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 |
| 0.4 | The revisions were completed according to the feedback 2. | 02.12.2023 | Abdullah, Aslı, Merve, Tarık |

# 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.

The philosophy behind using MVC pattern as our low-level architecture is as follows:

* Separation of Concerns:
  + Each component (Model, View, Controller) has a specific role and responsibility, making the codebase easier to understand and maintain.
  + Changes in one component do not directly affect the others, promoting code modularity and reusability.
* Maintainability:
  + Because of the separation of concerns, it's easier to make changes or updates to one part of the application without affecting the others.
  + Developers can work on different components simultaneously, fostering collaboration.
* Scalability:
  + As the application grows, the modular nature of MVC allows for easier scalability.
  + New features can be added without disrupting existing functionality, provided that the MVC structure is well-maintained.
* Testability:
  + Components can be tested independently, allowing for easier unit testing.
  + Unit tests for the Model, View, and Controller can be written separately to ensure that each component functions correctly.
* Reusability:
  + Models, Views, and Controllers can be reused in different parts of the application or even in other projects, enhancing development efficiency.

By adhering to the MVC pattern, developers can create more maintainable, modular, and scalable applications, which is crucial for building robust and adaptable software systems.

Also, the reasoning behind using a monolithic approach is as follows:

* Simplicity:
  + Monolithic architectures are often simpler to develop and maintain, especially for smaller or less complex applications.
  + The entire codebase is contained in one project, making it easier for developers to understand the application's structure and functionality.
* Ease of Development:
  + With all components residing in the same codebase, development tools and workflows are typically simpler.
  + Developers can work on different parts of the application without dealing with the challenges of distributed systems, inter-service communication, and deployment coordination.
* Easier Debugging and Testing:
  + Debugging and testing in a monolithic architecture can be more straightforward because everything is in one place.
  + Unit testing, integration testing, and debugging can be performed more easily since all components are part of the same application.
* Single Deployment Unit:
  + Deploying a monolithic application involves deploying a single unit, which can simplify the deployment process.
  + There's no need to manage the deployment of multiple independent services, and rollbacks can be more straightforward if issues arise.
* Performance:
  + In some cases, a monolithic architecture may offer better performance compared to distributed architectures, as there is no overhead associated with inter-service communication.
  + Shared memory and direct method calls can lead to faster execution of business logic.
* Resource Utilization:
  + Monolithic applications can be more resource-efficient, especially in scenarios where the entire application runs on a single server or in a tightly integrated environment.
* Initial Development Speed:
  + For smaller teams or projects with relatively simple requirements, a monolithic architecture may allow for faster initial development compared to more complex distributed architectures.

# 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.
* The users are expected to use web browsers which support HTML5 such as Microsoft Edge, Mozilla Firefox or Google Chrome preferably MS Edge, Google Chrome and Safari since these browsers are our main test subjects and they are the most popular targets for modern web development.

\*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:**

* **Ubuntu Server 1:** Will serve as the main server. Ubuntu will be used.
* **Ubuntu 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. Ubuntu 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)
* Flask Templates (for creating views)

**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.

**3.2.7. Hardware Limitations (AWS Cloud Services):**

Since we decided on using AWS Cloud Services we are dependent on Amazon’s decisions and updates. There are various hardware limitations as described below:

* Virtualization Overhead:
  + AWS relies on virtualization to create virtual instances (EC2 instances) that run on physical hardware. While this enables resource isolation and flexibility, there is some overhead associated with virtualization that might impact performance compared to running directly on physical hardware.
* Instance Types and Specifications:
  + AWS offers various instance types with different combinations of CPU, memory, storage, and networking capacities. Each instance type has its limitations, and it's crucial to choose an instance type that aligns with the performance requirements of your application.
* Network Performance:
  + AWS provides different network performance levels for EC2 instances. The network bandwidth and latency can vary based on the instance type. High-performance computing (HPC) workloads, for example, may require instances with enhanced networking capabilities.
* Storage Performance:
  + Storage performance in AWS is influenced by the type of storage used (e.g., Amazon EBS, Amazon S3) and the specific characteristics of the chosen storage solution. For example, different Amazon EBS volume types offer varying levels of performance.
* Region-Specific Hardware Availability:
  + The availability of specific instance types and features can vary across AWS regions. Some cutting-edge instance types or features may not be available in all regions, and users need to consider regional limitations when designing their infrastructure.
* Resource Limits:
  + AWS imposes certain resource limits on various aspects of your account, such as the number of EC2 instances, the amount of storage, the number of Elastic IP addresses, etc. Users need to be aware of these limits and request increases if needed.
* Spot Instance Availability:
  + Spot Instances, which allow users to bid for unused EC2 capacity at potentially lower costs, are subject to availability. There might be times when the desired Spot Instance type is not available, and users need to plan accordingly.
* Data Transfer Costs:
  + While AWS offers high bandwidth for data transfer within the AWS network, there are costs associated with data transfer out of the AWS region or to the internet. Users need to be mindful of these costs, especially for data-intensive applications.
* Scaling Considerations:
  + While AWS provides tools for auto-scaling and managing resources based on demand, rapid scaling or scaling to very high levels might encounter limitations. Users should understand the scalability limits of their chosen AWS services and design accordingly.

# 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 complementary server behind a load balancer which operates with Weighted Round Robin algorithm. The complementary server and the load balancer will help if the loads are starting to overwhelm the main server. This decision is done for increasing the performance and reliability of the architecture.
* 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

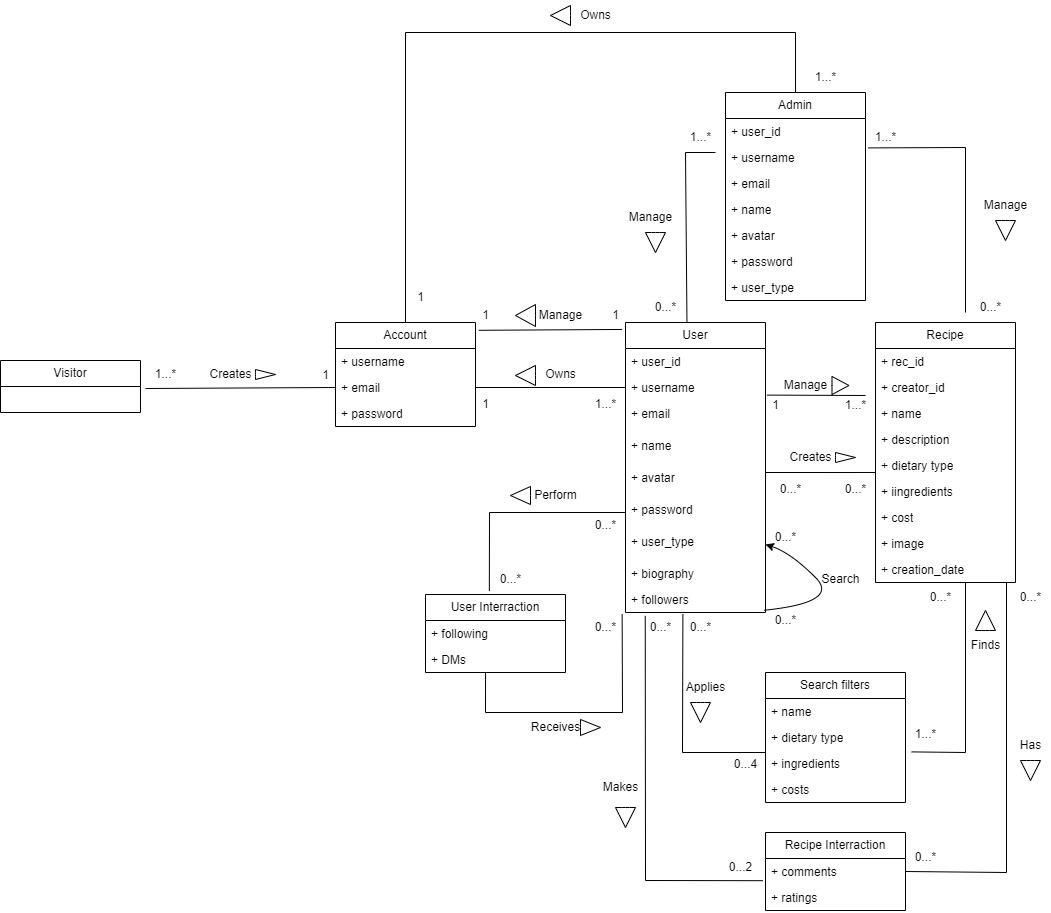
The Caching, which will be attained through the utilization of Redis datastores, is considered to be included to increase the performance and scalability of the system.

## Architectural Mechanism 4 - Security

The system will use hashed passwords and encrypted communication between client and the server via HTTPS method.

# Key abstractions

* **Visitor:** People who visit the website without logging in. They can create an account on the system.
* **User:** People who have an account on the system. They can create recipes, manage (update, delete) recipes, perform interactions (follow other users, send direct messages (DMs) to other users). They can report Users or recipes to the Admin. Their attributes are name, password, avatar, email, username, user type, biography, and followers.
* **Account:** The User account in the application. The account information contains a username, email address, password, and password confirmation.
* **Recipe:** Represents a culinary recipe shared on the platform. It enables interactions with rates and comments. Its attributes are rec\_id, creator, name, description, dietary type, ingredients, cost, image, creation date, ratings, and comments.
* **Interaction:** It represents users' engagement among themselves and with the recipes.
  + **Recipe interaction:** Its attributes are comments and ratings.:
  + **User interaction:** Its attributes are DM’s and followings.
* **Search:** Represent the system's search and filtering capabilities.
  + **Search filters:** It enables users to find and discover recipes based on specified criteria. Its attributes are filters (recipe name, dietary type, 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.

We use Flask framework’s template approach in order to implement the MVC pattern, whilst doing so we are also able to use Flask framework’s SQL manipulation advantages on the model layer.

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.

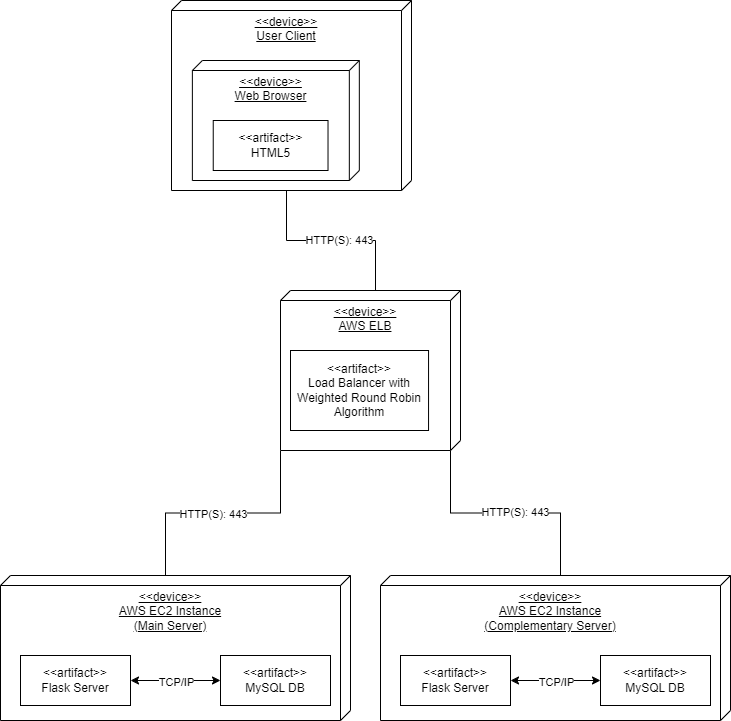
**Benefits:**

* **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.

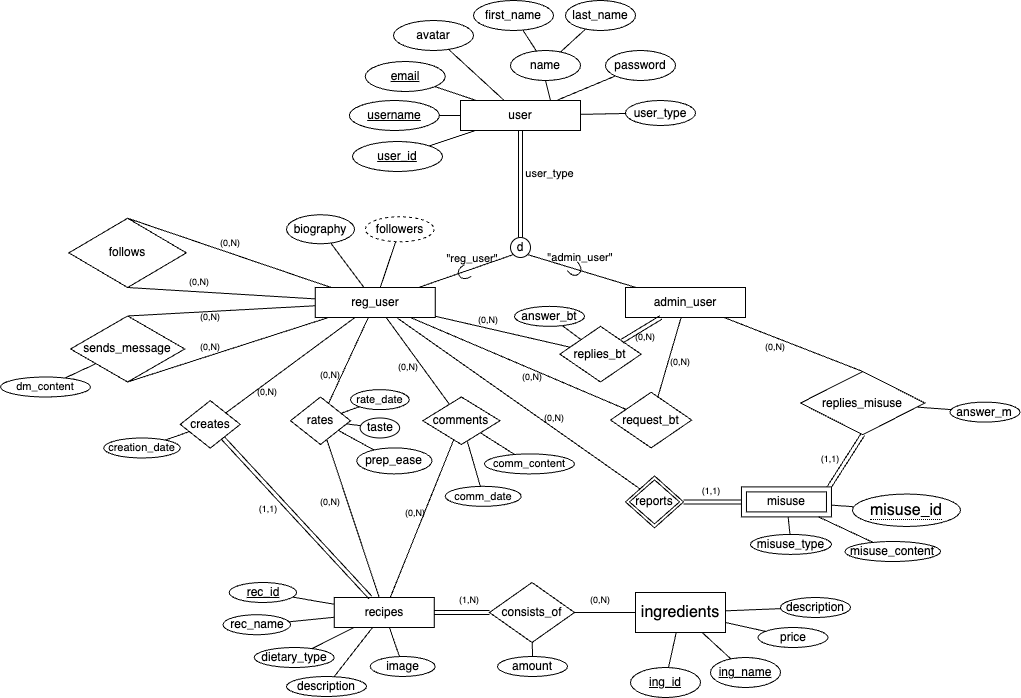
By adopting the MVC pattern, our web application benefits from a proven and widely accepted architectural approach, providing a solid foundation for building scalable, maintainable, and well-organized software.

# Architectural views

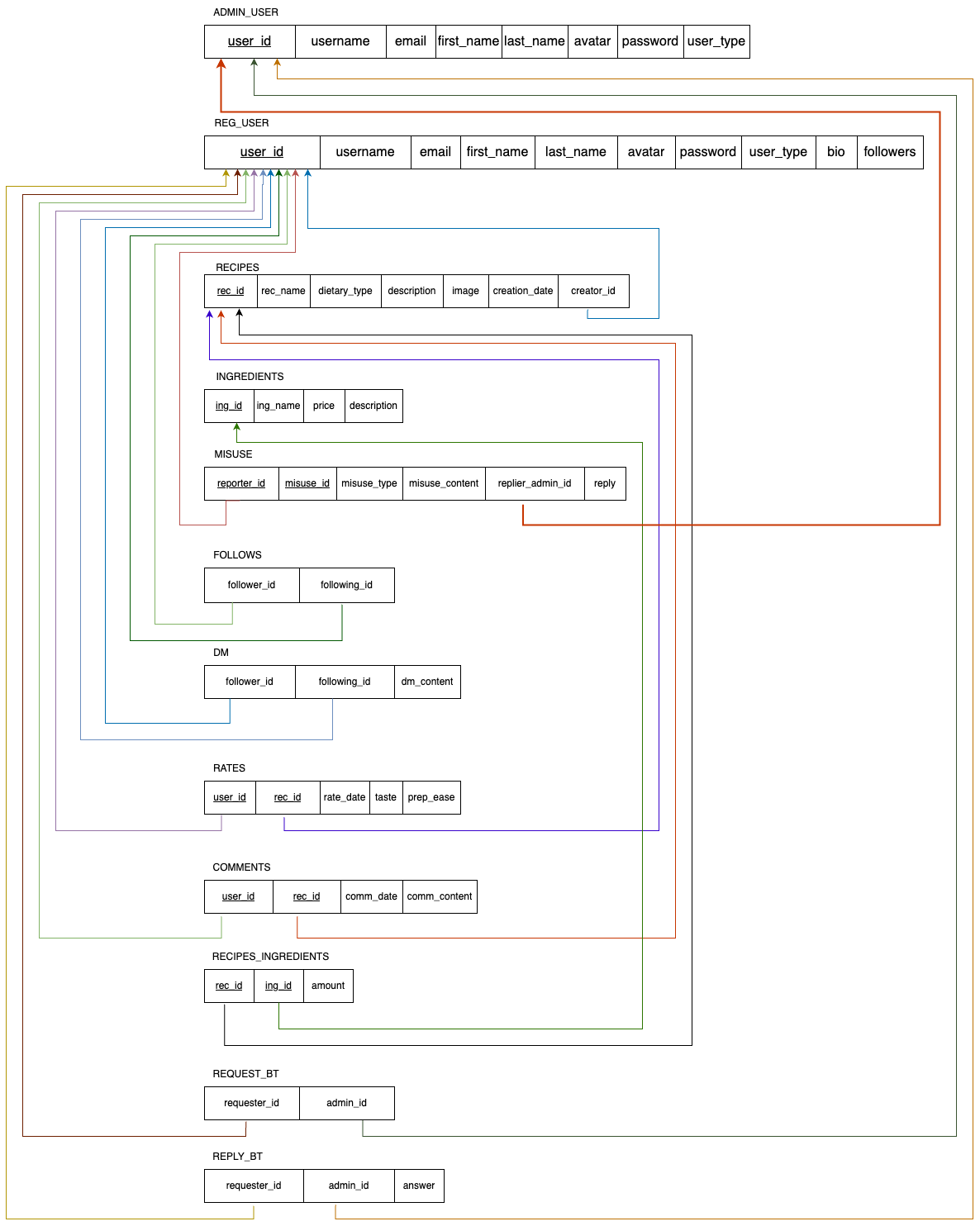
* **Physical view:**
  + The deployment diagram for Hungerger System is given below:
    - Deployment Diagram



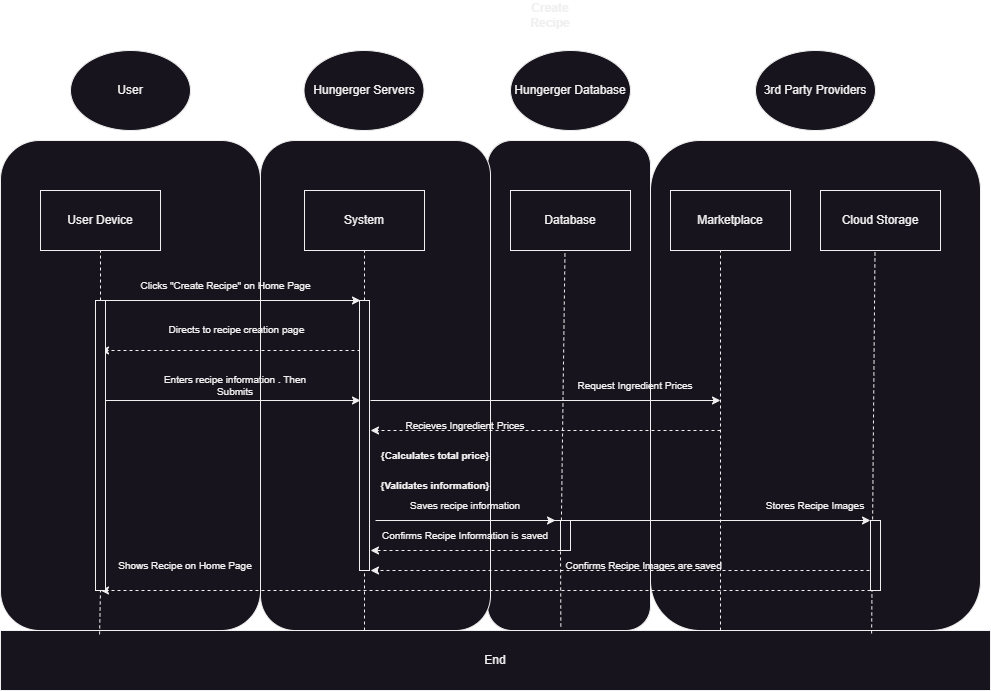
* **Information view:**
  + For the ER diagram and the relational model of our project, please refer to the following documents:
    - erDiagram.drawio



* + - RelationalModel.drawio



* **Process view:**
  + Please refer to CreateRecipe\_SequenceDiagram.drawio for the sequence diagram of UC4: Create Recipe use case.



* **Use case view:**

There are 12 defined use cases for our architecture. Their brief descriptions can be reached from

* uc\_specification\_Brief.docx.

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

Our use case diagram can be seen below, and also can be reached from “generalusecase.drawio”.

