Information Management & Systems Engineering
Milestone 2: Nutrition Plan Application

Sarah Thill[*] Leon Grund[†]

January 18, 2022

Abstract. The goal of our software system is to combine different aspects from health, nutrition, and fitness apps in one single platform. We help users to create optimal meal plans from a scientific perspective, which help them archive their fitness goals.

Technology stack. Docker, MongoDB, MySQL, Python-Flask

[*]E-mail: thills44@unet.univie.ac.at, Student-ID: 11905855, Study Program: Data Science
[†]E-mail: grundl89@unet.univie.ac.at, Student-ID: 01407726, Study Program: Data Science

Contents

1 RDBMS Part

1.1 Configuration of Infrastructure

We implemented our project in *Flask*, a micro web framework written in Python. For the front-end we chose *jinja* bootstrap templates, which is Flask's default template engine. As back-end we implemented the REST API in the Flask framework. For encryption we chose on-the-fly certificates when we start our Flask app.[1] For deployment we chose `Docker` to deliver our software in isolated containers, communicating between our Flask, MySQL, and MongoDB services.

- To build and start container run: `$ docker-compose up -d`
- Then open: https://localhost:5000/

1.2 Database design

We implemented the first part of our project in MySQL. The database design is rather straight forward, where each entity from our ERD model in Figure 2 becomes a table. Moreover, both many-to-many relationships at `GROCERY` were implemented as bridge tables, by referencing the primary keys of each associated entity. In total we created seven SQL tables, see Figure 1 and Appendix A.3.

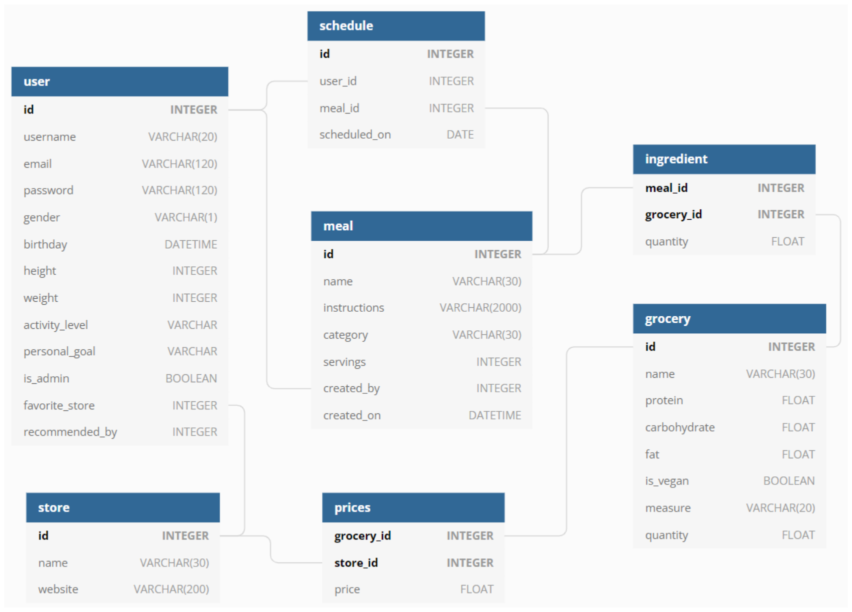


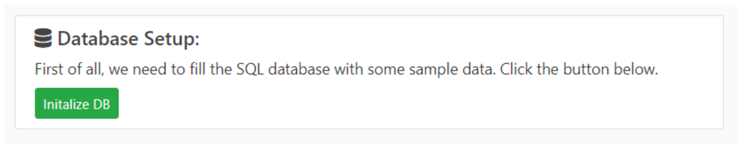
Figure 1: Logical database design generated with dbdiagram.io.

[1] Note that you will see a "Your connection isn't private" warning when you start the app. Just click the advanced button and continue to 192.168.88.244 (unsafe). In case of troubles with your browser, you could remove the `ssl_context='adhoc'` setting in the `run.py` file.

Since we only query records by `id`, except for login, we did not set additional keys even though the `name`, `email` or `website` columns were also defined to be unique. Users or stores cannot be deleted in our implementation (not defined in our use-cases), but we allow admins to delete meals. Therefore, we also delete associated records in the `INGREDIENT` and `SCHEDULE` tables in a cascading way.

1.3 Data import

The first page you should see after launching our app using Docker will ask you to initialize the MySQL database.


Database Setup:

First of all, we need to fill the SQL database with some sample data. Click the button below.

Clicking the button "Initialize DB" executes the `sql_create_all()` script, which executes all create table statements from Appendix A.3, followed by the script `sql_fill_sample_data()`, which fills all SQL tables with data from our sample `csv` files in the folder `sample_data`. Each file is named according to the respective table. We then read, format and insert the data using Python. Meal schedules are generated randomly for each user. Moreover, we didn't want to investigate too much time in looking up each grocery price for each store. Therefore, we only looked up prices for one store and estimated the other store prices normally distributed with given prices as mean and different variances per store.[2] To get more interesting results for our reports, we also randomly skipped a few grocery prices for particular stores. For instance, say store X does not provide a price for eggs, it lowers the meal coverage score (see "top 5 stores" report in 3.2) since all meals with eggs as ingredients are not covered.

1.4 Implementation of a Web system

After initialization of the database you will be redirected to the login page, where you can login with

- username: `"sarah"`, password: `"sarah"` (Admin), or
- username: `"leon"`, password: `"leon"` (User).[3]

Please note that only `sarah` is an admin, who has access to reports and NoSQL migration. The navigation bar and profile page will look a bit different for admins and normal users. In Appendix A.1 we present a more comprehensive user guide with screenshots and short summaries for each page we implemented.

[2]Formally, we generated $p_{i,j} \sim \mathcal{N}(p_{i,0}, 0.05j + 0.01)$, where $p_{i,0}$ refers to the given prices for the ith grocery from the file `prices.csv` and $p_{i,j}$ to the estimated price of the ith grocery at the jth store, $j \geq 1$. We lower bounded prices at 0.01 to ensure non-negative grocery prices.

[3]Of course, you can also login with other existing users, i.e. `george`, `jennifer`, `serena`, or `will`, using the username as password as well. Also feel free to register a new user yourself, but then the meal plan will be empty of course.

2 NoSQL Design

2.1 Design Motivation

The seven tables from our RDBMS design are reduced to five collections in the NoSQL design, by integrating our bridge tables into the collections to improve efficiency. In the following we summarize each collection with our reasons for indexing and denormalization.

2.1.1 User

The `user` collection is similar to the `user` table in the RDBMS design. The main difference is that only attributes which are not `NULL` are contained in the collection. If a user did not fill out some information, e.g., their height, there will not be an attribute height in their entry in the MongoDB collection. First, we thought of integrating schedule as sub-collections, but decided against it since schedule can grow infinitely large. Of course, one could consider deleting old schedules (or move them to some external backup database), but then we would loose the option to analyze historical data in our reports. Hence, we didn't use any denormalization on the `user` collection.

We usually query or filter users based on the document ids, except for login or registry. Since login is one of our use-cases we also made `username` an index in addition to the `_id`, which is by default an index.[4]

```
user_sample = {
    '_id': 3,
    'username': 'george',
    'email': 'george@clooney.com',
    'password': 'george',
    'gender': 'M',
    'is_admin': 0,
    'favorite_store': 3
}
```

Code 1: JSON representation of `user` document.

2.1.2 Meal

The `meal` collection has a few enhancements compared to the `meal` table. In addition to the RDBMS fields, we also added ingredients as sub-collections per meal. The list of ingredients cannot be changed and cannot grow too large, so this was a good candidate for denormalization. Moreover, in our RDBMS design, we always calculate the macronutritions based on ingredients. Again, as our users cannot edit meals and the protein, carbohydrate, fats and calories are fixed per grocery, we included these key fields per meal as well. Similarly, we argued that the grocery prices won't change to often per store, so we added another sub-collection of meal prices per store, which also helped to reduce the

[4]One could also argue that `email` should be an index as well since we query this field when new users try to register, but it is certainly not a main-use case of our website.

complexity of our reports a bit. We decided against including all the information from the `grocery` table, since this would cause significant overhead when more users join our platform and much more meals are created.

For indexing, we kept the default `_id` as index as well as the `created_on`, which is filtered in one of our reports. Moreover, we made `name` an index, since it is queried when creating a new meal, i.e. one of our main use-cases.

```
meal_sample = {
    '_id': 1,
    'name': 'Pancakes',
    'category': 'breakfast',
    'kcal': 418,
    'protein': 23,
    'carbohydrate': 48,
    'fat': 14,
    'is_vegan': True,
    'instructions': 'Some instructions to cook meal.',
    'created_by': 1,
    'created_on': '2022-01-01',
    'ingredients': [{'grocery_id': 1, 'quantity': 20}, ...],
    'prices': [{'store_id': 1, 'price': 3.58}, ...]
}
```

Code 2: JSON representation of `meal` document.

2.1.3 Schedule

At first we tried to include the `schedule` as sub-collection for either the user or each meal. However, this made our reporting use-cases more complicated (e.g., to calculate meal scheduled by other users or total meals scheduled per user). Moreover, we the `schedule` can grow very large even for one particular user. For example, after one year we could have more than 1000 records per user (even more per meal), which are very frequently access for adding and removing meals from the weekly plan. Hence, we decided to implement the `schedule` as an individual collection again.

In comparison to our RDBMS table, however, we added two new attributes to improve efficiency in the reporting use-cases. First, we added the meal category, which will always be accessed together with the schedule to create the meal overview. And second, we added the `is_vegan` flag, which is needed in our "top 5 user" report to calculated a ratio of vegan meals scheduled.

For indexing we used `user_id`, `meal_id`, as well as `scheduled_on`. The latter index is particularly relevant for filtering our reports as well as create meal plan overviews of a given week.

```
schedule_sample = {
    '_id': ObjectId('61dab69d514622ed49d18d9f'),
    'user_id': 1,
    'meal_id': 1,
    'category': 'breakfast',
    'scheduled_on': '2022-01-01',
    'is_vegan': True
}
```

Code 3: JSON representation of `schedule` document.

2.1.4 Grocery

The `grocery` collection is remains similar as before, but we included each prices per store as sub-collection. We decided for this change to improve efficiency for the use-cases and reporting cases. We will only have a limited number of stores, so the array cannot grow too large. Again, if a particular store does not have a price for a given grocery, we omitted the store in the sub-collection.

Here we only kept the default index `_id` since it is the only field we filter for. Even the the grocery name is an unique attribute as well, we thought it won't improve the performance of any use-cases or reports to make this an index.

```
grocery_sample = {
    '_id': 2,
    'name': 'Avocado',
    'protein': 1.0,
    'carbohydrate': 4.0,
    'fat': 13.0,
    'is_vegan': 1,
    'measure': 'x',
    'quantity': 1,
    'prices': [{'store_id': 1, 'price': 1.0}, {'store_id': 2, 'price': 1.20}, ...]
}
```

Code 4: JSON representation of `grocery` document.

2.1.5 Store

Finally, we have our last collection, which is identical to our `store` table from before. The only reason why we didn't include this as a sub-collection for each user and grocery is that we produce our second report based on stores. Moreover, we allow each user the update her favorite store, so we need a list of available stores anyway. Again, indexing is only based on the default `_id` attribute.

```
store_sample = {
    '_id': 1,
    'name': 'billa',
    'website': 'https://www.billa.at/'
}
```

Code 5: JSON representation of `store` document.

3 Query Comparison of Reports

Please note that we included our SQL queries below using `with` statements to create named sub-query blocks for reasons of clarity and comprehensibility. Unfortunately, we only managed to get our MySQL server running with an older version which doesn't support `with`, so we had to unwind these blocks again as nested sub-queries in our implementation submitted on Moodle.

For the first report in Section 3.1 we had to use 7 `join`'s in SQL to get the desired result, which we managed to reduce to only 3 `lookup`'s in our MongoDb aggregation pipeline. The second report in Section 3.2 looks even more complex with 9 `join`'s in SQL and 4 `lookup`'s using MongoDb.

3.1 Report 1: Top 5 users

The objective of the first reporting use case is to gain an overview of the platform's users by meals created, meals scheduled, meal popularity[5] and vegan ratio. In our SQL implementation we had to query the tables `user`, `meal`, and `schedule` where we filtered the dates when the meals were created and scheduled. Moreover, to compute the vegan ratio we also had to query the tables `ingredient` and `grocery`. The average runtime for 10 queries was 5.24 ms.

```
with meal_details as (
    select m.id, case when sum(g.is_vegan) = count(g.is_vegan) then 1 else 0 end as is_vegan
    from meal m
    inner join ingredient i on i.meal_id = m.id
    inner join grocery g on g.id = i.grocery_id
    group by m.id
    )
, meals_created as (
    select u.id, u.username as name, count(distinct m.id) as meals_created
    from user u
    left outer join meal m on m.created_by = u.id
        and m.created_on between '2022-01-01' and date_add('2022-01-31', interval 1 day)
    group by u.id, u.username
    )
, meals_scheduled as (
    select u.*, count(s.id) as meals_scheduled, cast(100 * sum(m.is_vegan) / count(s.id) as unsigned) as vegan_ratio_pct
    from meals_created u
    left outer join schedule s on s.user_id = u.id
        and s.scheduled_on between '2022-01-01' and date_add('2022-01-31', interval 1 day)
    inner join meal_details m on m.id = s.meal_id
    group by u.id, u.name
    )
, meals_scheduled_by_others as (
    select u.*, count(s.id) as meal_popularity
    from meals_scheduled u
    left outer join meal m on m.created_by = u.id
    left outer join schedule s on s.meal_id = m.id
        and s.scheduled_on between '2022-01-01' and date_add('2022-01-31', interval 1 day)
    group by u.id, u.name
    )

select id, name, meals_created, meal_popularity, meals_scheduled, vegan_ratio_pct
from meals_scheduled_by_others
order by name desc
limit 5
```

Code 6: SQL query for top 5 users report.

Our respective MongoDB aggregation pipeline uses only 3 lookups, where the average runtime for 10 queries tested was a bit higher with 7.81 ms. However, we assume that NoSQL will perform much better compared to SQL on larger datasets. It is also possible that we used too many `$unwind` operations. I think we were a bit biased to "replicate" our SQL implementation.

[5]Meal popularity for some user X means that we count how often any user on our platform have scheduled any meal created by X.

```
mongo_db['user'].aggregate([
    {
        '$lookup': {
            'from': 'schedule',
            'pipeline': [{'$match': {'scheduled_on': {
                '$gte': datetime.date(2022, 1, 1), '$lte': datetime.date(2022, 1, 31)}}}],
            'localField': '_id',
            'foreignField': 'user_id',
            'as': 'user_schedule'
        }
    },
    {
        '$unwind': {
            'path': '$user_schedule',
            'preserveNullAndEmptyArrays': True
        }
    },
    {
        '$group': {
            '_id': '$_id',
            'name': {'$first': '$username'},
            'meals_scheduled': {'$sum': {'$cond': ['$user_schedule', 1, 0]}},
            'vegan_ratio_pct': {'$avg': '$user_schedule.is_vegan'}
        }
    },
    {
        '$lookup': {
            'from': 'meal',
            'localField': '_id',
            'foreignField': 'created_by',
            'as': 'meals'
        }
    },
    {
        '$unwind': {
            'path': '$meals',
            'preserveNullAndEmptyArrays': True
        }
    },
    {
        '$lookup': {
            'from': 'schedule',
            'pipeline': [{'$match': {'scheduled_on': {
                '$gte': datetime.date(2022, 1, 1), '$lte': datetime.date(2022, 1, 31)}}}],
            'localField': 'meals._id',
            'foreignField': 'meal_id',
            'as': 'other_schedule'}
    },
    {
        '$unwind': {
            'path': '$other_schedule',
            'preserveNullAndEmptyArrays': True
        }
    },
    {
        '$group': {
            '_id': {'id': '$_id', 'name': '$name', 'meal_name': '$meals.name'},
            'id': {'$first': '$_id'},
            'name': {'$first': '$name'},
            'meals_created': {'$first': '$meals.created_on'},
            'meals_scheduled': {'$first': '$meals_scheduled'},
            'meal_popularity': {'$sum': {'$cond': ['$other_schedule', 1, 0]}},
            'vegan_ratio_pct': {'$first': '$vegan_ratio_pct'},
        }
    },
    {
        '$group': {
            '_id': '$id',
            'name': {'$first': '$name'},
            'meals_created': {'$sum': {'$cond': [{'$and': [
                {'$gte': ['$meals_created', datetime.date(2022, 1, 1)},
                {'$lte': ['$meals_created', datetime.date(2022, 1, 31)]}
                ]}, 1, 0]}},
            'meal_popularity': {'$sum': '$meal_popularity'},
            'meals_scheduled': {'$first': '$meals_scheduled'},
            'vegan_ratio_pct': {'$first': {'$toInt': {'$multiply': ['$vegan_ratio_pct', 100]}}},
        }
    },
    {'$sort': {'$name': -1}},
    {'$limit': 5}
])
```

Code 7: Mongodb aggregation pipeline for top 5 users report.

3.2 Report 2: Top 5 stores

The objective of our second reporting use case is to gain an overview of the platform's stores by revenues, number of users selected, number of scheduled meals, and meal coverage. Here we had to use 9 `join`'s, mainly to compute the meal coverage by store, which was not that straight forward. We had to query all tables except for `meal`. The average runtime for 10 queries tested was 12.48 ms.

```sql
with ingredients_per_meal as (
    select meal_id, count(*) as num_ing
    from ingredient
    group by meal_id
    )
, ingredients_per_store_and_meal as (
    select store_id, meal_id, count(*) as num_ing
    from ingredient i
    inner join prices p on p.grocery_id = i.grocery_id
    group by store_id, meal_id
    )
, coverage_per_store_and_meal as (
    select s.store_id, m.meal_id, sum(s.num_ing = m.num_ing) = count(s.num_ing) as coverage
    from ingredients_per_store_and_meal s
    left outer join ingredients_per_meal m on m.meal_id = s.meal_id
    group by s.store_id, m.meal_id
    )
, coverage_by_store as (
    select c.store_id, cast(sum(c.coverage) as unsigned) as meal_coverage
    from coverage_per_store_and_meal c
    group by c.store_id
    )
, revenues_selection_schedules as (
    select
        store.id,
        store.name,
        coalesce(round(sum(case when c.coverage = 1 then i.quantity / g.quantity * p.price else 0 end), 2), 0) as revenues_ccy,
        count(distinct u.id) as users_selected,
        count(distinct s.id) as scheduled_meals,
        count(p.price)
    from
        store
        left outer join user u on u.favorite_store = store.id
        left outer join schedule s on s.user_id = u.id
            and s.scheduled_on between '2022-01-01' and date_add('2022-01-31', interval 1 day)
        left outer join coverage_per_store_and_meal c on c.store_id = store.id and c.meal_id = s.meal_id
        left outer join ingredient i on i.meal_id = s.meal_id
        left outer join grocery g on g.id = i.grocery_id
        left outer join prices p on p.store_id = store.id and p.grocery_id = i.grocery_id
    group by
        store.name
)

select r.id, name, revenues_ccy, users_selected, scheduled_meals, meal_coverage
from revenues_selection_schedules r
left outer join coverage_by_store c on r.id = c.store_id
order by name desc
limit 5
```

Code 8: SQL query for top 5 stores report.

As before, we were able to reduce the number of joins or lookups in our MongoDB implementation to only 4, by using embedding. The average runtime for 10 queries tested was 14.75 ms, similar as for our SQL query. Again, the more data we accumulate in our database over time, the better we expect our MongoDB query performance.

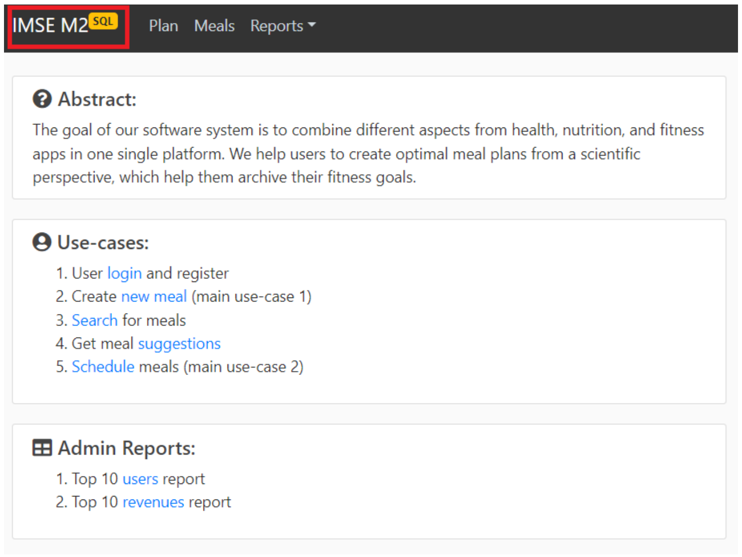
```
mongo_db['store'].aggregate([
    {
        '$lookup': {
            'from': 'meal',
            'localField': '_id',
            'foreignField': 'prices.store_id',
            'as': 'meals_by_store'
        }
    },
    {
        '$lookup': {
            'from': 'user',
            'localField': '_id',
            'foreignField': 'favorite_store',
            'as': 'users'
        }
    },
    {
        '$lookup': {
            'from': 'schedule',
            'pipeline': [{'$match': {'scheduled_on': {'$gte': datetime.date(2022, 1, 1), '$lte': datetime.date(2022, 1, 31)}}}],
            'localField': 'users._id',
            'foreignField': 'user_id',
            'as': 'schedules'
        }
    },
    {
        '$addFields': {
            'meal_coverage': {'$size': '$meals_by_store'},
            'users_selected': {'$size': '$users'},
            'scheduled_meals': {'$size': '$schedules'},
        }
    },
    {
        '$unwind': {
            'path': '$schedules', 'preserveNullAndEmptyArrays': True
        }
    },
    {
        '$lookup': {
            'from': 'meal',
            'localField': 'schedules.meal_id',
            'foreignField': '_id',
            'as': 'meals_by_schedule'
        }
    },
    {
        '$unwind': {
            'path': '$meals_by_schedule', 'preserveNullAndEmptyArrays': True
        }
    },
    {
        '$project': {
            '_id': 1,
            'name': 1,
            'meal_coverage': 1,
            'users_selected': 1,
            'scheduled_meals': 1,
            'revenues': {
                '$filter': {
                    'input': '$meals_by_schedule.prices',
                    'as': 'prices',
                    'cond': { '$eq': ['$$prices.store_id', '$_id'] }
                }
            },
        }
    },
    {
        '$unwind': {
            'path': '$revenues', 'preserveNullAndEmptyArrays': True
        }
    },
    {
        '$group': {
            '_id': '$_id',
            'name': {'$first': '$name'},
            'revenues_ccy': {'$sum': '$revenues.price'},
            'users_selected': {'$first': '$users_selected'},
            'scheduled_meals': {'$first': '$scheduled_meals'},
            'meal_coverage': {'$first': '$meal_coverage'},
        }
    },
    {'$sort': {'$name': -1}},
    {'$limit': 5}
])
```

Code 9: Mongodb aggregation pipeline for top 5 stores report.

A Appendices

A.1 Quick website tour

IMSE M2 gives a short introduction where to execute use-cases and reports. The “SQL” label indicates the current migration status.



Abstract:

The goal of our software system is to combine different aspects from health, nutrition, and fitness apps in one single platform. We help users to create optimal meal plans from a scientific perspective, which help them archive their fitness goals.

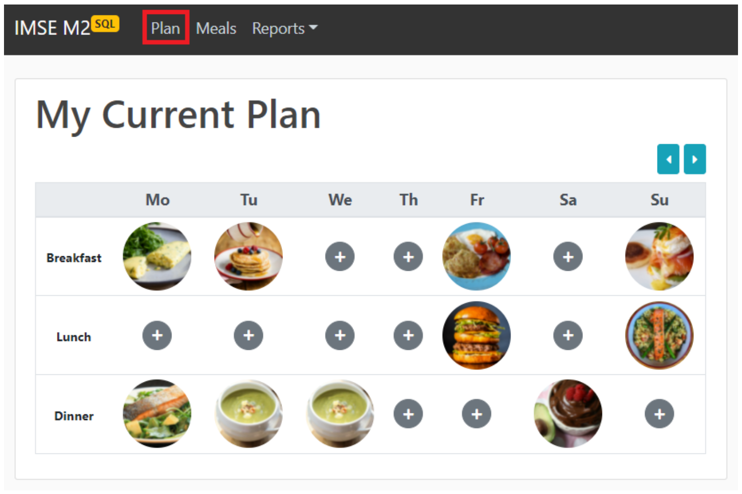
Use-cases:

1. User login and register
2. Create new meal (main use-case 1)
3. Search for meals
4. Get meal suggestions
5. Schedule meals (main use-case 2)

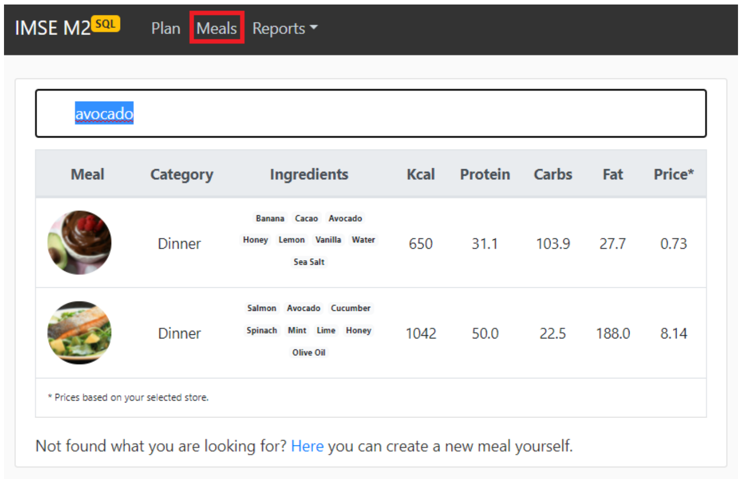
Admin Reports:

1. Top 10 users report
2. Top 10 revenues report

Plan summarizes schedules meals for a given week and list all respective aggregated ingredients. Here you can also add (by clicking on the “+” buttons) or remove meals (by clicking on the respective meal) of the current schedule.



Meals lists all meals available in our database with a few important summaries on calories, macros, and price. Above the table you can find a search field to filter meals by name, category or ingredients.

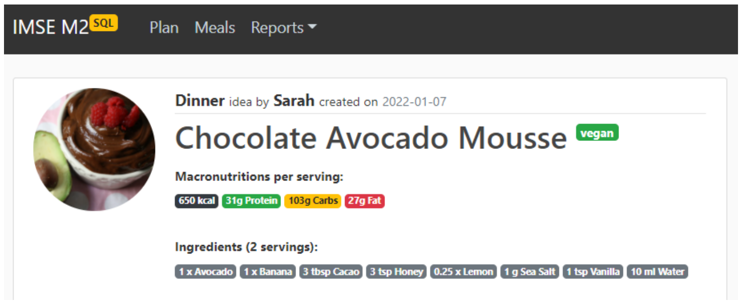


Meal	Category	Ingredients	Kcal	Protein	Carbs	Fat	Price*
	Dinner	Banana Cacao Avocado Honey Lemon Vanilla Water Sea Salt	650	31.1	103.9	27.7	0.73
	Dinner	Salmon Avocado Cucumber Spinach Mint Lime Honey Olive Oil	1042	50.0	22.5	188.0	8.14

* Prices based on your selected store.

Not found what you are looking for? Here you can create a new meal yourself.

By clicking on the meal image you will be redirected to a detailed meal overview. Moreover, at the bottom of the page you can add the meal to your plan.



Dinner idea by **Sarah** created on 2022-01-07

Chocolate Avocado Mousse vegan

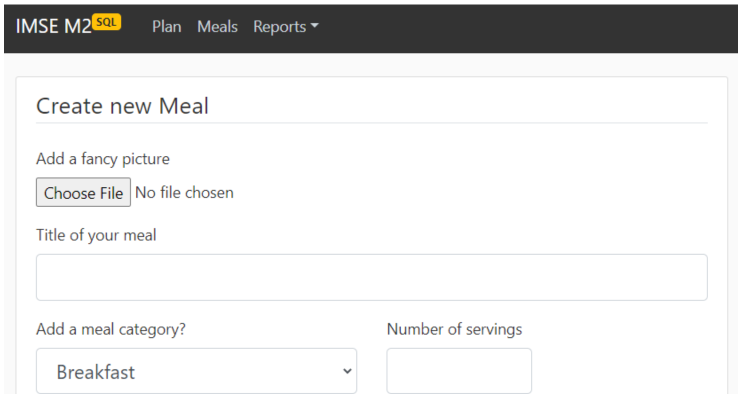
Macronutritions per serving:

650 kcal 31g Protein 103g Carbs 27g Fat

Ingredients (2 servings):

1 x Avocado 1 x Banana 3 tbsp Cacao 3 tsp Honey 0.25 x Lemon 1 g Sea Salt 1 tsp Vanilla 10 ml Water

Alternatively, you also have the option to create a new meal by clicking on "Here" right below the meals table, if you didn't find any meal you like.



Create new Meal

Add a fancy picture

No file chosen

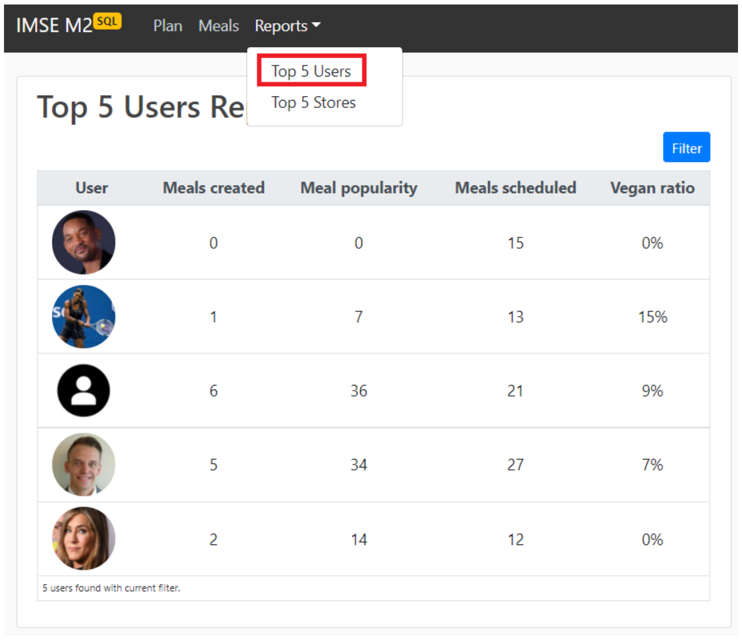
Title of your meal

Add a meal category?

Breakfast

Number of servings

Reports let you select either the user or store report, provided you are logged in as an admin. On the top-right you can click the "Filter" button to adjust the date range and sorted column to customize the report.

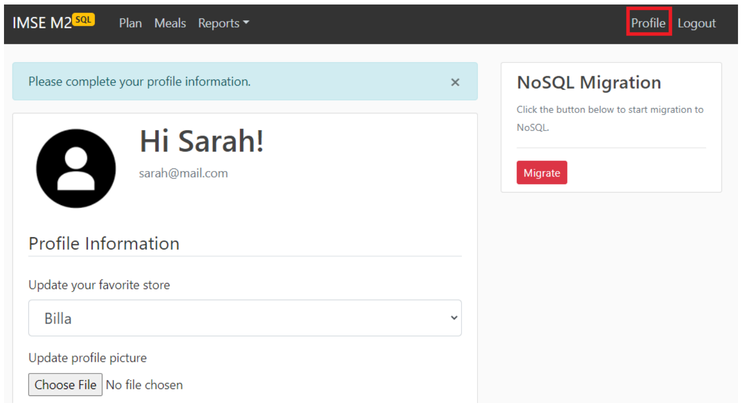


Top 5 Users Re

User	Meals created	Meal popularity	Meals scheduled	Vegan ratio
	0	0	15	0%
	1	7	13	15%
	6	36	21	9%
	5	34	27	7%
	2	14	12	0%

5 users found with current filter.

Profile summarizes the user information, like selected store, profile image, and further details about height, weight and fitness goals. If you want to change any profile information, you must commit you changes by the "Update" button.



Please complete your profile information. ×

Hi Sarah!

sarah@mail.com

Profile Information

Update your favorite store

Billa

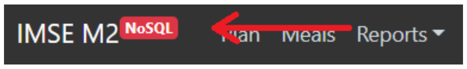
Update profile picture

No file chosen

NoSQL Migration

Click the button below to start migration to NoSQL.

Finally, in the right sidebar of an admins' profile page there is the option to migrate to NoSQL. After migration the label in the navigation bar should change to "NoSQL", but all functionalities outlined above remains the same.



A.2 Entity-relationship model

Figure 2 visualizes the entity-relationship model for this project. The only change compared to our design from M1 is the unary relationship, which moved from `GROCERY` to the `USER` entity.

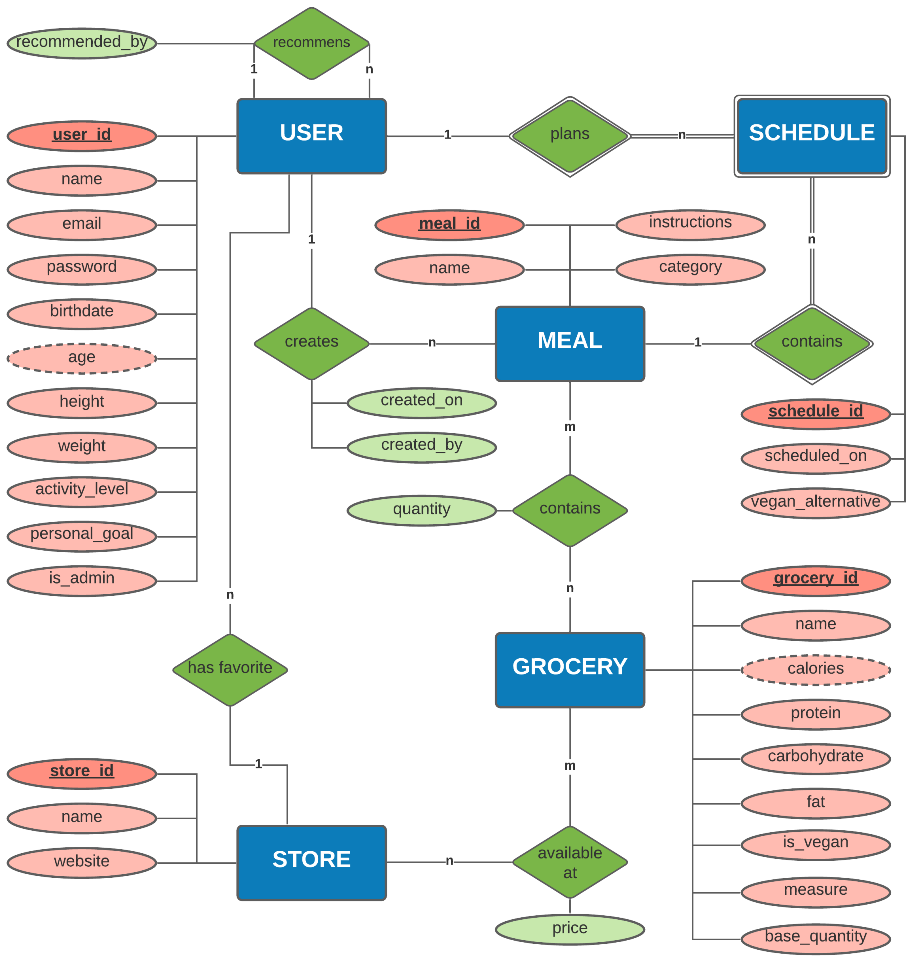


Figure 2: Entity-relationship model from M1. Entities in blue, relationships in green, and attributes in red (or light green), respectively.

A.3 Create table statements

Code 10 summarizes the create table statements to initialize our MySQL database.

```sql
CREATE TABLE store (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(30) NOT NULL,
    website VARCHAR(200),
    PRIMARY KEY (id),
    UNIQUE (name),
    UNIQUE (website)
);

CREATE TABLE user (
    id INTEGER NOT NULL AUTO_INCREMENT,
    username VARCHAR(20) NOT NULL,
    email VARCHAR(120) NOT NULL,
    password VARCHAR(120) NOT NULL,
    gender VARCHAR(1),
    birthday DATETIME,
    height INTEGER,
    weight INTEGER,
    activity_level VARCHAR(2),
    personal_goal VARCHAR(2),
    is_admin BOOLEAN,
    favorite_store INTEGER DEFAULT 1,
    recommended_by INTEGER,
    PRIMARY KEY (id),
    UNIQUE (username),
    UNIQUE (email),
    FOREIGN KEY(favorite_store) REFERENCES store (id)
);

CREATE TABLE meal (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(30) NOT NULL,
    instructions VARCHAR(2000) NOT NULL,
    category VARCHAR(30) NOT NULL,
    servings INTEGER NOT NULL,
    created_by INTEGER NOT NULL,
    created_on DATETIME DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    UNIQUE (name),
    FOREIGN KEY(created_by) REFERENCES user (id)
);

CREATE TABLE schedule (
    id INTEGER NOT NULL AUTO_INCREMENT,
    user_id INTEGER,
    meal_id INTEGER,
    scheduled_on DATE NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY(user_id) REFERENCES user (id) ON DELETE CASCADE,
    FOREIGN KEY(meal_id) REFERENCES meal (id) ON DELETE CASCADE
);

CREATE TABLE grocery (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(30) NOT NULL,
    protein FLOAT,
    carbohydrate FLOAT,
    fat FLOAT,
    is_vegan BOOLEAN,
    measure VARCHAR(20),
    quantity FLOAT,
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE TABLE ingredient (
    meal_id INTEGER NOT NULL,
    grocery_id INTEGER NOT NULL,
    quantity FLOAT NOT NULL,
    PRIMARY KEY (meal_id, grocery_id),
    FOREIGN KEY(meal_id) REFERENCES meal (id) ON DELETE CASCADE,
    FOREIGN KEY(grocery_id) REFERENCES grocery (id)
);

CREATE TABLE prices (
    grocery_id INTEGER NOT NULL,
    store_id INTEGER NOT NULL,
    price FLOAT NOT NULL,
    PRIMARY KEY (grocery_id, store_id),
    FOREIGN KEY(grocery_id) REFERENCES grocery (id),
    FOREIGN KEY(store_id) REFERENCES store (id)
);
```

Code 10: SQL create table statements.

A.4 Tree structure submission folder

We assume that the reader is familiar with the Flask framework and its folder structuring. In the root directory we only kept the most important files to start the application (i.e. Docker files, requirements and `run.py`). All other libraries and templates are in the `app` folder, see Figure 3.

```
imse_m2
├── app
│   ├── lib ................................. contains additional Python scripts
│   │   ├── database_functions.py
│   │   ├── forms.py
│   │   ├── initialization_functions.py
│   │   └── migration_functions.py
│   ├── sample_data ................. contains CSV files to initialize the database
│   │   ├── groceries.csv
│   │   └── ...
│   ├── static .................. contains custom CSS file, JS scripts, and images
│   │   ├── img .5 meals
│   │   │   │   ├── 1.jpg
│   │   │   │   └── ...
│   │   │   ├── stores
│   │   │   │   └── ...
│   │   │   └── users
│   │   │       └── ...
│   │   ├── main.css
│   │   └── meal_filter.js
│   ├── templates ............................. contains all HTML templates
│   │   ├── _layout.html
│   │   ├── home.html
│   │   └── ...
│   └── main.py
├── docker-compose.yml
├── Dockerfile
├── README.md
├── requirements.txt
└── run.py
```

Figure 3: Directory tree structure of M2 submission. Python files highlighted blue, Docker red, folders black, and other files gray.

A.5 Work protocol

Date	Hrs	Who	Category	Description
17.12.	2	LG/ST	Flask	- Implement basic app, i.e. first part of [2]. - Reading first two chapters of [1].
17.12.	2	LG/ST	Docker	- Watching tutorial from Moodle. - Install and run basic app using Docker.
17.12.	1	LG	Git	- Create Git repository with initial version.
18.12.	2	LG/ST	Brainstorming	- Decide which project to implement. - Discuss who implement which use-case and report. - Agree on time plan to meet M2 project deadline.
18.12.	6	LG/ST	Flask	- Start app development, i.e. tutorials 1-4 of [2].
19.12.	6	LG/ST	Flask	- Finalize blueprint version, i.e. tutorials 5-8 of [2].
20.12.	4	LG	Data	- Source sample data from [3] in `csv` files. - Add further details for groceries such as macros and prices.
26.12.	2	ST	SQLite	- Setup SQLite database with initial sample data.
26.12.	3	LG	Flask	- Finalize meal overview and individual meal pages.
27.12.	4	ST	Flask	- Add meal plan overview including use-case 2.
29.12.	5	LG	Flask	- Finalize meal plan overview from database with custom recommendations (use-case 4 and 5).
30.12.	3	LG	Flask	- Finalize last details with modal recommendations.
31.12.	2	ST	SQLite	- Implemented button to start SQLite filling process. - Adding further login and DB initialization checks.
31.12.	4	LG/ST	MongoDB	- Watching tutorial from Moodle. - Implement first sample database with `pymongo`.
01.01.	3	ST	Flask	- Finalize both SQL reports. - Implement migration button.
02.01.	4	LG	Docker	- Setup `.yml` file with `flask` and `mongo` as linked services.
03.01.	2	LG	Report	- Create LaTeXtemplate and start with first section. - Change unary relationship in ERD model.
07.01.	2	ST	MongoDB	- Create user, schedule, meal, grocery and store collections.
07.01.	1	ST	Flask	- Fix meal deletion, user recommendation and sidebar.
07.01.	1	LG	Report	- Finalize first report draft for RDBMS section.
09.01.	6	LG/ST	MongoDB	- Implement all database functions with MongoDB.
10.01.	2	ST	Flask	- Query from SQL or NoSQL based on migration status. - Testing all use-cases after migration using MongoDB.
11.01.	4	LG	MongoDb	- Trying to implement report 1 in MongoShell syntax.
14.01.	3	LG	MongoDb	- Finalizing report 1 and 2 in MongoShell syntax.
15.01.	5	ST	MongoDb	- Finalizing and testing MongoDB implementation. - Starting NoSQL design motivation report.
16.01.	1	LG	SSL	- Adding and testing SSL encryption.
16.01.	4	ST	Presentation	- Create PowerPoint with DB design, use-cases and reports.
16.01.	3	LG	MySQL	- Trying to switch from SQLite to MySQL. - Docker communication and initial test case worked.
17.01.	3	ST	Report	- Completing final draft of report.
17.01.	4	LG	MySQL	- Rewriting all functions to work with new framework. - Avoid any ORM functions for SQL and NoSQL.
18.01.	8	LG/ST	Submission	- Extensive UAT phase with final bug fixes. - Finalizing report and presentation sildes.

Table 1: Workload summary for M2 project.

References

[1] Grinberg, M., 2018. *Flask web development: developing web applications with python.* "O'Reilly Media, Inc.".

[2] Schafer, C., 2018. *Python Flask Tutorial: Full-Featured Web App.* Youtube.

[3] TheMealDB, 2021. *https://www.themealdb.com/.* JSON API.