FitNesse App

Analysis and Design Document

Student: Oancea Eduard

**Group: 30431**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 18/Mar/20 | 1.0 | Initial analysis and design | Oancea Eduard |
| 30/Apr/20 | 1.1 | Add Component and Deployment diagrams, as well as Data Model and Design Model | Oancea Eduard |
| 27/May/20 | 1.2 | Add Package Design, Future Improvements, Class Diagram/Architecture | Oancea Eduard |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4

2.2 Component and Deployment Diagrams 5

III. Elaboration – Iteration 1.2 5

1. Design Model 5

1.1 Dynamic Behavior 5

1.2 Class Design 5

2. Data Model 5

IV. Future Improvements 7

# Project Specification

Design and implement a desktop application for a fitness tracking application. The application should have 3 types of users(a regular user, a super user and an anonymous user).

The regular user can perform the following operations:

* CRUD on tracking entries
* CRUD on goals
* CRUD on the ingredients database
* View certain statistics

The anonymous user can perform the following operations:

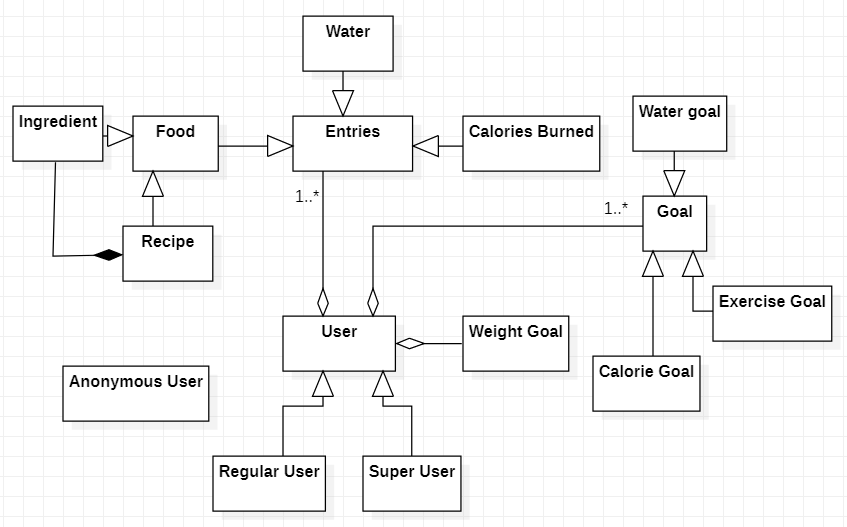
* Sign in
* Sign up

The super user can perform the following operations:

* CRUD on users
* CRUD on the food database

# Elaboration – Iteration 1.1

# Domain Model

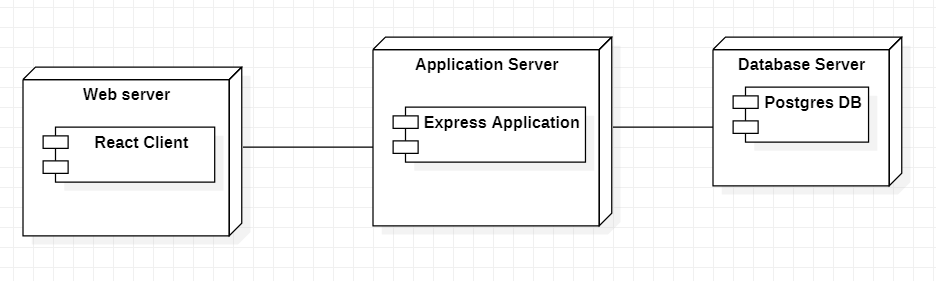


# Architectural Design

## Conceptual Architecture

This project is designed using the layered architecture pattern, as it is a client-server application. It’s built with Javascript with React as the client-side framework and Express as the server-side framework. The data will be persisted in a Postgres relational database. The React application will use Redux for state management and redux-thunk for handling async operations.

## Component and Deployment Diagrams

**

# Elaboration – Iteration 1.2

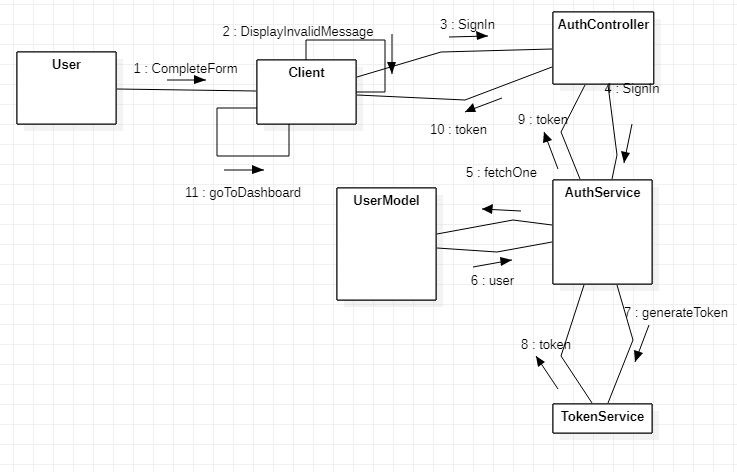
# Design Model

## Dynamic Behavior

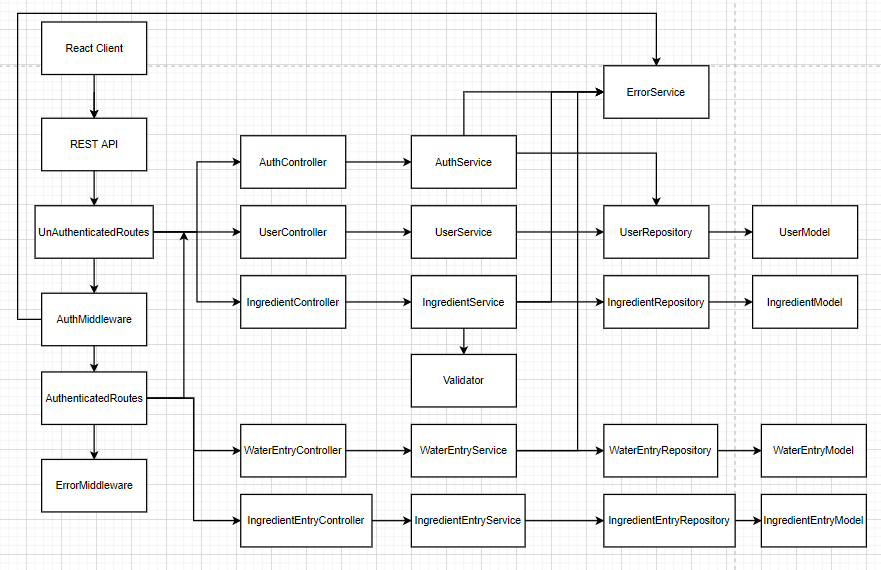
Sequence diagram for the authentication process.

*A screenshot of a cell phone

Description automatically generated*

**

## Class Design



The entire application is written entirely in functional style javascript.

The Express Framework uses middlewares. These are essentialy functions of the type (req, res, next) => void or (path, req, res, next) => void.

UnAuthenticatedRoutes and AuthenticatedRoutes are mappings between a route (e.g. /entries/food) and a controller method. If a request is satisfied, the next function will not be called. If any error is thrown, the ErrorMiddleware will catch it and send it to the client accordingly. The AuthMiddleware checks whether the token is valid and grants access to the AuthenticatedRoutes.

Afterwards, the controllers parse the request object and call the appropriate service.

The services contain the business logic, call validators and use the Repositories.

The repositories contain methods over the models. This way, the services don’t know how the database is handled.

The Client is written in React and is using redux and redux-thunk for state management. The redux state is accessible via selectors, which provide react components which subscribe to them with the current state value. The state is modified via actions, which are either called by reducers, either make API calls and may dispatch other actions. The reducers take actions, which have a type and a payload and modify a part of the state. When this happens, all selectors are called and the react components receive the fresh data.

The design patterns used (not necessarily implemented by hand) are:

* Dependency injection via bottle.js
* Middleware pattern
* Factory pattern, essentially every single component on the backend application is a function which receives its’ dependencies. This makes adding/removing dependencies very easy, without having to add them in the DI container if they are already registered for something else.
* React uses the observer internally, as well as redux
* The Façade pattern is used to make a wrapper over the fetch function.

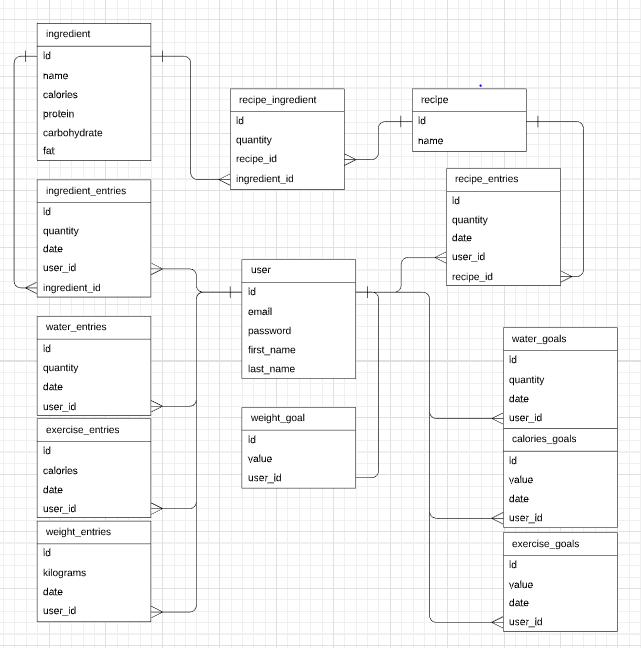
## Package Design

On the backend, the packages are split like this:

* app: entry point of the application
* bottle: the dependency injection container
* controllers
* mediator: improperly named, it holds the routes and maps the controller functions
* middlewares
* migrations
* models
* repositories
* services
* validators

This respects the Reuse-release Equivalence Principle (REP), as they all contain reusable classes.

# Data Model

**

# Future Improvements

There are many improvements available, as the initial project idea had more features than I thought I could do. As far as features go, the next things to do are:

* + Add 3rd party authentication (e.g. aws)
  + Add periodic goals for the user
  + Add foods made from ingredients and allow entries to be of this type
  + Provide metrics
  + Add weigh-in functionality
  + Make the UI mobile-responsive
  + Improve the UI