|  |
| --- |
| Bogazici University |
| FaTApp |
| Food and Activity Tracking Application |

|  |
| --- |
| Fatih Küçük  Fall 2016 |

# Introduction

This project has been developed for

# Requirements Document

This document contains three main sections. In the first section, functional requirements have been placed. In the second section non-functional requirements have listed. And lastly the terminology used in this project has been defined in the glossary.

There are also mockups added to clarify the flow of the screens.

## Functional Requirements

### Registration

1. The user must register to the system to use it
2. The user must provide the following information to register: (E-mail Address, Password, First Name, Last Name, Date of Birth, Gender [male, female], Height, Weight). All fields are mandatory.
3. All fields must have appropriate validations.
4. The user must be forced to enter complex password. The password must satisfy these complexities:

The password must

* 1. be at least six characters
  2. contain at least one capital letter
  3. contain at least a number
  4. contain at least a letter

1. An email has to be sent to user to validate his/her email address.
2. The user is allowed to use the system, unless he/she validates his/her email address.

### Login

1. The user logins the system with the email address and the password that he/she provided in the registration.
2. If the login is successful, the user must directed to the home page.
3. The user must have the ability to reset his/her password, if he/she forgets it. The user enters his/her email address. The system sends a link to confirm this behavior. This link redirect he/she to the password definition page.

### Updating Weight

1. User can be able to update his/her weight anytime he/she wants.
2. The update moment (day | time) must be recorded.
3. The user must have the ability to view his/her previous recordings (through timeline).

### Entering Consumed Food Information

1. There will be meals in the system: Breakfast, Lunch, Dinner, Snack.
2. The user must choose a meal before entering food.
3. The system recommends daily values. This [source](https://www.dsld.nlm.nih.gov/dsld/dailyvalue.jsp) shall be used to calculate them.
4. User enters the food the he/she ate and its quantity via searching from the data provided by the USDA Api.
5. The system should help the user to find the appropriate food he/she is looking for by the following actions:
   1. Listing the recent foods added by the user
   2. Listing the most common foods added by user for the last three months
   3. Listing the recent meals added by the user
   4. Listing the recipes defined by the user.
6. User must have the ability to change the quantity of the food selected from the search results. The quantity units provided by the USDA Api is used.
7. The user shall see the basic total calorie information of the selected food and quantity.
8. The user shall see the macronutrient values of the food selected. These are: Carbohydrate, Protein, and Fat, Fiber,
9. The user shall see the micronutrient values of the selected food also. These are: Calories from fat (1 g fat = 9 calories), Total Fat (g), Saturated fat (g), Trans fat (g), Cholesterol (mg), Sodium (mg), Total Carbohydrate (g), Fiber (g), Sugars (g), Protein (g), Vitamins and Minerals
10. The application must store this data, and the record moment.
11. The user can modify this data later on.
12. The user can enter past data.

### Entering Activity Information

1. The user can enter his/her activity by searching from the list provided nutristrategy.com through our application.
2. The system must show recent activities to the user, so the user can select them without making a new search.
3. The system must show distinct top ten most common activities entered by the user in the last two month. So the user can select from them, in spite of making a database search.
4. The user must enter the duration of the activity in minutes.
5. The calorie burned must be calculated according to the activity type and the duration.
6. The burned calories must be visible to the user.
7. This information must be recorded.
8. The user can modify this data later.
9. The user can enter past data.

### BMI

1. The application must calculate the BMI according to the data user provided by the following formula:
2. The BMI must be updated as the user updates his/her weight information.

### Timeline

1. The application must keep a timeline of the data user provided.
2. Timeline must contain following information: activities, food.
3. Timeline can be viewed daily, weekly or monthly.
4. Timeline must be limitless. No retention.

### Reports

1. The user must be able to see the macro and micro nutrients’ values, daily, weekly, monthly or for a custom time period.
2. The user must be able to see his/her weight trend, unless he/she did not log this data.

### User Interfaces (Mockups)

The conceptual design has been generated based on the Software Description document provided by the customer and the meetings had with the customer.

These mockups are also available online with working links through the pages: <https://ninjamock.com/s/SQLTF>

****

Figure 1 Registration Page

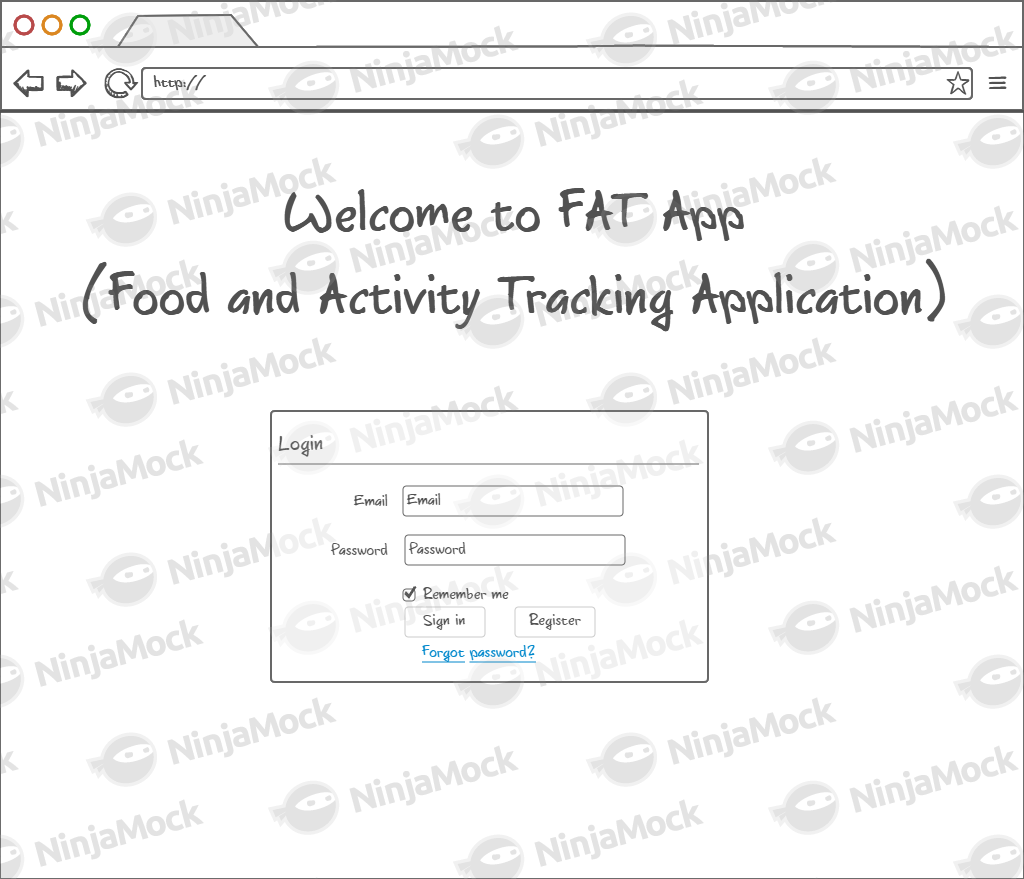


Figure 2 Login Page

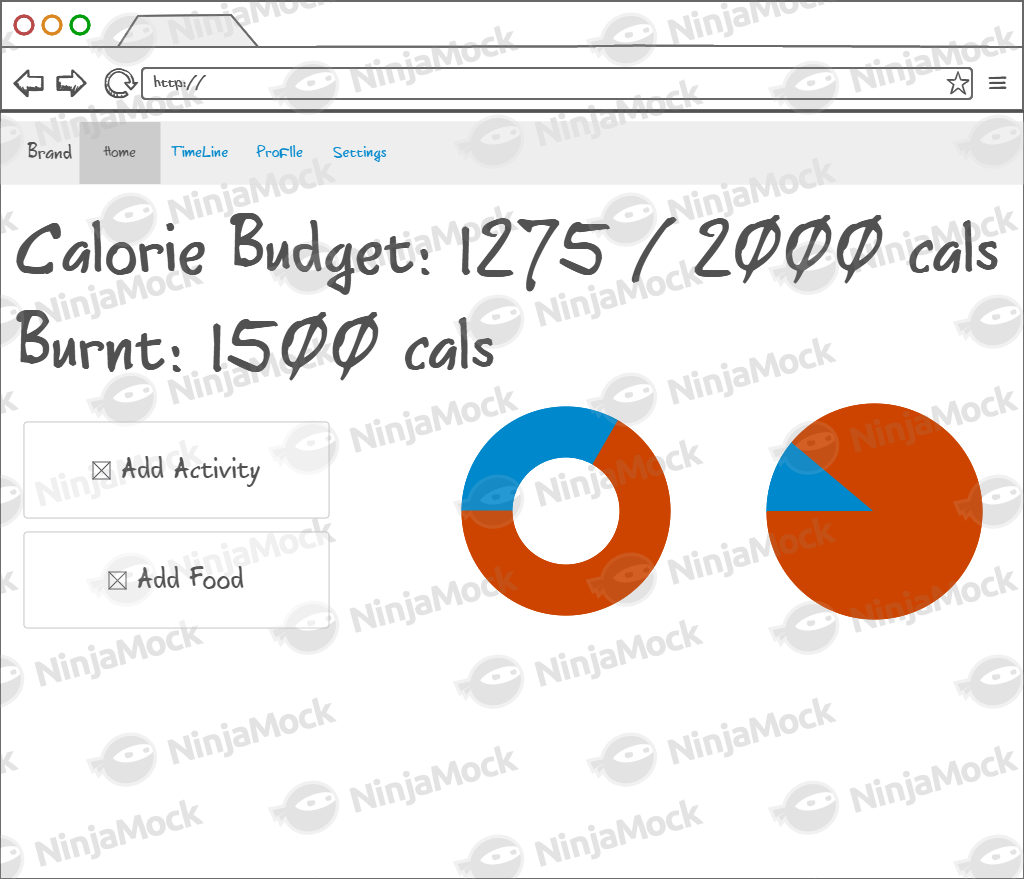


Figure 3 Home Page – Dashboard



Figure 4 Log Food Page



Figure 5 Log Activity Page



Figure 6 Timeline and Reports Page

## Non-Functional Requirements

1. The password of the user must be stored as encrypted in the database.
2. All search actions must not exceed four seconds. Caching must be arranged where necessary.
3. Metric system will be through the system.
4. The system must be developed using Tomcat/Java or Tornado/Python

## Glossary

**Food:** Any kind of atomic item in the USDA database. Example: Apple.

**Meal:** Breakfast, Lunch, Dinner or Snack. A meal contains at least a food or a recipe.

**Recipe:** A custom dish, made by combining different food. Example: 2 apples and a pie.

# Software Design Specifications Document

## Introduction

This application is built on client server architecture.

## System Architecture

### Architectural Design

The system will be built layered architecture. The layers are different java packages, and interact with each layer using java interfaces. This gives the ability to separate the layers physically, and also tends to a much more maintainable and testable code.

Each layer can only interact with adjacent layers. The cross cutting layers are available in every package, they are shared.

C:\Users\fat\Google Drive\egitim\BOUN\SWE 573 Software Development Practice\Diagrams\FatApp_Architecture.png

Figure 7 System Architecture n-Layer Design

#### UI Layer

This layer is responsible for user interface design and some basic validations. This layer should not contain any business logic.

Since this is a web project it will be implemented with html, javascript and css. But since it is abstracted from the whole project, it will be easy to implement a UI for other different platforms. A native mobile UI can be integrated with this system easily, if it is needed later.

#### WebApi

This layer is the service layer. For this project it will be an implementation of REST Api.

#### Repository

This layer will be used to abstract the data. The WebApi will interact with this layer to get data with the ignorance of the data source. For this project it will convert the database models and USDA’s Food Api into domain models.

#### Domain Models

This layer contains the Domain Models which are available for all layers. The domain models are the java objects that carry data or contain model information.

#### Security

This layer is responsible for authentication and authorization.

#### Logging

For logging information and error, this layer will be used.

### WebApi – Resource Enpoints

This section describes the endpoint addresses and their responsibilities. Sample endpoint addresses are shown next to path declaration.

All resources will be available in the “../webapi/” path.

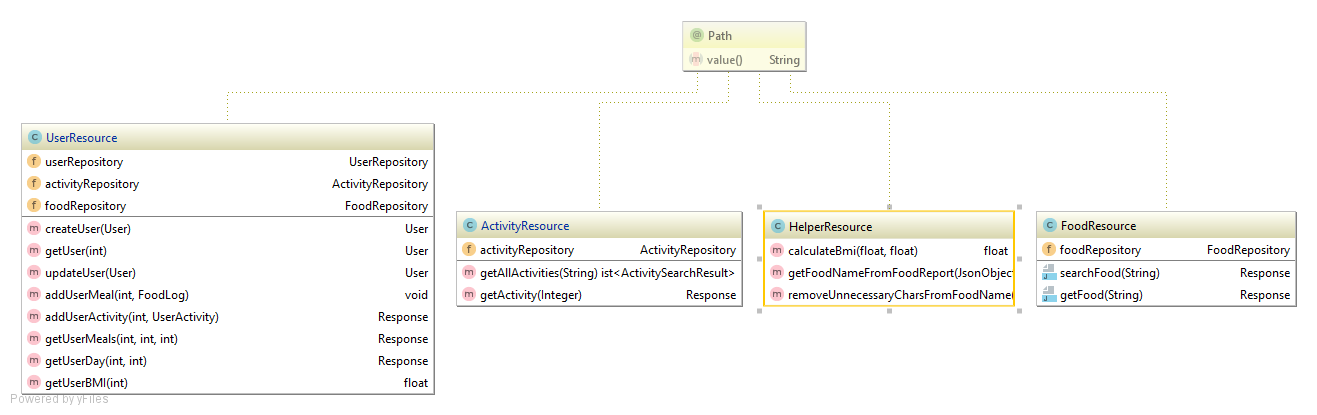


Figure 8 Resources Diagram

#### User Resource

@Path(**"users"**) *//http:localhost:8080/webapi/users***class** UserResource

@POST  
**method** User createUser(User user)

Should save the user object to the database and returns it back with the userId generated by the database.

@GET  
@Path(**"{userId}"**) *//http:localhost:8080/webapi/users/123***method** User getUser(@PathParam(**"userId"**) **int** userId)

This endpoint should get the User object by its id.

@PUT  
**method** User updateUser(User user) *//http:localhost:8080/webapi/users/123*

This endpoint should update the user based on changes in the parameter.

@POST  
@Path(**"{userId}/meals"**) *//http:localhost:8080/webapi/users/123/meals*  
**method void** addUserMeal(@PathParam(**"userId"**) **int** userId, FoodLog foodLog)

Endpoint for logging food consumption.

@POST  
@Path(**"{userId}/activity"**)*//http:localhost:8080/webapi/users/123/activity*  
**method** Response(UserActivity) addUserActivity(@PathParam(**"userId"**) **int** userId, UserActivity userActivity)

Endpoint for logging a user’s activity. Returns a response object.

@GET  
@Path(**"{userId}/meals"**)

*//http:localhost:8080/webapi/users/123/meals?startDay=20161208&endDay=20161215*

*Or for a day*

*//http:localhost:8080/webapi/users/123/meals?startDay=20161208&endDay=-1*  
  
**method** Response(UserMeal) getUserMeals(@PathParam(**"userId"**) **int** userId,  
 @QueryParam(value = **"startDay"**) **int** startDay,  
 @QueryParam(value = **"endDay"**) @DefaultValue(**"-1"**) **int** endDay)

Endpoint for loading consumed meals information. Response object contains “UserMeal”.

@GET  
@Path(**"{userId}/{day}"**) *//http:localhost:8080/webapi/users/123/20161231*

**method** Response(UserDay) getUserDay(@PathParam(**"userId"**) **int** userId,  
 @PathParam(**"day"**) **int** day)

Endpoint for loading a whole day with all information about UserMeal and List<UserActivity>. Day will be in yyyyMMdd format. Forinstance 31.12.2016 must be in 20161231 format.

@GET  
@Path(**"{userId}/bmi"**) *//http:localhost:8080/webapi/users/123/bmi*  
**method float** getUserBMI(@PathParam(**"userId"**) **int** userId)

Returns the specified user’s calculated BMI.

#### Food Resource

This resource contains all necessary endpoints related with Food.

@Path(value = **"foods"**) *//http:localhost:8080/webapi/foods*  
**class** FoodResource

@GET *//http:localhost:8080/webapi/foods?keyword=apple*  
**method** Response searchFood(@QueryParam(value = **"keyword"**) String keyword)

This endpoint is for searching food. It is abstracted from USDA Food Api.

@GET  
@Path(**"{foodId}"**) *//http:localhost:8080/webapi/foods/0042457*  
**method** Response(Food) getFood(@PathParam(**"foodId"**)String foodId )

Returns Food object inside the Response body.

#### Activity Resource

Activity related endpoints will be in this resource.

@Path(**"activities"**) *//http:localhost:8080/webapi/activites***class** ActivityResource

@GET *//http:localhost:8080/webapi/activites?keyword=swim*  
**method** List<ActivitySearchResult> getAllActivities( @QueryParam(value = **"keyword"**) String keyword)

Gets the search result of the activities, based on the keyword.

@GET

@Path(**"{activityId}"**) *//http:localhost:8080/webapi/activites/1234***method** Response(Activity) getActivity(@PathParam (**"activityId"**) Integer activityId)

Gets the activity by its id. If non existent activity is specified, Response object returns “Not Found” status int its header.

### Models

#### Food, Nutrient, Measure

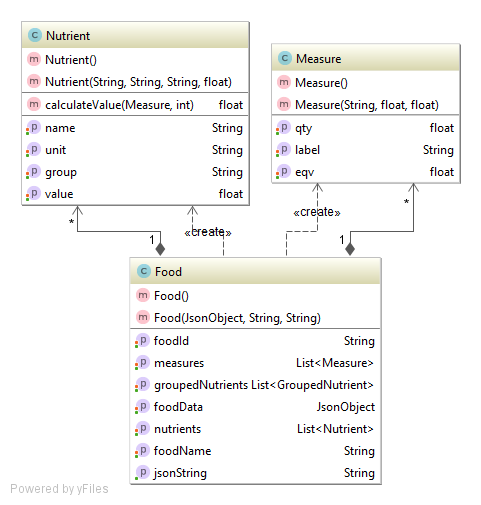


Figure 9 Food and related Classes

This model is the main Food object. It contains available measures and nutrient values. The USDA’s Food Api result will be carried in foodData property. And when this object is initialized the measure and nutrient values must be generated from the Json data.

#### UserActivity, Activity

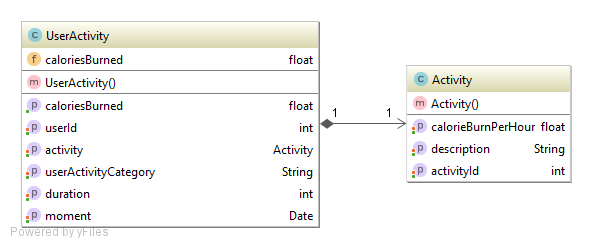


Figure 10 Activity and Related Classes

The Activity object contains the id, description and calorie burn for an hour. The UserActivity object has the Activity object. The caloriesBurned can be (calculated by UserActivity.duration \* UserActivity.Activity.calorieBurnPerHour).

#### UserMeal, Meal, FoodConsumption, MealType

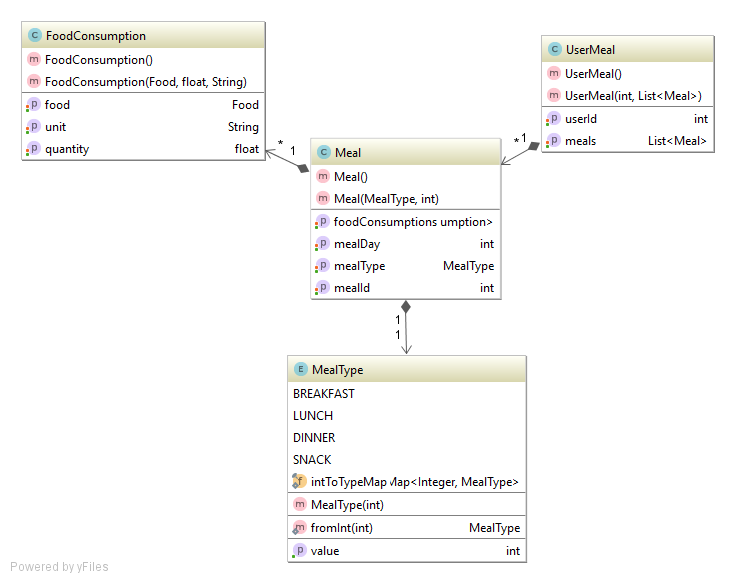


Figure 11 Meal and related classes

MealType should be an enum, for type safety and easier development. A UserMeal consists of list of meals. A meal has a MealType and contains FoodConsumptions. FoodConsumption object also has a 1 to 1 relationship with the Food object. FoodConsumption object holds how much a Food is consumed. Meal object holds the information of the MealType (Breakfast, Lunch, Dinner or a Snack). And the UserMeal holds the day information and related user.

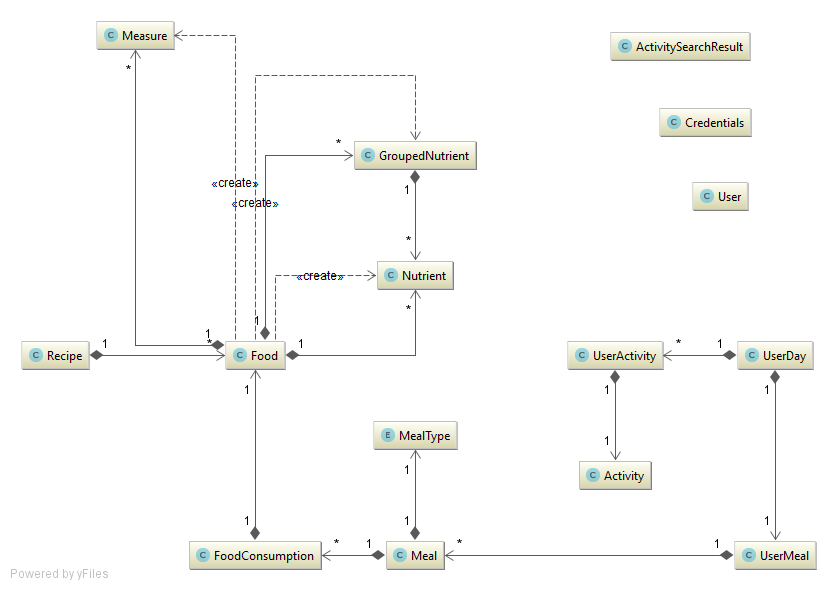


Figure 12 All Models and their Relations

### Repository

This section will cover the classes that interact with the data. The interfaces will be given. The implementation should be done based on these interfaces. The basic behavior of the methods are CRUD operations to the database. The special cases are explained for the specified method.

#### Activity Repository

**public interface** IActivityRepository {  
 **int** addUserActivity(UserActivity userActivity);  
  
 List<ActivitySearchResult> searchActivities(String keyword);  
  
 Activity getActivity(Integer activityId);  
  
 List<UserActivity> getUserActivities(**int** userId, **int** day);  
}

#### User Repository

**public interface** IUserRepository {  
 User createUser(User user);  
  
 User getUser(**int** userId);  
  
 User updateUser(User user);  
  
 User getUserByEmail(String email);  
  
 **int** getUserIdByEmail(String email);  
  
 **boolean** authenticate(String username, String password);  
}

#### Food Repository

**public interface** IFoodRepository {  
 Response searchFood(String keyword);  
  
 UserMeal getUserMeal(**int** userId, **int** startDay, **int** endDay);  
  
 **int** addUserMeal(**int** userId, Meal meal);  
  
 **void** logUserFood(**int** userId, FoodLog foodLog);  
  
 **int** createUserMeal(**int** userId, **int** mealTypeId, **int** day);  
  
 JsonObject getFoodReportFromUSDA(String foodId);  
  
 Food getFoodData(String foodId);  
}

searchFood method should contact to the USDA’a food api.

getFoodData method should first check if the data is available in our internal database, if it is then it should return it from the database. If it is not it should get it from the USDA’s api and saves it to the database.

### Data

A relational database will be used. The tables, columns, their types and relationships are shown on the ER-Diagram.

For data integrity, no physical foreign key will be used. Data integrity will be provided by the software itself.

There will be no database object (trigger, stored procedure, function). And the database will not contain any business logic.

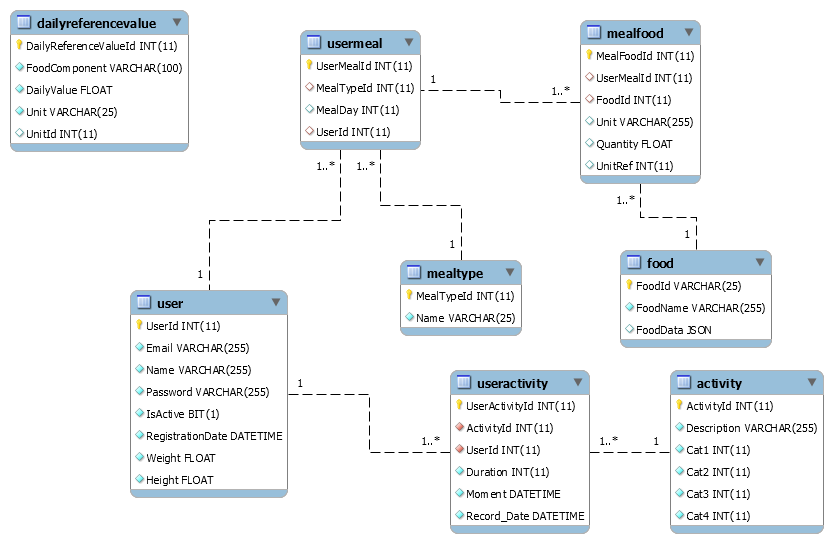


Figure 13 ER-Diagram

# Tests

The Models, Repository and WebApi layers are tested separately. All tests are passing, so the results are not mentioned here.

## Unit Tests

|  |  |  |
| --- | --- | --- |
| Test Method | Unit to Be Tested | Explanation |
| WHEN  \_CalculateBMI  \_SHOULD\_BeCorrect | HelperResource  .calculateBMI  (float height, float weight) | Provide Height and Weight info for the tested method. Expect the right calculation of the BMI with 0.05 delta. |
| WHEN  \_CallHelperGetFoodName  \_SR\_ProperFoodName | HelperResource  .getFoodNameFromFoodReport  JsonObject object) | Provide a dummy JSON object from the USDA’s api. The tested method should remove the unnecessary annoying characters. |
| Check  NutrientMeasureCalc | Nutrient  .calculateValue  (Meausure m, float quantity) | The Nutrient model calculates its own nutrition values, based upon provided measure. |
| Check  NutrientMeasureCalc  DifferentQuantity | Nutrient  .calculateValue  (Meausure m, float quantity) | Same unit, with different quantities. |
| Food  MeasureTest | Food  .getMeasures() | Food object carries its own provided measures. This test checks whether it is converting its measure values correctly. |
| Food  Measure\_Label | Food  .getMeasures() | Food object carries its own provided measures. This test checks whether it is converting its measure labels correctly. |
| Food  Nutrients\_Count | Food  .getNutrients() | Food object contains its own nutrients. This test tries to check the correctness of the measures by counting them. |
| Food  Nutrients\_Count  \_FOR\_01009 | Food  .getNutrients() | Food object contains its own nutrients. This test tries to check the correctness of the measures by counting them, fort the food item with id 01009. |
| Food  Nutrients  \_CalculateNutritientValue | Food  .getNutrients() | This is test gets the specified nutrient from the specified food and check its value. |
| Food  Nutrients  \_CalculateNutritientValue  WithDifferentQuantity | Nutritient  .calculateValue() | The behavior of the specified nutrient when the measure is changed. |
| WHEN  \_GetDemoKey  \_SR\_DemoKey | HelperRepository  .getUSDAApiKey() | For limitations, two different keys have been used for USDA. The system gives one of the randomly. |

## Integration Tests

|  |  |  |
| --- | --- | --- |
| Test Method | Related Method(s) | Explanation |
| WHEN  \_GetFood  \_SR\_Food | FoodRepository  .getFoodData  (String foodId) | Should load the Food from database. |
| WHEN  \_GetFood  \_SR\_FoodJSON | FoodRepository  .getFoodData  (String foodId) | Should load the Food.FoodData Json Object correctly from the database. |
| WHEN  \_NonExistentGetFood  \_SR\_FoodJSON | FoodRepository  .getFoodData  (String foodId) | Provide a nonexistent foodId in the database. Should load it correctly from the USDA’s Food Api. |
| WHEN  \_GetFoodReportFromApi  \_SR\_ReturnRelatedReport | FoodRepository  .getFoodReportFromUSDA  (String foodId) | Testing if the Food Api is working. |
| WHEN  \_GetFoodReport  FromApiForNonExistentFood  \_SR\_ReturnNull | FoodRepository  .getFoodReportFromUSDA  (String foodId) | Provide a valid but nonexistent foodId for the USDA’s Food Api. It should return NULL. |
| WHEN  \_GetNonExistentFood  \_ShouldBeSaved | FoodRepository  .getFoodReportFromUSDA  (String foodId) | Provide a nonexistent Food in the database. The system should save it to the database. |
| WHEN  \_GetFood  \_SR\_CorrectFormatted  FoodName | FoodRepository  .getFoodReportFromUSDA  (String foodId) | The Food Name should be saved in the database with proper name. |
| WHEN  \_GetUserMeal  \_SR\_NotNull | FoodRepository  .getUserMeal(int userId, int day(yyMMdd)) | Load a user’s meals, for a date. |
| WHEN  \_AddValidUser  \_SHOULDRETURN\_PositiveID | UserRepository.createUser(User) | Saving a user to the database. |
| WHEN  \_AddMealToUser  \_SR\_ValidMeal | FoodRepository.addUserMeal(int userId, Meal) | Saving a user’s meal to the database. System should create a meal in the database if it is not existing. If it is existing should add the logged food to it. |
| WHEN  \_AddUserActivity  \_SR\_ActivityID | ActivityRepository  .addUserActivity  (UserActivity) | Saving a user’s activity. |
| WHEN  \_SearchActivity  \_SR\_ActivityResults | ActivityRepository  .searchActivities  (String keyword) | Should return all activities containing keyword provided. |
| WHEN  \_SearchActivityWithMinus1  \_SR\_AllActivityResults | ActivityRepository  .searchActivities  (String keyword) | If “-1” is provided as the keyword, all activities should return. |
| WHEN  \_GetActivityById  \_SR\_RightActivity | ActivityRepository  .getActivity(int activityId) | Testing for getting the Activity by its id. |

# Time Management

## Activities

### Activity Log

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Label | Comment | Estimated Duration (min) | Actual Duration (min) | Date (m/d/yyyy hh:mm) |
| Meeting | Class Meeting | 120 | 120 | 9/20/2016 19:00:00 |
| Training | GitHub HelloWorld activity (https://guides.github.com/activities/hello-world/) | 10 | 20 | 9/25/2016 21:00:00 |
| Research | Research on Java technologies | 60 | 180 | 9/25/2016 22:00:00 |
| Meeting | Class Meeting | 120 | 120 | 9/27/2016 19:00 |
| Analysis | Examine the Project Description Document provided by the Instructor | 60 | 60 | 10/2/2016 1:00:00 |
| Research | [Read the "Quantified Self" article on Wikipedia](https://en.wikipedia.org/wiki/Quantified_Self) | 60 | 60 | 10/2/2016 2:00:00 |
| Research | Applied for API key for ndb api, and stored the key at the private spreadsheet file | 10 | 20 | 10/2/2016 3:00:00 |
| Research | Examine the USDA API | 60 | 120 | 10/2/2016 3:30:00 |
| Research | [Read the "Body Mass Index" article on Wikipedia](https://en.wikipedia.org/wiki/Body_mass_index) | 15 | 60 | 10/3/2016 12:00:00 |
| Meeting | Class Meeting | 120 | 120 | 10/4/2016 19:00 |
| Training | HTML | 180 | 180 | 10/7/2016 1:00 |
| Training | HTML | 60 | 180 | 10/7/2016 23:00 |
| Analysis | Requirements Elicitation | 120 | 150 | 10/8/2016 1:30 |
| Analysis | Creating initial mockups | 120 | 180 | 10/09/2016 23:00 |
| Meeting | Class Meeting | 120 | 120 | 10/11/2016 19:00 |
| Training | CSS | 180 | 180 | 10/15/2016 23:00 |
| Research | Java Web Frameworks | 120 | 120 | 10/16/2016 1:00 |
| Research | Javascript frameworks | 120 | 180 | 10/17/2016 4:00 |
| Meeting | Class Meeting | 120 | 120 | 10/18/2016 19:00 |
| Training | Coursera: SPA | 180 | 250 | 10/21/2016 23:00 |
| Training | Coursera: SPA | 120 | 200 | 10/22/2016 1:00 |
| Analysis | Refining Requirements | 180 | 200 | 10/23/2016 23:00 |
| Analysis | Creating refined mockups | 120 | 150 | 10/24/2016 3:00 |
| Meeting | Class Meeting | 180 | 180 | 11/1/2016 19:00 |
| Meeting | Class Meeting | 180 | 180 | 11/8/2016 19:00 |
| Meeting | Class Meeting | 180 | 180 | 11/15/2016 19:00 |
| TechnicalIssue | tomcat setup | 30 | 180 | 11/20/2016 0:00 |
| Training | Inclass training with Mustafa | 360 | 360 | 11/20/2016 10:00 |
| Training | Django Tutorial | 180 | 180 | 11/20/2016 22:00 |
| Analysis | Creating Architecture | 120 | 120 | 11/21/2016 1:00 |
| Analysis | Creating models | 120 | 120 | 11/22/2016 1:00 |
| Meeting | Class Meeting | 180 | 180 | 11/22/2016 19:00 |
| Analysis | Deciding webapi endpoints | 120 | 120 | 11/23/2016 0:30 |
| Analysis | Deciding db tables and columns | 120 | 120 | 11/24/2016 0:45 |
| Training | Jersey Tutorial | 180 | 180 | 11/25/2016 0:15 |
| Training | Spring Tutorial | 180 | 180 | 11/26/2016 1:00 |
| Development | bootstrapping hello world with web api (Jersey) | 60 | 120 | 11/27/2016 0:10 |
| Development | creating endpoints with intelliJ | 120 | 240 | 11/27/2016 14:00 |
| Development | installing mySql | 60 | 60 | 11/28/2016 0:10 |
| Meeting | Class Meeting | 180 | 180 | 11/29/2016 19:00 |
| TechnicalIssue | integrating MySql with IntelliJ problem | 30 | 120 | 11/30/2016 0:45 |
| Development | integrating MySql with IntelliJ | 60 | 120 | 11/30/2016 2:45 |
| Development | implementing repository | 120 | 120 | 12/2/2016 0:10 |
| Analysis | deciding UI components | 60 | 160 | 12/3/2016 12:00 |
| TechnicalIssue | integrating sql2o with mysql and tomcat | 30 | 180 | 12/3/2016 16:00 |
| TechnicalIssue | Deploying db with Tomcat | 30 | 180 | 12/4/2016 0:10 |
| Development | creating homePage (UI) | 60 | 120 | 12/4/2016 12:00 |
| Development | creating Users table (db) and integration with project | 60 | 120 | 12/4/2016 17:00 |
| Development | Log Food Design | 30 | 60 | 12/5/2016 0:00 |
| Meeting | Class Meeting | 180 | 180 | 12/6/2016 19:00 |
| Development | Log Activity Design | 30 | 120 | 12/7/2016 22:00 |
| Development | Log Food REST integration | 30 | 180 | 12/8/2016 23:00 |
| TechnicalIssue | UI javascript ajax post problem | 60 | 240 | 12/9/2016 9:00 |
| Development | Food model nutrient calculation | 30 | 60 | 12/9/2016 23:45 |
| TechnicalIssue | UI css problem | 120 | 220 | 12/10/2016 14:00 |
| Development | Log Activity REST integration | 30 | 120 | 12/10/2016 22:45 |
| TechnicalIssue | JSON Object backslach problem | 60 | 180 | 12/11/2016 1:00 |
| TechnicalIssue | UI javascript jquery selection error | 60 | 180 | 12/11/2016 14:00 |
| Meeting | Demo with the Customer | 180 | 180 | 12/13/2016 19:00 |

Figure 14 Time Spent Activity Report

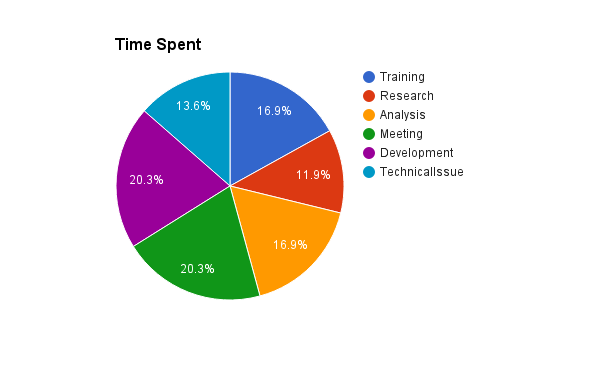


Figure 15 Estimated vs Actual Time

# C:\Users\fat\Desktop\Diagrams\EstimatedvsActualReport.png

# Resources

Project has a GitHub page available in: <https://github.com/fkucuk/Fall2016Swe573>

The project has been deployed in Amazon AWS with the following address:

<http://fatappenv.eu-central-1.elasticbeanstalk.com/>

# System Requirements

The application runs on Tomcat Server (8.5.9), and MySql Server (5.7.16).

The application also needs the following libraries to run.

Java 8

junit 4.12

mysql.connector.java 5.1.40

org.glassfish.hk2.api 2.5.0

javax.json.1.0.4

jersey 2.24.1

org.sql2o 1.5.4

Java SDK 1.8

# Deployment

The files for deployment are available in the deployment folder.

The application is in WAR file, named FatApp.WAR.

The dump files for creating the database tables and filling the required data is available in deployment/db folder.

Database parameters are configurative. The application will read the following parameters, if a property file is specified.

RDS\_HOSTNAME *127.0.0.1*

RDS\_DB\_NAME *fatapp*

RDS\_PORT *3306*

RDS\_USERNAME *fatapp*

RDS\_USERNAME *stronpwd*