Hungerger

Architecture Notebook

| **REVISIONS** | | | |
| --- | --- | --- | --- |
| **Rev. No** | **Description** | **Date** | **Person** |
| 0.1 | Creating the document | 18.11.2023 | Aslı |
| 0.2 | The first template was finished | 23.11.2023 | Abdullah, Aslı, Merve, Tarık |
| 0.3 | The appearance was edited  Item 9 was added | 25.11.2023 | Aslı |

# Purpose

This document describes the philosophy, decisions, constraints, justifications, significant elements, and any other overarching aspects of the Hungerger Application that shape the design and implementation.

The system's design is guided by principles of simplicity, modularity, scalability, reliability, and security. Key design decisions include a modal view architecture, API-driven design, and test-driven development. The system is subject to performance, security, scalability, and maintenance constraints. The design choices are justified by alignment with business needs, technical feasibility, resource availability, and future growth. Significant elements include API gateway, testing strategy, deployment platform, and CI/CD practices.

# Architectural goals and philosophy

The proposed system will have a main server that is expected to serve all users. However, in case the number of requests exceeds the capabilities of that main server or in the event of the main server being out of service, the system will rely on a backup server connected to the main server through a load balancer.

The main goal of this architecture is to serve the early adopters of the application while satisfying the following performance criteria:

* The system shall be able to react to every request under 1 second.
* The system shall be able to react to 99% of the requests at all times.
* The system should be able to serve 30000 concurrent users.

# Assumptions and dependencies

**3.1. Technical Assumptions:**

* Using a main server accompanied by a backup server and a load balancer should be sufficient to serve the users with the stated functionality of the app.
* Python, React, and MySQL will be utilized in this project to fit the skills of the product and development team.
* The images will be saved on a cloud storage service to preserve the quality of the content uploaded.
* Ingredient prices will be retrieved from a third-party marketplace service.
* The system will be deployed on GitHub pages.
* Bcrypt will be used to hash users' passwords.
* The system will be supported on Chrome, Edge, Mozilla, Opera, and Safari at early stages. After that, it will support IOS and Android mobile platforms.

**3.1.1. User Behavior Assumptions:**

* Outline assumptions regarding user behavior, preferences, or interactions within the application.
* For instance, assumptions about user engagement, frequency of content sharing, or preferred devices for accessing the platform.
* Users are expected to use mobile and personal PC devices to access the app.
* At the early stages of the project and due to the nature of the platform\*, we are expecting a low frequency of content sharing.
* We are expecting a moderate level of interactivity with the recipes. The interactivity includes rating and sharing recipes with their friends.

\*Hungerger will allow users to share recipes and pictures of recipes only. Hence, we forecast that the inclination to share recipes will not be as high as sharing personal posts with a wide variety.

**3.1.2. Performance Assumptions:**

Document assumptions related to system performance, such as expected load, concurrency, or response times under certain conditions.

* The system shall be able to react to every request under 1 second.
* The system shall be able to react to 99% of the requests at all times.
* The system is expected to username 30000 concurrent users.
* Under heavy loads, the backup server and the load balancer should kick in to take some of the load off the main server and improve response times.

**3.1.3. Security and Compliance Assumptions:**

* The system will comply with KVKK, GDPR, and HIPAA.
* Bcrypt will be used to hash users' passwords.
* HTTPS will be used to provide confidentiality.
* Load balancer will be used for availability.

**3.2. Dependencies:**

**3.2.1. External APIs and Services:**

* The images will be saved on a cloud storage service to preserve the quality of the content uploaded.
* Ingredient prices will be retrieved from a third-party marketplace service.

**3.2.2. Infrastructure:**

* **Windows server 1:** Will serve as the main server. Windows 2022 will be used.
* **Windows server 2:** Will serve as a backup and will kick in if the capacity of the main server is starting to be overwhelmed. Or in case the main server was down. Windows 2022 will be used.
* **Load balancer:** This will balance the load coming to the main server and will kick in if the capacity of the main server is starting to be overwhelmed. Or in case the main server was down.

**3.2.4. Technologies and Frameworks:**

* **Flask:** Will be used to build the application's backend.
* **React:** For front-end development.
* **MySQL:** Will serve as the application's database..
* **Redis:** Will be used for Caching.

**3.2.5. Libraries:**

* Bcrypt (for hashing passwords)
* Redux & Redux Toolkit (for state management)
* React Router (will be used for navigation)

**3.2.6. Data Dependencies:**

* The system relies on a marketplace as a data source for providing information about ingredient prices. The integration part is still under discussion. Therefore, we will use a database instead of a marketplace in release 1.

# Architecturally significant requirements

Please refer to the [**system-wide requirements**](https://docs.google.com/document/u/0/d/12ZMCwUdPSosGHx5YZefjdZ6rGvgVMqse/edit):

* HGG-AVAILABILITY (3.1)
* HGG-PERFORMANCE(3.2)
* HGG-RELIABILITY(3.6)
* HGG-DEVELOPMENT (3.8)
* HGG-SERVER (3.9)
* HGG-BROWSER(3.10)

# Decisions, constraints, and justifications

* The system will have one main server and one backup server. The backup server and the load balancer will help if the loads are starting to overwhelm the main server.
* The app will use caching through a Redis database. Caching will improve the performance of the app and will decrease the number of unnecessary requests to our single main server.
* The app may depend on external APIs. We will need to store the images and get recipe prices that we do not currently hold.

# Architectural Mechanisms

## Architectural Mechanism 1 - Model-View-Controller (MVC)

The Model-View-Controller approach will be used within the architectural design of the system since it gives each individual within the development team a standard to follow whilst increasing the modularity throughout the development of the project.

## Architectural Mechanism 2 - Load Balancing

The system is considered to have a complimentary server that acts as a backup server which is planned to have resources less or similar to the actual server of the system. The load between the servers will be distributed according to the Weighted Round Robin algorithm. NGINX will be used to perform the load balancing.

## Architectural Mechanism 3 - Caching

The Caching, which will be attained through the utilization of Redis datastores, is considered to be included to increase the performance and scalability of the system.

# Key abstractions

* **Recipe:** Represents a culinary recipe shared on the platform. It enables interactions with rates and comments. Its attributes are name, description, ingredients, cost, dietary type, image, creator, ratings, and comments.
* **User:** Represents individuals registered on the platform. It manages recipes and interactions, follows other users, sends direct messages (DMs) to other users, and engages with content. Its attributes are name, password, avatar, email, username, user type, bio, and followers.
* **Interaction:** Represents engagements between users and recipes. It tracks and manages user interactions with recipes, facilitating user engagement and feedback. Its attributes are comments and ratings.
* **Search:** Represents the system's search and filtering capabilities. It enables users to find and discover recipes based on specified criteria. Its attributes are filters (recipe name, dietary preferences, ingredients, and cost).
* **Admin:** Represents authorized individuals responsible for managing the platform and the Users. It monitors and ensures platform adherence to guidelines usernames reported content, and manages user-related issues. Its attributes include privileges for content moderation, user management, and platform oversight.

# Layers or architectural framework

The architectural pattern chosen for our web application is the Model-View-Controller (MVC) pattern. This pattern clearly separates concerns, dividing the application into three interconnected components.

The model represents the data and business logic of the application. It encapsulates the application's data structure and defines the rules for data manipulation. Classes and components are responsible for data storage, retrieval, and manipulation. This includes database models, business logic components, and data access layers.

View is responsible for presenting the user interface and displaying information to the user. It receives input from users and forwards it to the Controller for processing. User interface components, HTML templates, and other presentation-related elements. Views are designed to be modular and independent, allowing for easy modification and customization.

The controller manages user input, processes it, and updates both the Model and the View accordingly. It acts as an intermediary between the Model and the View, handling data flow and user interactions. Controller components, route usernames, and application logic that govern the interaction between the Model and the View. Controllers interpret user input, trigger appropriate actions in the Model, and update the View.

The MVC pattern ensures consistency and uniformity in our web application by enforcing a clear separation of concerns. Each component (Model, View, and Controller) has a distinct role, making it easier to understand, maintain, and extend the application. This separation also facilitates parallel development, as different teams or developers can work on different components without tightly coupling their code.

**Benefits:**

* **Modularity:** The MVC pattern promotes modularity, allowing for independent development and testing of each component. This makes it easier to scale the application and add new features.
* **Maintainability:** With a clear separation of concerns, changes in one component are less likely to affect others. This enhances maintainability, making updating or extending the application over time simpler.
* **Reusability:** Components within the MVC architecture can be reused in different parts of the application or in other projects, promoting code reuse and efficiency.
* **Testability:** The separation of concerns in MVC makes writing unit tests for each component easier, ensuring that the application is robust and reliable.

By adopting the MVC pattern, our web application benefits from a proven and widely accepted architectural approach, providing a solid foundation for building scalable, maintainable, and well-organized software.

# Architectural views

* **Information view:**
  + For the ER diagram and the relational model of our project, please refer to the following documents:
    - erDiagram.drawio
    - RelationalModel.drawio
* **Process view:**
  + Please refer to CreateRecipe\_SequenceDiagram.drawio for the sequence diagram of UC4: Create Recipe use case.
* **Use case view:** 
  + Our use case diagram can be seen from “generalusecase.drawio”.
  + Their brief descriptions can be reached “from uc\_specification\_Brief.docx.”
  + There are 12 defined use cases for our architecture. The fully-dressed forms of 7 of them can be reached from
    - UC1\_CreateAccount.docx
    - UC2\_EditProfile.docx
    - UC3\_DeleteAccount.docx
    - UC4\_CreateRecipe .docx
    - UC5\_EditRecipe.docx
    - UC6\_DeleteRecipe.docx
    - UC7\_SearchRecipe.docx