Microservices Application: Recruitment manager

This repository defines a microservices-based architecture, using Docker Compose to orchestrate various services. The system is designed to be modular, with the intention of easily sclaing it in the future.

Overview of Services

We will shortly summerize the structure of each service. The used database with each service is not mentioned seperatly.

1. **Gateway API (`api_gateway`)**

We have build our own gateway in FASTAPI, that fungates as a caching proxy and an orchestrator when multiple actions need to take place (including the SAGA). We have replicated this gateway threefold (with a nginx loadbalancer in front of it), this way offering redundancy and availability to our customers. Each of these instances has their own Redis cache, providing high speed response times and lower traffic to other called services. This service also implements retry-strategies, with random exponential backoff and implements a circuitbreaker if to much failure occurs in one service.
It also protects other microservices endpoints, by checking if the user is logged in and the user is allowed to perform that action using the auth service (discussed later).

- Structure of the gateway api service
  - `main.py` starts the FASTAPI app, defines the calls for login/signup, adds routers to other services
  - `<servicename>.py` are the endpoints that relay to that service, and adds protection to those private endpoints
  - `caching.py` defines all the caching functions used
  - `retry_circuit_breaker.py` defines logic for combination of threadwide circuitbreaker and retry-strategies
  - `rest_interfaces/<servicename>_interfaces.py` are all the interfaces used in communication with the mentioned service
  - `rabbit.py` is code for defining a RabbitMQ publishing channel, used in `matching.py`

2. **JWT Authentication service (`jwt_auth`)**

The **JWT Authentication Service** manages user registration, login, and token verification. It interacts with the **API Gateway** to secure access to protected resources.

Key Functions:

1. **User Registration (`POST /auth/`)**:
   - User is created by hashing the password with `bcrypt` and storing it in the database.
2. **Login & Token Issuance (`POST /auth/token`)**:

- User provides username and password. If valid, a JWT token is issued, which includes the username and user ID.

3. **Token Verification (`POST /auth/verify-token`)**:
   - The token is sent to the Authentication Service for validation. If valid, user info is returned. If invalid, a `401 Unauthorized` error is returned.

Interaction with API Gateway:

1. **Login**:
   - The API Gateway forwards login credentials to the JWT Authentication Service and returns the JWT token to the client.
2. **Token Validation**:
   - The API Gateway checks the `Authorization` header for a token, verifies it by calling the `/auth/verify-token` endpoint, and grants access if valid.
3. **Caching**:
   - The API Gateway caches user info associated with valid tokens to reduce redundant verification requests.

This system ensures secure, token-based user authentication for protected routes.

3. **Profile Management Service (`profile_management`)**

This service is responsible for CRUD actions on the profile of a JobSeeker or a Recruiter. It is designed to follow the union architecture, where there are only inward dependencies. It connects to a mysql database `mysql_profiles`.

- Structure of the Profile Management Service:
  - `main.py` defines all the endpoints and uses the application layer
  - `application_layer` contains most of the logic , using dependency injection to insert the used DBadapters and PublisherAdapters
  - `domain_model` contains the state and minor functionality
  - `interfaces.py` define the interfaces for the adapters (application layer)
  - `rest_interfaces/*` contain the interfaces/ contracts with the gatewayAPI
  - `publisher.py` defines the publisher that implements the publisher adapter, and sends updates of the profile to the Recommendation service

4. **Job Management Service (`job_management_service`)**

This service is responsible for CRUD actions on the jobs for a given Recruiter. It is designed to follow the union architecture, just like the previous service and it almost has the same structure with some different functionalities. It connects to a mysql database `mysql_jobs`.

- Structure of the Jobs Management Service:

- `main.py` defines all the endpoints and uses the application layer
- `application_layer` contains most of the logic , using dependency injection to insert the used DBadapters and PublisherAdapters
- `domain_model` contains the state and minor functionality
- `interfaces.py` define the interfaces for the adapters (application layer)
- `rest_interfaces/*` contain the interfaces/ contracts with the gatewayAPI
- `publisher.py` defines the publisher that implements the publisher adapter, and sends updates of the job to the Recommendation service

7. **Job Service (Future Addition)**

The Job Service will manage job listings, job postings, and related functionalities. It will interact with other services (like profile management) to offer job search and posting features.

- **Dependencies**: To be defined.
- **Environment Variables**: To be defined.

8. **Matching Service**

The Recommendation Service is a microservice designed to efficiently link jobseekers with jobs. It analyses incoming data and performs advanced searches to identify the best matches using ElasticSearch. The results are stored in a relational database for future use and retrieval.
The service listens to two RabbitMQ queues:

1. **job_update**: Handles new or updated job information.
2. **jobseeker_update**: Handles new or updated jobseeker information.

Jobseeker Update Processing Flow

1. **Receive Message**: A `jobseeker_update` message (in JSON format) is consumed from RabbitMQ.
2. **JSON to Java Object**: The JSON structure is deserialized into a Java object representing the jobseeker.
3. **ElasticSearch Update**: The jobseeker data is inserted/updated in the ElasticSearch database.
4. **Find Matches**:
    - Searches the ElasticSearch database for matching jobs based on predefined criteria.
    - Leverages ElasticSearch's advanced query capabilities for efficient and fast results.
5. **Process Matches**:
    - Store new matches in the MySQL database.

The Processing Flow for job updates is equivalent.

- Structure of the Recommendation Service
    - `main.java` Sets up and configures the necessary components.
    - `Job.java, JobSeeker.java, Salary.java` are the Java classes that correspond to these entities and are deserialized from JSON messages received from the message broker.
    - `ElasticsearchConnector.java` sets up the connection to the Elasticsearch database using the right credentials

- `ElasticDB.java` Provides methods for inserting and retrieving jobseekers and jobs into the database. Executes search queries to match jobseekers with relevant jobs based on their profiles.
- `MatchingDB.java` sets up the connection to the *mysql_matching* database. It contains a method to insert recommendation matches.
- `Match.java` is a simple class to store a recommendation match.
- `RabbitClient.java` sets up the connection to RabbitMQ and starts consuming from two seperate queues. It contains the callback functions to process a new message.

9. **Recommendation Service (Future Addition)**

- **Dependencies**: To be defined.
- **Environment Variables**: To be defined.

Network Configuration

All services are connected via a private Docker network (`private-network`) to ensure secure communication between containers.

Volumes

- **msql_auth_data**: Persists data for the MySQL authentication database.
- **msql_profiles_data**: Persists data for the MySQL profiles database.

How to Run the Application

Prerequisites

- Ensure that Docker and Docker Compose are installed on your system.

Steps to Start

1. Clone the repository to your local machine.
2. Navigate to the project directory.
3. Build and start the services using Docker Compose:

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

How to Test the Application

Prerequisites

- Ensure that `newman` is installed on your system

Steps to start

1. Populate the application with random Jobseekers and Recruiters

First, the application needs some mock data so you'll be able to swipe and match in the following steps. The data is added to the application with a bash script that calls 2 postman scripts:

- `add-jobseeker.json`: creates 50 jobseekers
- `add-recruiters-and-create-jobs`: creates 30 recruiters, where each of them creates 5 jobs

A third json-file is included to ensure the environment variables used in the scripts can be found.
The Postman scripts execute HTTP-requests to the api_gateway. In a later step, you'll be using the same endpoints to create your own profile.

The script is executed by running the following commands in the root directory of the project (this shouldn't take more than 3 minutes):

```
chmod +x run_scenario.sh
./run_scenario.sh
```

In the terminal, you should see 3 calls for the creation of each Jobseeker:

- `sign-up`: creates a username and password
- `login`: login using the newly created username and password
- `create profile`: fills in personal details of the Jobseeker

When the creation of the Jobseekers finishes, the scripts starts doing 8 calls for each Recruiter:

- `sign-up`: creates a username and password
- `login`: login using the newly created username and password
- `create profile`: fills in personal details of the Recruiter
- 5x `create job`: creates 5 jobs with their own random details

When the script finishes, it's time to move on to the next step.

2. Create your own profile as a Jobseeker

For this step, you are going to manually sign-up, login and create a profile as a Jobseeker.
This can easily be done using the UI (http://localhost:3000).