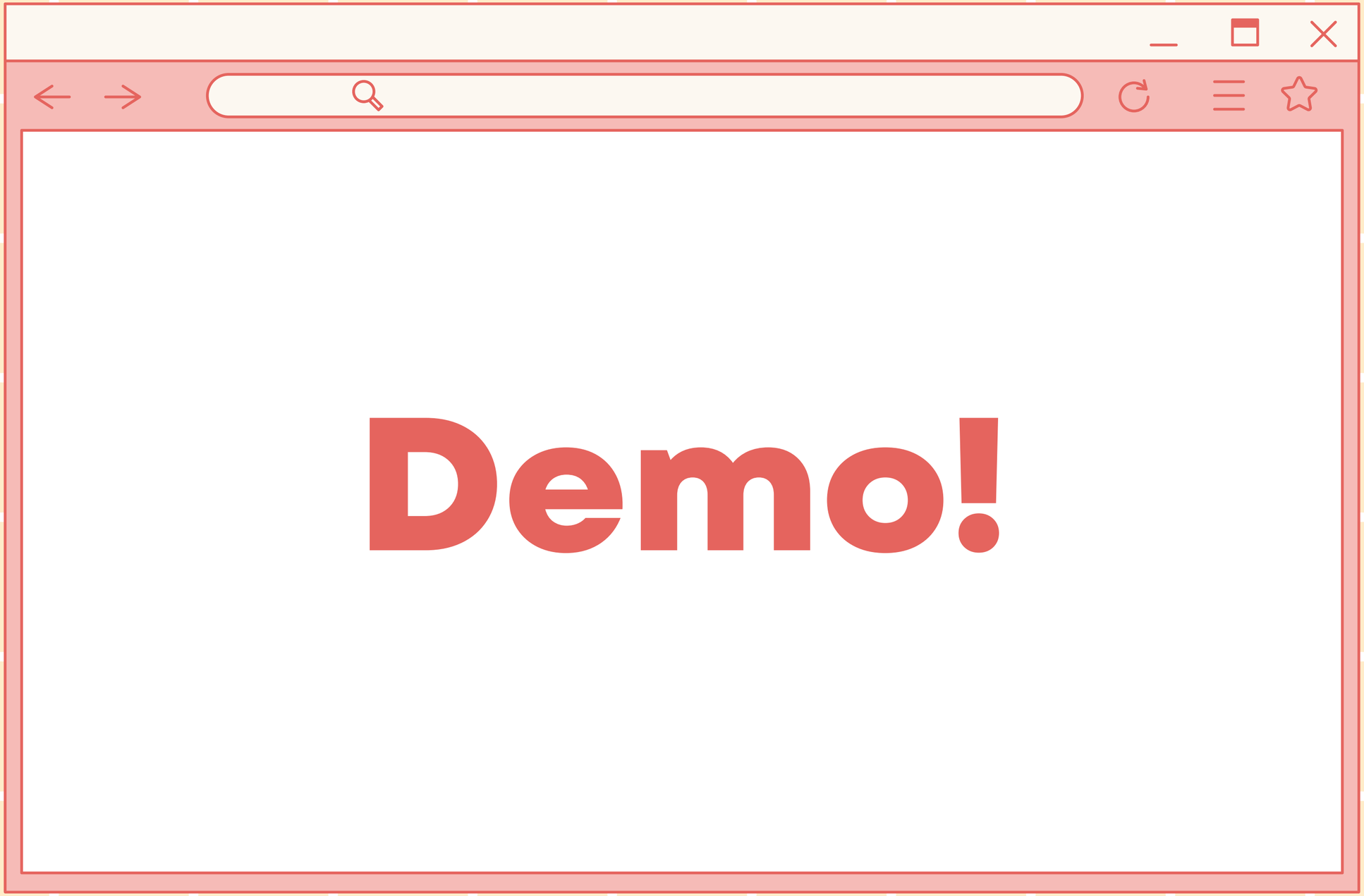
We began to work alone, and each of us has done something that is not fully coherent with the others.

We needed to fully understand what the others have done!

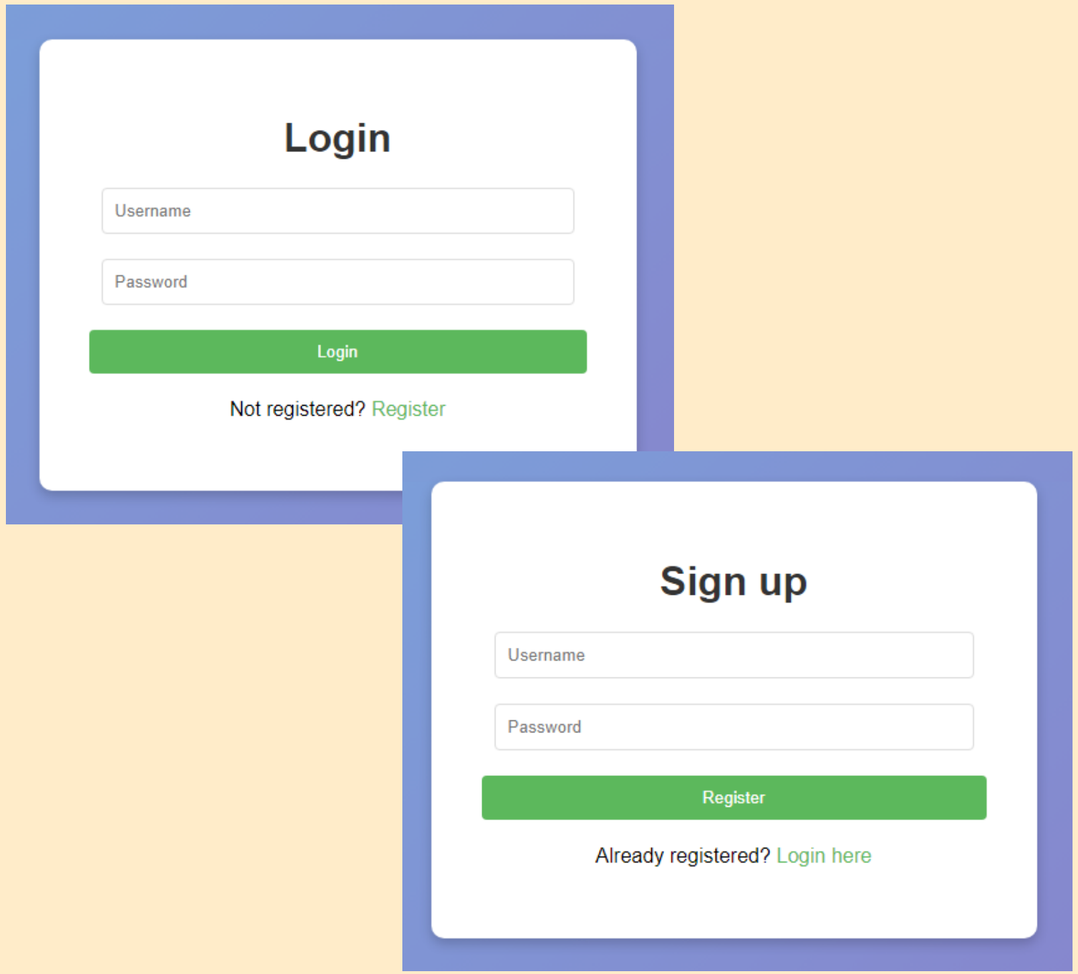
- Implement the CI/CD

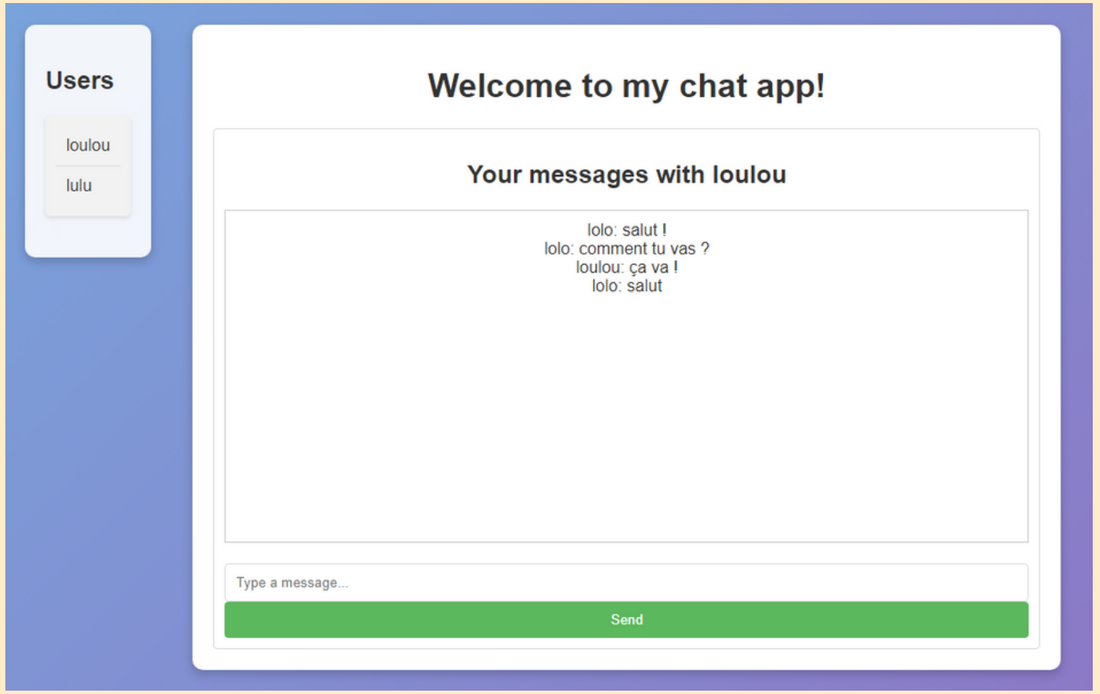
We struggled with kics bugs of its new release

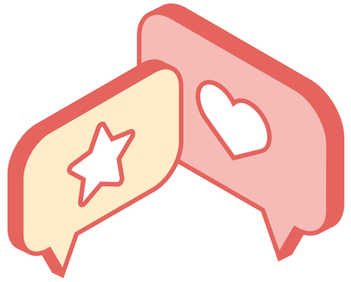



Demo!

If the demo doesn’t work







Thank you!

Feel free to ask some questions!