

Distributed Systems
Documentation

AI-Health-Coach

Semester: Spring 2023



Version: 1.0

Date: 2023-05-22 15:48:53+02:00

Project Team: Dario Berther
Fabrice Bosshard

Project Advisor: Thomas Bocek

Contents

1	Requirements	2
1.1	Non Functional Requirements	2
1.2	Functional Requirements	2
2	Architecture	3
2.1	Docker Container	3
2.2	Traefik Reverse Proxy	3
2.3	Blazor Website	4
2.4	Postgres Database	4
3	Configuration	5
3.1	Build and Release	5
3.1.1	Blazor Dockerfile	5
3.2	Startup	5
3.3	Sticky Cookie	5
3.4	HTTP/3	5
4	Future Features	6
4.1	Software	6
4.2	Security	6
4.3	Scalability	6

1 Requirements

1.1 Non Functional Requirements

- Load balancing with scalable service
- Failover of a service instance
- Dockerized and Frontend connects via HTTP/3
- Persistent storage
- Use latest stable releases of chosen libraries and frameworks

1.2 Functional Requirements

- The user wants to create routines for workouts and meals
- The user wants to get a weeks worth of exercise or meals in a single routine
- The user wants be able to access past routines
- The user wants to be able to personalize his account with various parameters like height, weight, age etc.
- The user wants to specify his preferences (kind of workouts and not liked food) to get personalized routines

2 Architecture

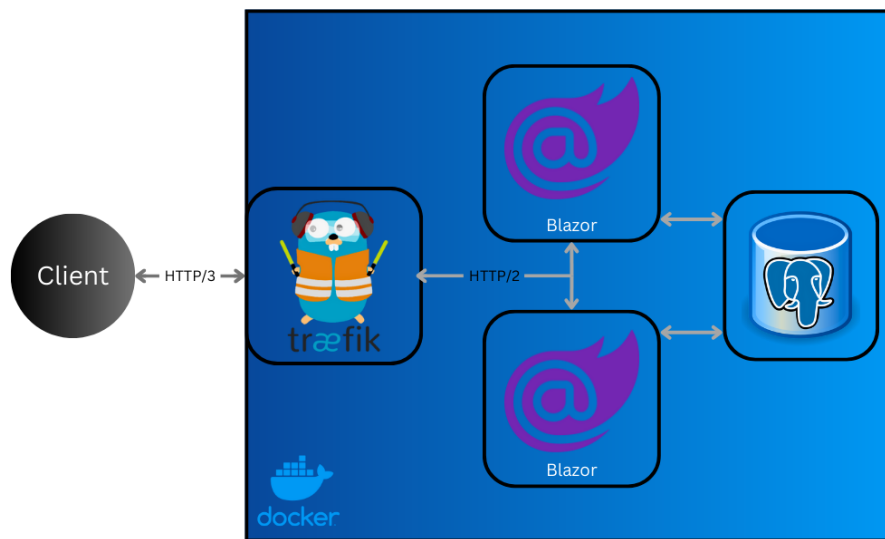


Figure 1: Overview of the structure of the AI Health Coach application

2.1 Docker Container

All of the components of the program are housed within Docker containers, and they all communicate with one another across the internal Docker network. Benefits of Docker:

- **Development:** Any developer can launch the program, which enables simple local development independent of a server.
- **Isolation:** The configuration of each container is unique, and there are no dependencies on other containers, which makes management and updates easier.
- **Scalability:** The Blazor server's instances can be increased at will and with no effort.
- **Consistency:** Consistency across the units is guaranteed because both Blazor servers are built on the same image and share the same codebase.
- **Portability:** The application is hardware independent because it can operate on any platform that supports Docker.

2.2 Traefik Reverse Proxy

The Traefik reverse proxy and load balancer controls the traffic between the client and the services. All service instances in this operation share the load, and traffic is diverted if one fails. Why Traefik:

- **Integration with Docker:** Since Traefik interacts directly with the Docker API, it is able to detect when a service has been terminated.
- **Load Balancing and Sticky Sessions:** Supports persistent sessions, which is crucial for managing websockets and Blazor server applications, in addition to load balancing.
- **Scalability:** Traefik's dynamic configuration supports scaling of Docker containers. It can be configured with more containers without having to restart it.
- **Automatic SSL/TLS configuration:** Traefik can work with ACME servers. The automatic request and integration of certificates is very advantageous in the production environment.

2.3 Blazor Website

In order to create web apps in C#, Microsoft provides the Blazor framework. This means both the web application itself and the server-side logic can be written in the same language. Why Blazor:

- **C# as a language:** We don't need to learn any new languages for this project because we are comfortable with C#. Furthermore, IDEs like Visual Studio and JetBrains Rider offer good Blazor integration and development assistance.
- **.NET ecosystem:** It is possible to access the .NET libraries, tools, and features. For example Entity Framework is utilized for the connection between the application and database. Additional application development is made simpler by other frameworks like Identity for authentication.
- **No JavaScript:** Blazor enables the creation of interactive user interfaces without JavaScript, much like React or Angular.
- **Reusable components:** Blazor allows for the division of the UI into separate, reusable components. As a result, individual UI building blocks can be developed and tested independently.
- **Blazor Models:** Blazor offers two different hosting models: Blazor Server and Blazor WebAssembly. Blazor Server runs code on the server and updates the UI with help of a SignalR connection. This is advantageous in our case, as failover can be ensured more easily. With Blazor WebAssembly, all code is executed directly in the browser. If an instance would shut down, all code has to be downloaded again from another server.

2.4 Postgres Database

Postgres is an open-source database and offers high reliability and data integrity. Why Postgres:

- **Open-Source:** This database can be freely incorporated into our program.
- **ACID Compliance:** Due to its compliance with ACID (Atomicity, Consistency, Isolation, Durability), Postgres offers data reliability and consistency even in the case of mistakes. These characteristics are crucial since application crashes and faults (in the sense of a service shutdown) are to be expected.
- **.NET Integration:** Postgres supports Entity Framework for database access, which makes it very simple to update the application's content.

MySQL or Microsoft SQL would have been possible alternatives, but we chose Postgres because it is open-source and already familiar to us.

3 Configuration

3.1 Build and Release

A single Docker Compose file is used to start the whole application stack. It is responsible for:

- Starting Traefik with the necessary parameters
- Loading the database files from the disk and starting it
- Constructing the Blazor application using a unique Dockerfile

3.1.1 Blazor Dockerfile

The Blazor application's Dockerfile uses a multi-stage construction procedure. This approach produces an image that is as lightweight as possible, which improves the amount of resources needed by the docker container. This is accomplished by separating the build step, in which all required dependencies and build tools are included, from the final runtime stage, in which the produced program is transferred into a ASP.NET environment free of dependencies and build tools.

3.2 Startup

We use a makefile and run the command `make start i=n` to start the application. The number of concurrent service instances that should be started is indicated by the variable `i` in this case. Blazor will raise an error if there are more than 128 occurrences, despite the fact that `n` can be any integer value. The load balancing in Traefik is made to automatically detect the number of services and change as necessary.

3.3 Sticky Cookie

To make sure that the client connects with the same server during a session, the Blazor Services use a sticky cookie. This is necessary because Blazor connects to the client via WebSockets through a SignalR connection.

SignalR maintains a stateful connection between the client and server, and if the client was redirected to a different server with each request (or upon reloading), the state information could be lost or mismatched, resulting in application problems or unexpected behavior. When a service fails and is shut down, the whole SignalR connection is re-established and no state is maintained (outside the database).

3.4 HTTP/3

It is important to switch on the experimental feature in Traefik's setup in order to enable HTTP/3 communication between Traefik and the client. Additionally, the endpoint for which HTTP/3 should be activated must be specified.

It employs the QUIC protocol, which runs using UDP rather than TCP. As a result, port 443 needs to be expressly opened for UDP transmission. However, we also had to make sure that port 443 is kept open for TCP traffic in order to guarantee backwards compatibility.

As a result, we explicitly open port 443 for both TCP and UDP.

4 Future Features

Possible improvements for the future include software, security and scalability improvements:

4.1 Software

- **User Login:** The system as it is at the moment must be self-hosted, because there is no way to detect, differentiate and securely login users. As mentioned above, the .NET Identity framework could be utilised for that.
- **Newer AI models:** With the current hype for AI, the underlying models are evolving rapidly and we could utilise the better consistency to give the user an even better experience and more possibilities. GPT-4 for example can accept up to 24000 words in english, which creates much more room for personalized requests by users.
- **Better filters:** At the moment the user has to define his preferences via text, but in the future many of those choices could be relocated into clickable UI-elements with different predefined options. For example in the meal planner there could already be predefined food items a user does not want to have included, based on past inputs.
- **Improved UI:** This project was more focused on the networking and distributed system part, than the application itself. That had impact on the UI and UX, which could be improved in future updates.

4.2 Security

- **Certificates:** At the moment only self-signed certificates are used, which is not feasible for a real world application. For that the Traefik should utilise real certificates with automatic integration via ACME server to ensure security for the user.
- **Traefik Dashboard:** Use authentication to access the Traefik dashboard.
- **Remove SSL Termination:** It would be desirable to remove SSL termination at the load balancer and maintaining encryption all the way to the Blazor Server. By ensuring end-to-end encryption, it lowers the danger of data leakage within the internal network. This is especially important as the application deals with private user information.

4.3 Scalability

- **Database:** Although a single database may manage numerous service instances, it becomes increasingly important to guarantee speed and reliability as the quantity of services grow. To boost performance, availability, and fault tolerance, it will be important to implement a distributed database system. A distributed database can also offer redundancy to prevent total system failure.