Description of the software and its Functionality

Nowadays social media has an increasingly significant influence on users 'lives.
People often lose a sense of individuality and tend to imitate those who are perceived as successful. People is losing their humanity and just try to copy who have his success in life.

In the social media the users are often influenced primarily on the physical appearance and popularity. Users frequently make connections based on superficial criteria than genuine personal affinity.

Letmeknow app is a social media application that allows users to showcase their personal characteristics on their homepage. Users can also gain insights into their own personality through structured forms which defines their profile parameters. User with the app can also discover and connect with others based on their affinity or what their needs rather than appearance or popularity and this is why the identity of the users is anonymous. This Is not a meeting app but is an app for get the opportunity to know new people from a different way.

Documentation[Italian] for a full introduction and past deployment :
Italian Documentation

Brief introduction based on the frontend features of the actual microservice version:

- **Create User**

  Users can create a profile using a unique email address which is restricted to a single account. After this phase the user is assigned to a user id.

- **Login Page**
  The user can access their accounts by their **user ID** or the correlated email and **Password**. Passwords are stored in the database in a encrypted format for security.

- **Home Page**
  Displays the user's **current parameters**, which should be calculated from the answer to the forms.

- **Edit User**

  Enable the user to edit his own parameters, in the original app this is not available because the user can not set his own parameters.

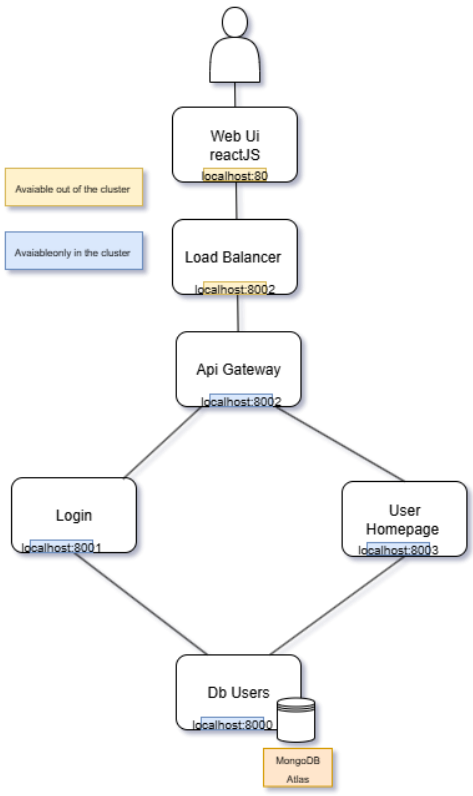
The frontend and the backend are build for get more confident on my knowledge and for use real instrument used in the nowadays projects. Backend in fastapi and frontend in react.
Both of them integrated in the JetBrains software, PyCharm for python and IntelliJ for react.

Link to the repository in github:
https://github.com/FrancescoAstolfiDev/CloudComputingAssignment_v1.git

A software architecture design of your application

Letmeknow is implemented as a set of small, stateless microservices deployed in a container orchestration platform in Kubernetes. The user access through the frontend that interacts with the backend through an API Gateway. Persistent data (user profiles and form responses) is stored in a managed MongoDB (MongoDB Atlas). The platform applies cloud patterns for resilience, security, and scalability.

Architecture draw in drawio



Description of the components:

- **Web Ui Provide** modern, responsive pages for Create User, Login, Home, Edit Profile and forms that compute user parameters. Call the API Gateway for all backend operations. It has an inner state with redux, used for now just for store the logged user.
- **Load Balancer** External ingress that distributes the incoming HTTP traffic from the client to the cluster. Route the traffic to the Gateway-Service.
- **Api Gateway** Single stable entry point that distributes incoming HTTP requests to the traffic.
- **Login** has the role of login the user into the application, checking the match of the password. Enable the creation of new users and guarantee correct validation from the outside data and creation of the userid.
- **Homepage** Shows the params to the users and give the user the opportunity to change the params. Validate the data from the outside.
- **DB Users** Persistent storage for the user profiles, host at the moment the only collection of the users and provide high availability and a global distribution with the supply of mongodb atlas

Pattern used:

- **Stateless services:** Backend microservices are stateless; state is in MongoDB or caches. This enables horizontal scaling and rolling updates.
- **API Gateway pattern:** Centralizes auth, routing presents a stable API to the frontend.
- **Load balancing:** Cloud LB at edge + k8s service routing balances load across replicas.
- **Service discovery (k8s ClusterIP):** Internal services discover each other via Kubernetes DNS and Services—no hard-coded IPs.

Benefits and Challenges of the Architecture Design

Benefits

- **Security by design** Each microservice communicates only through defined APIs, which limits exposure. Sensitive data (such as passwords) is encrypted in transit (TLS) and at rest (database encryption). Since every request is scoped to a small set of data, no single service has broad, unrestricted access, reducing the attack surface.
- **Scalability and resilience** By adopting a microservices architecture with container orchestration Kubernetes, the system can easily scale horizontally. Services can be replicated independently based on demand, and failures in one service do not necessarily impact others.
- **Ease of maintenance and modularity** Developers are encouraged to write stateless, independent services. This separation improves maintainability, as updates or bug fixes can be deployed to a single service without requiring a full system restart.
- **Cloud-native deployment** Using YAML configuration files for orchestration enables repeatable and consistent deployments. Infrastructure as Code (IaC) ensures reproducibility and simplifies environment setup.
- **Data protection:** All sensitive data (passwords, tokens) is encrypted with strong hashing (bcrypt/Argon2) and transmitted over TLS.
- **Service isolation:** Each microservice has least-privilege access to the database to prevent data leaks.

Challenges and Mitigation

- **Database access complexity** Each service must go through its own API layer to interact with the database. This adds complexity when simple read/write operations are needed.
- **Code replication across services** For maintain independence, similar logic same code is often repeated across multiple services.
- **Service-to-service communication overhead** Handling responses between APIs adds latency and complexity and sometimes it is hard to relocate where there is a problem in communication. The use of the API Gateway helps centralize the communication between the service and the frontend.
- **Orchestration configuration complexity** Kubernetes YAML manifests and orchestration rules are complex to manage. When I started to structure YAML files by service and environment + in single files in the same directory I found a more safe and easy way for handle the complexity of the application.
- **Consistency between code and container images** Sometimes container images fall out of sync with the code, requiring manual rebuilds and testing (e.g., via PowerShell scripts and Postman).