Report on Phases - Recipe Website

group 32

MileStone 2, 2023S Information Management & Systems Engineering, 052400-1

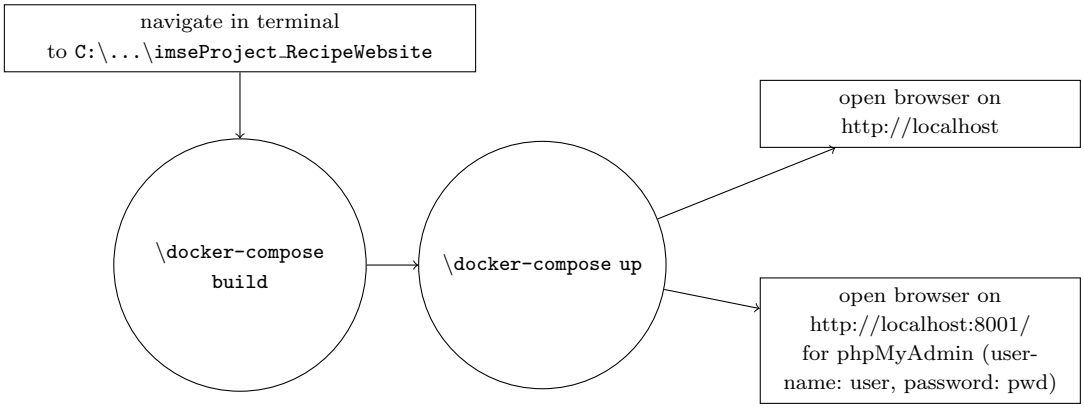
Mariia Ermeichuk (12243893) and Malte Klaes (01650623)

University of Vienna, 1010 Vienna, Austria

1 Project structure

1.1 Boot the Website

To start the application, go to the root directory of the project and enter the command 'docker-compose build' in the terminal. This will create the necessary front-end and back-end containers from the provided images. Once the build process is complete, you can launch the application by running 'docker-compose up'. When the commands have finished executing, open any web browser and enter "http://localhost/" in the address bar. Currently, there is no implementation of an HTTPS version with a certificate file. Once on the website you can push a green button 'Create MySQL Data', then login with one of the users and test all main use-cases and reports, which were described in M1 report.



1.2 Tech stack

The technical structure of the project is as follows: while docker provides the necessary images or the respective instances as containers (MySQL, MongoDB, nginx), the remaining stack can be divided into three parts, analogous to a classic software layer model and like it has also been implemented in the folder structure.
First nginx establishes the necessary connection to a browser and runs on port 80 as version "1.21.5-alpine". Underneath, PHP implements the small service and mainly the business logic and thus represents a LAMP-like approach. Finally, the persistence layer is MySQL as a table-based relational database management system and MongoDB as a document-based NoSQL database management system.
PHP was chosen because it is a fairly mature programming language and therefore fairly well-documented, is one of the most commonly used on the server side, and also uses a proven object-oriented approach that interfaces well with HTML.
As an alternative to nginx, Apache HTTP Server could also have been used to host our webpage, but nginx is currently (as of 2022) very popular webserver software or reverse proxy because of its speed and efficiency, so this solution was tried out in this project.

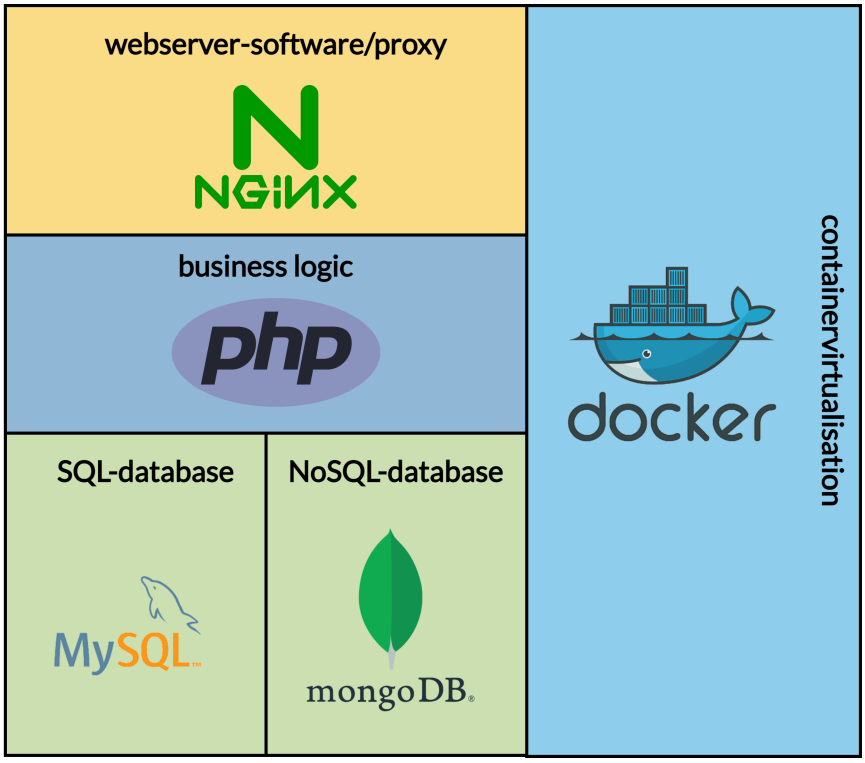


Fig. 1. Software stack diagram

1.3 Folder overview

Similar to the three-part software layer model, the code was also designed in an object-oriented manner. First of all, the index.php can be found outside by default, the "Dockerfile" organizes all the necessary library inputs and LayoutHTML organizes the headers of the respective webpages. The top of the layer-system build the externally visible GUI part ("01_UI"), which manages all webpages with a respective php file. A mixture of PHP and HTML can therefore be found throughout these files in this folder.
The second layer ("02_SERVICE") is the service layer, which, however, is very narrow here and does not control and regulate user input to the usual extent. Therefore, there is only one controllerer class in this folder.
The third layer ("03_PERSISTENCE") forms the main part of the project. On the one hand, this contains the business logic and thus access to the two database management systems. This part is therefore split into two parts, MySQL and MongoDB. Thus, both database management system worlds were cleanly separated from each other.

In order to dwell a little more on the main structure of the project, it is worth considering the third layer in more detail. The first half is the MySQL part, which organizes and initially prepares the basic data structure with *MySQLEntity* and the included folders. Initially, SQL tables are created here and filled with sample code. The folder *MySQLInserterHelper*, which creates the respective entity objects with its factory classes, which are later injected into the SQL tables, also helps here.

The second half is the MongoDB part, which initially provides the basic data structure with *MongoDBEntity* and other MongoDB flow classes (e.g. *MongoDBWriter.php, MongoDBInserter.php* or *MongoDBConnector.php*), analogous to the MySQL part. On the other hand, this folder contains the location where the migration from MySQL to MongoDB is carried out and thus contains some SQL-code.

1.4 Docker

Docker-compose file setup with its mapped (and exposed) **ports**:

- docker MySQL: 3306
- myPHPAdmin: 8001
- docker MongoDB: 27017
- frontend/browser: 80
- nginx (1.21.5-alpine): 80

In summary, the docker-compose file consists of two major parts. The first half represents the frontend area, whereby a docker nginx image with the instance (container-name) *"nginx_UI-recipeWebsite"* and a separate volume with the instance *"recipeWebsite_frontend_backend"* are created for this purpose. The other half is the backend, each with a mysql image provided by docker with the *"recipeWebstieSQL"* instance and a mongo image with the *"recipeWebsite_MongoDB"* instance.

```yaml
version: '3.6'

services:

  dbSQL:
    image: mysql:latest
    container_name: recipeWebsiteSQL
    environment:
      - MYSQL_USER=user
      - MYSQL_PASSWORD=pwd
      - MYSQL_ROOT_PASSWORD=pwd
      - MYSQL_DATABASE=RecipeWebsiteDB
      - MYSQL_ALLOW_EMPTY_PASSWORD=1
  phpmyadmin:
    depends_on:
      - dbSQL
    image: phpmyadmin/phpmyadmin
    ports:
      - "8001:80"
    environment:
      - PMA_HOST=dbSQL
      - PMA_PORT=3306
```

```yaml
  docker-mongo:
    platform: linux/amd64
    image: mongo:latest
    container_name: recipeWebsite_MongoDB
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: user
      MONGO_INITDB_ROOT_PASSWORD: pwd
      MONGO_INITDB_DATABASE: RecipeWebsiteMongoDB
    ports:
      - "27017:27017"

  frontend:
    build: ./frontend_backend
    container_name: recipeWebsite_frontend_backend
    restart: always
    expose:
      - 80
    volumes:
      - ./frontend_backend:/var/www/html/

  nginx:
    image: nginx:1.21.5-alpine
    container_name: nginx_UI-recipeWebsite
    restart: always
    ports:
      - "80:8000"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/conf.d/nginx.conf


volumes:
  mysql-data:
```

Fig. 2. docker-compose file with main parts: MySQL, MongoDB, frontend/nginx

2 Project phases

2.1 Phase 1 (RDBMS design): MySQL database

Compared with M1 we have made only one change in ER-model. We decided to remove the attribute "recipeMedia" in Recipe, because it does not form the logic of the model, but the generation of pictures and photos for the recipe could cause additional problems in the development process. The final ER-model is depicted on Figure 3.

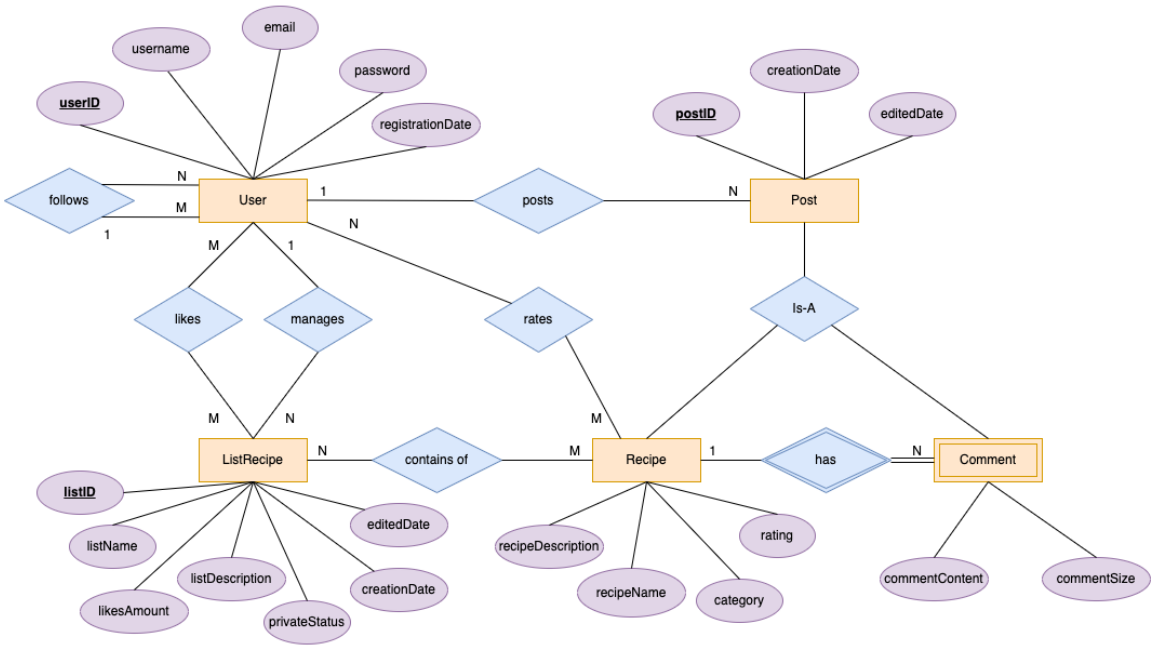


Fig. 3. ER Model of the Recipe Website

2.1.1 Logical Design

We implemented the first part of the project using MySQL. The database was designed this way:

- for each entity from Er Model in Figure 3 (*User, Post, Recipe, Comment, ListRecipe*) we created a table in MySQL;
- for each many-to-many connection we implemented a connecting table (*UserRatesRecipe, User LikesRecipe, userLikesListRecipe, userFollow, listContainsOfRecipe*), by referencing the primary keys of each associated entity.

PHP was employed to create classes for each table. Additionally, specialized classes, *UserID, PostID, and listID*, were constructed to facilitate the generation of respective database keys. Overall, this phase involved the creation of nine tables in MySQL and the development of twelve PHP classes, see in Figure 4.

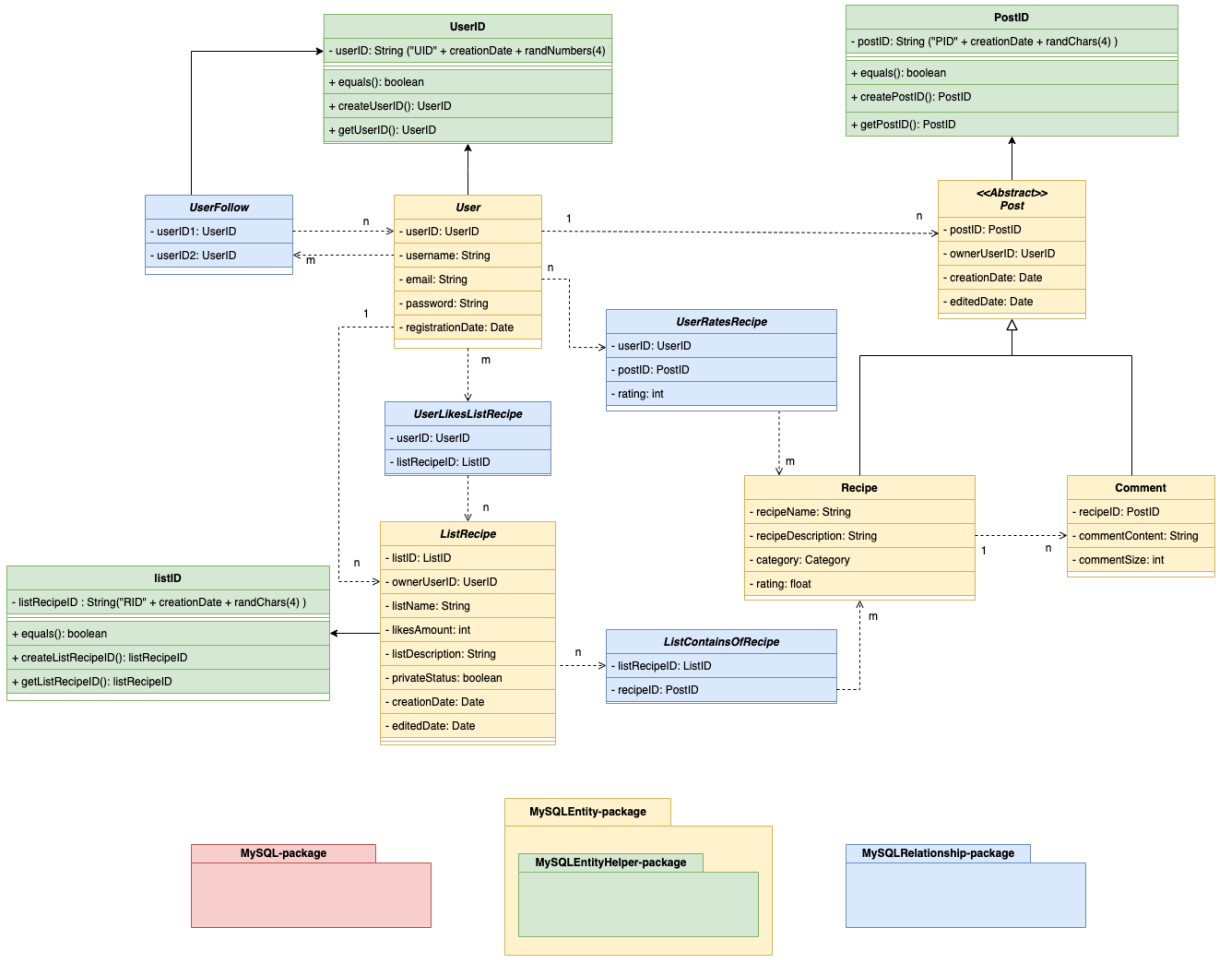


Fig. 4. Classes Model for Relational Database

In order to optimize the querying process, our focus was primarily on using IDs for record retrieval, rather than implementing additional keys for unique fields such as usernames. Although logically it is necessary for usernames to be unique, we deemed it sufficient to rely on ID-based queries.
To generate the IDs for the database records, custom functions in additional classes were implemented. These functions took a string (e.g., "UID" for user, "PID" for post, and "RID" for listRecipe) and combined it with the precise creation date (including day, month, year, hour, second, and millisecond). Furthermore, random numbers or characters were appended to ensure uniqueness. These generated IDs were then stored within the MySQL database.

2.1.2 Physical Design

The logical design, as presented earlier, served as the foundation for the physical implementation. In this phase, we translated the logical design into a physical representation within the MySQL database.

```
CREATE TABLE user (
    userID VARCHAR(255) PRIMARY KEY NOT NULL,
    username VARCHAR(255) NOT NULL,
    email VARCHAR(255),
    pwd VARCHAR(255) NOT NULL,
    registrationDate DATETIME NOT NULL);

CREATE TABLE recipe (
    postID VARCHAR(255) PRIMARY KEY NOT NULL,
    recipeName VARCHAR(255) NOT NULL,
    recipeDescription TEXT,
    category VARCHAR(255),
    rating FLOAT,
    ownerUserID VARCHAR(255) NOT NULL,
    creationDate DATETIME NOT NULL,
    editedDate DATETIME);

CREATE TABLE comment (
    postID VARCHAR(255) PRIMARY KEY NOT NULL,
    commentContent TEXT,
    ownerUserID VARCHAR(255) NOT NULL,
    recipeIDReference VARCHAR(255),
    creationDate DATETIME NOT NULL,
    editedDate DATETIME);

CREATE TABLE listRecipe (
    listID VARCHAR(255) PRIMARY KEY NOT NULL,
    ownerID VARCHAR(255) NOT NULL,
    listName VARCHAR(255) NOT NULL,
    likesAmount INT,
    listDescription TEXT,
    privateStatus BOOLEAN NOT NULL,
    creationDate DATETIME NOT NULL,
    editedDate DATETIME);

CREATE TABLE userRatesRecipe (
    userID VARCHAR(255) NOT NULL,
    postID VARCHAR(255) NOT NULL,
    rating INT NOT NULL);

CREATE TABLE userLikesListRecipe (
    userID VARCHAR(255) NOT NULL,
    listID VARCHAR(255) NOT NULL);

CREATE TABLE userFollow (
    user1ID VARCHAR(255) NOT NULL,
    user2ID VARCHAR(255) NOT NULL);

CREATE TABLE listContainsOfRecipe (
    listID VARCHAR(255) NOT NULL,
    recipeID VARCHAR(255) NOT NULL);
```

2.1.3 Data import

The website's start page provides users with the functionality to fill in the database with sample data and also to erase this data. This feature is accessible when the website is launched in a web browser. By clicking the "Create MySQL Data" button, users can generate new data entries in the MySQL database. This action can be repeated multiple times, with each click resulting in the creation of fresh data.

After data generation the SQL database contains sufficient information to work with main use-cases and reports. Users have the option to select one of the generated users and log in to the system. This login functionality gives access to main use-cases and reports, and the ability to convert data to MongoDB if desired.

Fig. 5. Start page of the Website

2.2 Phase 2 (NoSQL Design): MongoDB database and migration

The migration to MongoDB was initially achieved by significantly reducing the tables and the current documents. While MySQL previously had a total of eight tables, four entity-tables and four relational tables, here you can get by with exactly half of the translated documents while the relational tables are incorporated into the entities: *user_collection, recipe_collection, comment_collection* and *listRecipe_collection*.

2.2.1 MongoDB collections

- user_collection
- recipe_collection
- comment_collection
- listRecipe_collection

If you compare the respective class diagrams for preparing the creation of the MySQL tables or the creation of the MongoDB documents, you can clearly see the reduction in table-amount and the elimination of "joins".

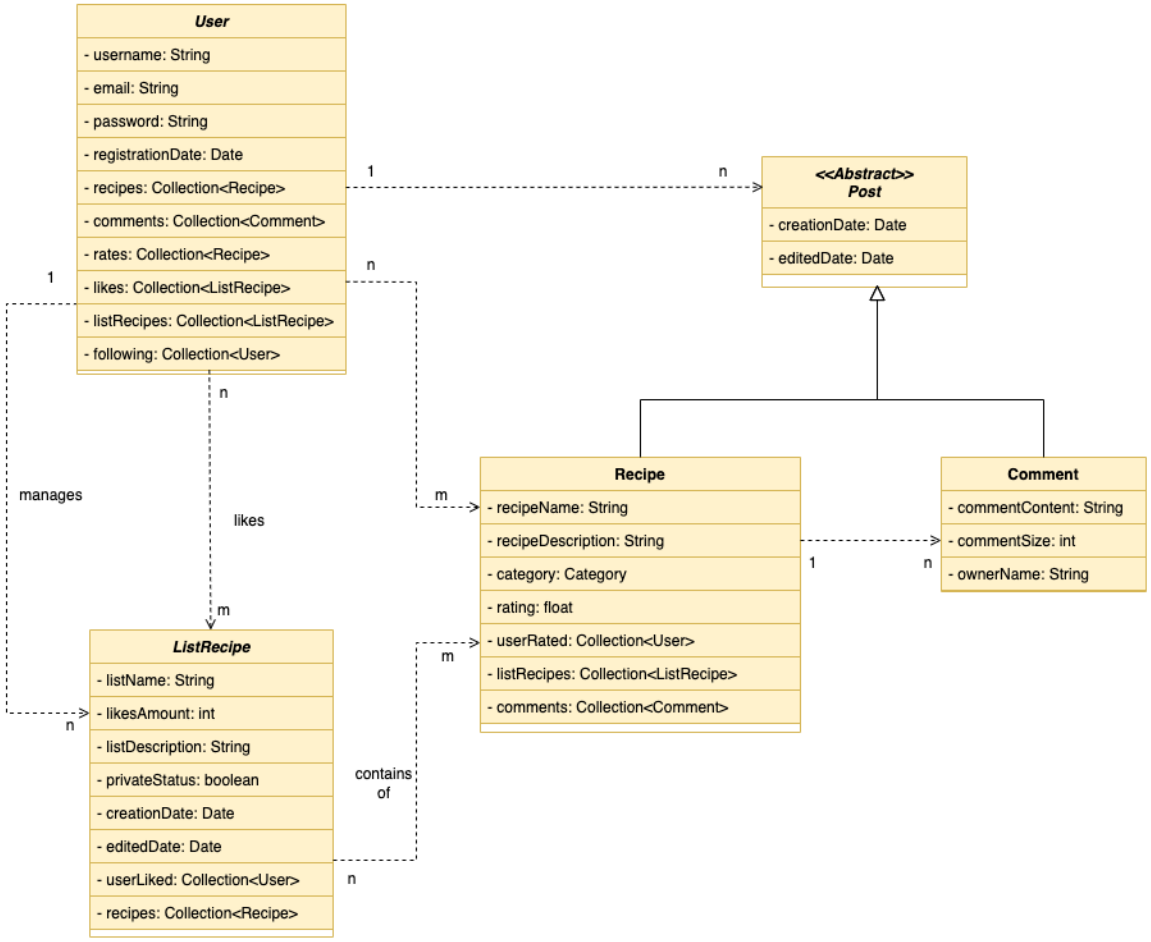


Fig. 6. Classes Model for NoSQL Database

During the migration, which first translate MySQL-data to MongoDB and secondly swiped out all MySQL-data for the rest of the run, two essential translations were implemented from the table-based logic to the document-based logic in the sense of migration:

(1) Instead of **IDs** (such as UID, PID, RID), explicitly written IDs were not used here, since they are automatically provided by MongoDB in the form of **"oid"**. On the other hand, many auxiliary tables such as *"userLikesListRecipe", "userRatesRecipe"* or *"listContainsOfRecipe"* have been incorporated into the respective documents. Therefore, the importance of IDs decreases, but does not make them disappear completely. In many searches, a search using ID (or oid) was dispensed with and instead searched using data-specific characteristics (in concrete terms: when searching in MongoDB, it is relied on that an intersection of e.g. the same listname, listDescription, creationDate and editedDate almost excluded and therefore identity is provided on a data-specific way).

(2) As already mentioned in (1), the **auxiliary tables** *"userLikesListRecipe", "userRatesRecipe", "userFollow"* and *"listContainsOfRecipe"* were integrated into the documents using arrays or generally collection-like data structures, which reduced the implementation effort in the individual searches (see main usecases and reports) is significantly simplified.

In the following part, there will be a specification with types and example for MongoDB-design for each entity (user, recipe, comment, listRecipe).

2.2.2 MongoDB entity: user

```
stdClass Object
(
    [_id] => MongoDB\BSON\ObjectId Object
        (
            [oid] => 64940838593e4e22c9066d63
        )

    [username] => string (davidtaylor)
    [email] => string (davidtaylor@example.com)
    [pwd] => string (pwd)
    [registrationDate] => DATE (2016-09-13 21:41:47)
    [comments] => Array
        (
            [0] => stdClass Object
                (
                    [commentContent] => string (the texture was off)
                    [ownerName] => string (davidtaylor)
                    [creationDate] => DATE (2016-05-09 05:49:58)
                    [editedDate] => DATE (2016-05-09 06:20:19)
                )

        )

    [recipes] => Array
        (
            [0] => stdClass Object
                (
                    [recipeName] => string (stir-fry)
                    [recipeDescription] => string (vegetables, meat, sauce)
                    [category] => string (Stir-Fry)
                    [rating] => INT (3)
                    [comment] => Array
                        (
                            [0] => stdClass Object
                                (
                                    [commentContent] => string (made me want seconds)
                                    [ownerName] => strin (cookiecraver)
                                    [creationDate] => DATE (2013-02-16 16:53:15)
                                    [editedDate] => DATE (2013-02-17 07:06:18)
                                )

                        )

                    [creationDate] => DATE (2014-10-18 16:10:01)
                    [editedDate] => DATE (2014-10-19 13:36:28)
                )

        )
```

```
49
50      [listRecipes] => Array
51          (
52              [0] => stdClass Object
53                  (
54                      [listName] => string (vegetarian delights)
55                      ...
56                      [recipes] => Array
57                          (
58                              [0] => stdClass Object
59                                  (
60                                      [recipeName] => string (smoothie)
61                                      ...
62                                  )
63                          )
64                      ...
65                  )
66
67          )
68
69      [listRecipesLiked] => Array
70          (
71              [0] => stdClass Object
72                  (
73                      [listName] => string (quick and easy recipes)
74                      [listDescription] => string (recipes that are quick to prepare)
75                      [privateStatus] => INT (0)
76                      [creationDate] => DATE (2010-04-03 11:10:00)
77                      [editedDate] => DATE (2010-04-03 15:11:21)
78                  )
79
80          )
81
82      [ratedRecipes] => Array
83          (
84              [0] => stdClass Object
85                  (
86                      [recipeName] => string (pancakes)
87                      [recipeDescription] => string (flour, milk, eggs, sugar)
88                      [category] => string (Breakfast)
89                      [creationDate] => DATE 2019-10-20 17:11:57
90                      [editedDate] => DATE 2019-10-21 00:13:09
91                  )
92
93          )
94
95      [following] => Array
96          (
97              [0] => stdClass Object
98                  (
99                      [username] => string (davidtaylor)
100                     ...
101                     [following] => [none]
102         )
103 )
```

A user no longer has a UID, but instead an automatically created oid and six arrays: *[comments], [recipes], [listRecipes], [listRecipesLiked], [ratedRecipes], [following].* All included entities were not fully included, but care was taken to include creationDate and editedDate in order to guarantee/achieve object uniqueness in a search.

Most interesting, however, is the [following] - array: Potentially recursive data structures can arise here and completely destroy a realistic JSON structure. Therefore it was decided to show only the first layer of acquaintances and not to show another [following] - array in the [following] - array ([following] = [none]).

2.2.3 MongoDB entity: recipe

```
stdClass Object
(
    [_id] => MongoDB\BSON\ObjectId Object
        (
            [oid] => 64940838593e4e22c9066e34
        )

    [recipeName] => string (steak)
    [recipeDescription] => string (beef steak, marinade, grill)
    [category] => string (Steak)
    [rating] => INT (4)
    [comments] => Array
        (
            [0] => stdClass Object
                (
                    [commentContent] => string (a bit too spicy)
                    [ownerName] => string (aidenwilson)
                    [creationDate] => DATE (2018-12-29 08:03:38)
                    [editedDate] => DATE (2018-12-29 16:42:44)
                )

        )

    userRated] => Array
        (
            [0] => stdClass Object
                (
                    [username] => string (laurenscott)
                    [rating] => INT (4)
                )

        )

    [listRecipe] => Array
        (
            [0] => stdClass Object
                (
                    [listName] => string (hearty soups)
                )

        )

    [creationDate] => DATE (2010-06-19 20:40:41)
    [editedDate] => DATE (2010-06-20 01:43:42)
)
```

2.2.4 MongoDB entity: comment

```
stdClass Object
(
    [_id] => MongoDB\BSON\ObjectId Object
        (
            [oid] => 64940838593e4e22c9066d76
        )

    [commentContent] => string (the texture was brilliant)
    [ownerName] => string (davidtaylor)
    [creationDate] => DATE (2016-05-09 05:49:58)
    [editedDate] => DATE (2016-05-09 06:20:19)
)
```

Recipes and comments have been migrated similar to users. Here, their common PID (formerly implemented in the Post interface) has been omitted in favor of auto-generated oids. Helper classes have been integrated into the documents as arrays and creationDate and editedDate, among other document fields, ensure uniqueness. In terms of the index, it would have been more helpful to save the comment-length directly as a field in the comments. However, the calculation of this is not a big effort. On the other hand, the field rating in recipe is helpful in the sense of the index for later reports, as are the creationDates in each collection.

2.2.5 MongoDB entity: listRecipe

```
stdClass Object
(
    [_id] => MongoDB\BSON\ObjectId Object
        (
            [oid] => 64940838593e4e22c9066e94
        )

    [listName] => string (hearty soups)
    [listDescription] => string (hearty soups for cold days)
    [privateStatus] => INT (1)
    [recipes] => Array
        (
            [0] => stdClass Object
                (
                    [recipeName] => string (pancakes)
                    [recipeDescription] => string (flour, milk, eggs, sugar)
                    [category] => string (Breakfast)
                    [rating] => DOUBLE (3.5)
                    [comment] => Array
                        (
                            [0] => stdClass Object
                                (
                                    [commentContent] => string (best recipe ever)
                                    [ownerName] => string (aidenwilson)
                                    [creationDate] => DATE (2010-06-30 17:35:48)
                                    [editedDate] => DATE (2010-07-01 02:50:03)
                                )

                        )

                    [creationDate] => DATE (2019-10-20 17:11:57)
                    [editedDate] => DATE (2019-10-21 00:13:09)
                )

        )

    [likes] => INT (3)
    [creationDate] => DATE (2018-01-01 11:08:14)
    [editedDate] => DATE (2018-01-02 04:44:45)
)
```

Finally, ListRecipes show the same migration pattern as the three documents already described: no more RID but therefor an oid, recipes are incorporated directly as an array, creationDate and editeDate ensure (practical, not theoretical) uniqueness with the other document fields. It should also be mentioned here that the "recipeDescription" field means the actual recipe ingredients, while listDescription is more of a description of the "cookbook".
The field likes in listRecipe is helpful in terms of the index for the second report, as is the creationDate.

3 Main usecases and reports

3.1 Main Use-Case 1: "Add recipe to a recipe list" (Mariia E.)

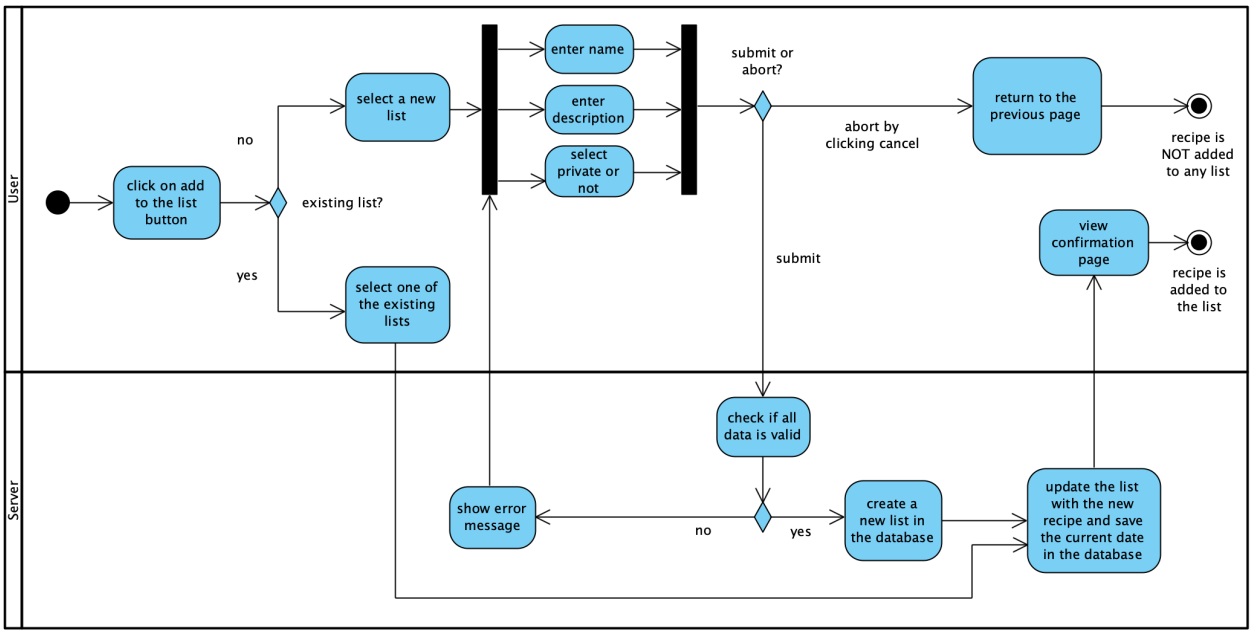


Fig. 7. Activity Diagram: add recipe to a recipe list

3.1.1 Procedure for MySQL

The activity diagram (Figure 7) illustrates that in the main scenario the result of the use-case involves adding a selected recipe to an existing listRecipe. In the MySQL database, the following code is used to insert the relevant IDs into the connecting table, *listContainsOfRecipe*. This table establishes the relationship between the existing list and the chosen recipe, allowing the user to add the recipe to the selected list.

```
1     INSERT INTO listContainsOfRecipe(listID, recipeID)
2               VALUES ('". $listID ."','". $recipeID . "');
```

where *listID* is the ID of the selected list and *recipeID* is the ID of the selected recipe, which has to be added to the list.

In an alternative scenario, the user may choose to add a recipe to a new list. In such cases, it becomes necessary to create a new list and add it to the *ListRecipe* table within the MySQL database. The following code is employed to insert a new listRecipe into the table, with the values provided by the user for the respective fields.

```
1  INSERT INTO listRecipe(listID, ownerID, listName, likesAmount, listDescription,
2      privateStatus, creationDate, editedDate)
3  VALUES ('". $exampleListRecipe->getListID()."','".
4          $exampleListRecipe->getOwnerUserID()."','".
5          $exampleListRecipe->getListName()."','".
6          $exampleListRecipe->getLikesAmount()."','".
```

```
        $exampleListRecipe->getListDescription()."','".
        (($exampleListRecipe->getPrivateStatus()) ? 1 : 0)."','".
        date('YmdHis', strtotime($exampleListRecipe->getCreationDate()))."', '" .
        date('YmdHis', strtotime($exampleListRecipe->getEditedDate())) . "');
```

where *exampleListRecipe* is the new listRecipe with the field values taken from the user input and all get methods return the value stored in the corresponding field of the recipe entity.

Default value of *likesAmount* field of the new listRecipe is zero, both dates are set automatically to the current date, *ownerID* is set as the ID of the current user, while all other fields (*listName, listDescription, privateStatus*) are mandatory. If user leaves them empty, he is asked to fill the form one more time.

3.1.2 Procedure for MongoDB

When updating a listrecipe with a new recipe, a possible object is first searched for in the respective collection using a cursor (a kind of iterator). If nothing was found, an empty array is created and a new recipe is copied into it, otherwise the old array is adopted and finally the respective collection is updated with a filter query for the respective oid with a final executeBulkWrite by an instantiated manager.

```
public function insertRecipeToList($recipe, $listRecipe){
        ...

        $manager = new MongoDB\Driver\Manager("mongodb://user:pwd@recipeWebsite_MongoDB:27017");
        $filter = ['_id' => new MongoDB\BSON\ObjectId($listRecipeId)];
        $query = new MongoDB\Driver\Query($filter);
        $cursor = $manager->executeQuery('RecipeWebsiteMongoDB.listrecipe_collection', $query);


        if ($cursor->isDead()) {
            echo 'The listRecipe-document with the specified ObjectId was not found!';
        } else {
            $listRecipe = $cursor->toArray()[0];

            ...

            $newRecipe = [
                'recipeName' => $recipe[0],
                'recipeDescription' => $recipe[1],
                'category' => $recipe[2],
                'rating' => $recipe[3],
                'comment' => $commentArray,
                'creationDate' => $recipe[4],
                'editedDate' => $recipe[5],
            ];

            $listRecipe->recipes[] = $newRecipe;

            // update recipes listrecipe
            ...

            $filter = ['_id' => new MongoDB\BSON\ObjectId($listRecipeId)];
            $update = ['$set' => ['recipes' => $listRecipe->recipes]];
            $bulk = new MongoDB\Driver\BulkWrite();
            $bulk->update($filter, $update);
            $manager->executeBulkWrite('RecipeWebsiteMongoDB.listrecipe_collection', $bulk);
        }
}
```

3.2 Main Use-Case 2: "Create new recipe" (Malte K.)

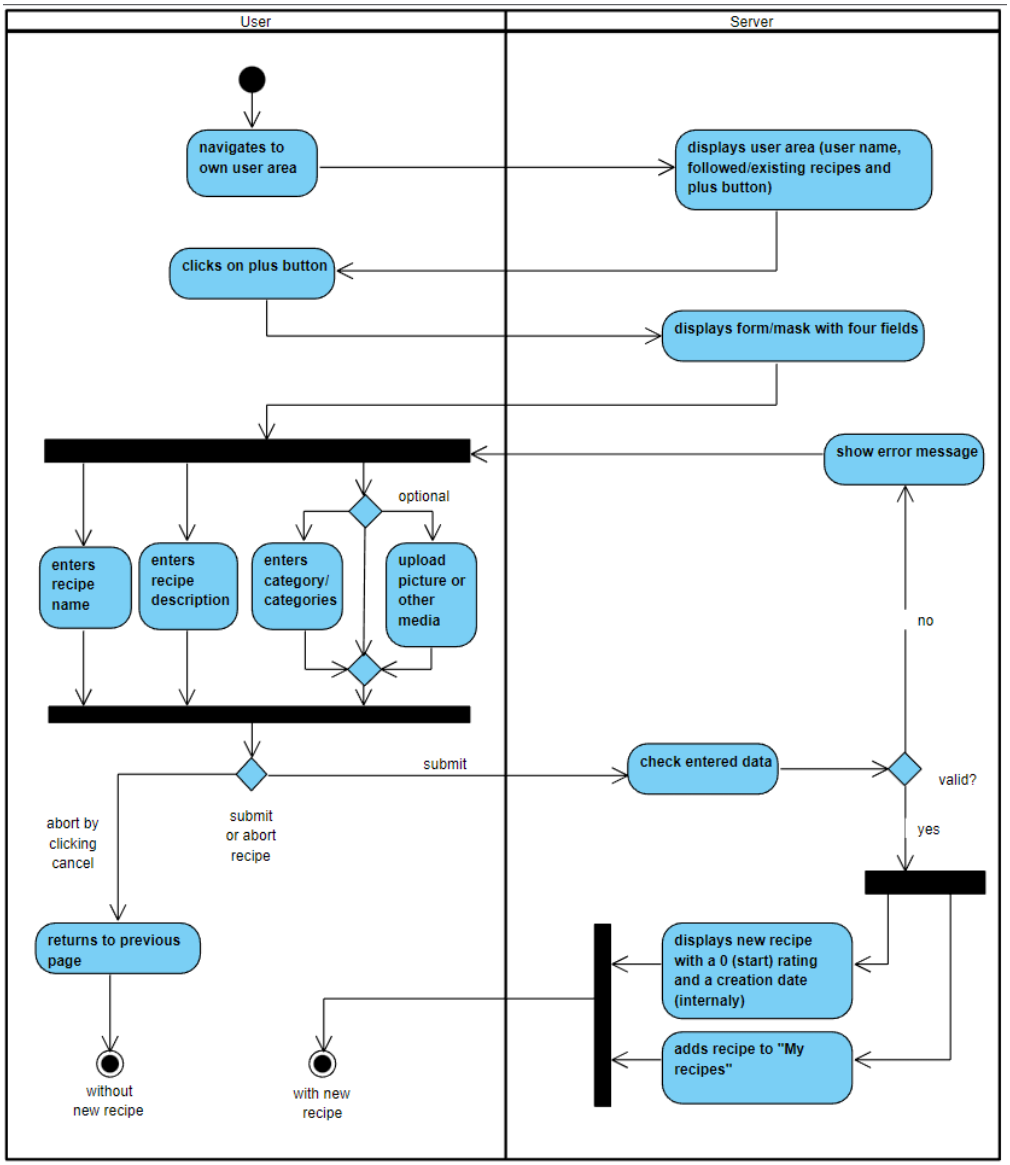


Fig. 8. Activity Diagram: create new recipe

3.2.1 Procedure for MySQL

The activity diagram (Figure 8) depicts that in the main scenario of the use-case a new recipe should be added to the database. This involves capturing the values entered by the user into the form on the website (excluding the media field, as it has been removed). The following code was implemented, which inserts a new recipe with the user-provided values into the Recipe table within the MySQL database.

```
1    INSERT INTO recipe(postID, recipeName, recipeDescription, category, rating, ownerUserID,
2    creationDate, editedDate)
```

```
        VALUES ('". $exampleRecipe->getPostID()."','".
        $exampleRecipe->getRecipeName()."','".
        $exampleRecipe->getRecipeDescription()."','".
        $exampleRecipe->getCategory()."','".
        $exampleRecipe->getRating()."','".
        $exampleRecipe->getOwnerUserID()."','".
        date('YmdHis', strtotime($exampleRecipe->getCreationDate()))."', '" .
        date('YmdHis', strtotime($exampleRecipe->getEditedDate())) . "');
```

where *exampleRecipe* is the new recipe with the field values taken from the user input and all get methods return the value stored in the corresponding field of the recipe entity.
Default value of *rating* field of the new recipe is zero, both dates are set automatically to the current date, *ownerUserID* is set as the ID of the current user, while all other fields (*recipeName, recipeDescription, category*) are mandatory. If user leaves them empty, he is asked to fill the form one more time.

3.2.2 Procedure for MongoDB

Similar to listRecipe, a new recipe can be added very easily (in 5 steps). First, a manager is created, then an empty recipe document (array) is prepared, filled with the actual data that was previously collected and stored in a PHP object, and finally inserted into the corresponding recipe_collection with BulkWrite.

```

public function insertRecipe($recipeObject){
    //* [1] create manager
    $manager = new MongoDB\Driver\Manager("mongodb://user:pwd@recipeWebsite_MongoDB:27017");

    //* [2] initialize the recipeDocs array
    $recipeDocs = [];

    //* [3] for every recipe, create a new document and add it to the array
    $recipeDoc = [
        'recipeName' => $recipeObject->getRecipeName(),
        'recipeDescription' => $recipeObject->getRecipeDescription(),
        'category' => $recipeObject->getCategory(),
        'rating' => $recipeObject->getRating(),
        'comments' => $recipeObject->getRawCommentsArray(),
        'userRated' => array(),
        'listRecipe' => array(),
        'creationDate' => $recipeObject->getCreationDate(),
        'editedDate' => $recipeObject->getEditedDate(),
    ];
    $recipeDocs[] = $recipeDoc;


    //* [4] insert all recipe documents into MongoDB
    $bulkWrite = new MongoDB\Driver\BulkWrite();
    foreach ($recipeDocs as $recipeDoc) {
        $bulkWrite->insert($recipeDoc);
    }

    //* [5] write all data to the database and the specific collection
    $manager->executeBulkWrite($this->dbnameMongo . '.recipe_collection', $bulkWrite);
}
```

3.3 Top rated recipes by a category (Mariia E.)

3.3.1 Theoretical outline

[A] **Description:** The report finds 10 top rated recipes of a specific category.

$$ratesNumber := \text{number of people who rated the recipe} \quad (1)$$
$$rating := \text{rating stored in the database} \quad (2)$$
$$\boldsymbol{newRating} = \frac{rating * ratesNumber + 5 + 1}{5 * ratesNumber + 5 + 5} \quad (3)$$

[B] **Entities:** User, ListRecipe, Recipe, Post
[C] **Sorted by:** newRating, number of ListRecipes
[D] **Filtered by:** category
[E] **Columns:** "recipeName", "creationDate Post", "rating", "newRating", "number of ListRecipes"

3.3.2 Procedure for MySQL

The following code represents MySQL procedure specifically designed to calculate the essential data required for report explained above.
Method **calculateTopRecipesReport1()** sorts the result of method **retrieveAllRecipesReport1()** by first two parameters which are value of *newRating* and number of listRecipes to which the recipe was added. Method **retrieveAllRecipesReport1()** calculates the value of *newRating* based on *rating* which is stored in recipe table, and on *newNumber*, which is the result of method **retrieveNumUsersRatedRecipe()**. This method calculates the result based on one SQL query. For sorting the result also needed the number of listRecipes, to which recipe is added, so this value is calculated by method **retrieveNumUsersRatedRecipe()**. Finally, *category* is used as a filter for this report, so all categories are retrieved by method **retrieveAllCategories()** based on SQL query.

```
public function calculateTopRecipes_Report1($myUserID, $category) {
            $resultArray = $this->retrieveAllRecipes_Report1($myUserID, $category);

            usort($resultArray, function($a, $b) {
                // Compare the first field of the inner array
                $firstComparison = $b[0] <=> $a[0];
                // If the first fields are equal, compare the second field
                if ($firstComparison === 0) {
                    return $b[1] <=> $a[1];
                }
                return $firstComparison;
            });
            return $resultArray;
        }

        private function retrieveAllRecipes_Report1($myUserID, $category) {
            $result = array();
            if ($category == null) {
                $sql = "SELECT * FROM recipe";
                $dataBaseConnection= new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB",
                "user", "pwd");
                $stmt = $dataBaseConnection->prepare($sql);
```

```
        }
        else {
            $sql = "SELECT * FROM recipe WHERE category = :category";
            $dataBaseConnection= new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB",
            "user", "pwd");
            $stmt = $dataBaseConnection->prepare($sql);
            $stmt->bindParam(':category', $category);
        }
        $stmt->execute();

        while ($userData = $stmt->fetch(PDO::FETCH_ASSOC)) {
            $newRating = 0;
            $recipeToResult = array();
            $recipe = $userData['postID'];
            $rating = $userData['rating'];
            $ratesNumber = $this->retrieveNumUsersRatedRecipe($recipe);
            $newRating = ($rating * $ratesNumber + 5 + 1) / (5 * $ratesNumber + 5 + 5);
            $listRecipeNumber = $this->retrieveNumListRecipesWithRecipe($recipe);

            array_push($recipeToResult, round($newRating, 2));
            array_push($recipeToResult, $listRecipeNumber);

            if($userData['ownerUserID'] === $myUserID){
                array_push($recipeToResult, $userData['recipeName'] . " <font color=#f56342><b>
                (myRecipe) </b></font>");
            }
            else {
                array_push($recipeToResult, $userData['recipeName']);
            }
            $result[] = $recipeToResult;
        }
        return $result;
    }

    private function retrieveNumListRecipesWithRecipe($recipeID) {
        $sql = "SELECT count(*) as cnt FROM listContainsOfRecipe WHERE recipeID = :recipeID";
        $dataBaseConnection= new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB",
        "user", "pwd");
        $stmt = $dataBaseConnection->prepare($sql);
        $stmt->bindParam(':recipeID', $recipeID);
        $stmt->execute();

        return ($userData = $stmt->fetch(PDO::FETCH_ASSOC)) ? $userData['cnt'] : 0;
    }

    private function retrieveNumUsersRatedRecipe($postID) {
        $sql = "SELECT count(*) as cnt FROM userRatesRecipe WHERE postID = :postID";
        $dataBaseConnection= new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB",
        "user", "pwd");
        $stmt = $dataBaseConnection->prepare($sql);
        $stmt->bindParam(':postID', $postID);
        $stmt->execute();
```

```
75
76              return ($userData = $stmt->fetch(PDO::FETCH_ASSOC)) ? $userData['cnt'] : 0;
77          }
78
79          public function retrieveAllCategories() {
80              $result = array();
81              $sql = "SELECT Distinct(category) as cat FROM recipe";
82              $dataBaseConnection= new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB", "user", "pwd");
83              $stmt = $dataBaseConnection->prepare($sql);
84              $stmt->execute();
85
86              while ($userData = $stmt->fetch(PDO::FETCH_ASSOC)) {
87                  $cat = $userData['cat'];
88                  array_push($result, $cat);
89              }
90
91              return $result;
92          }
93
```

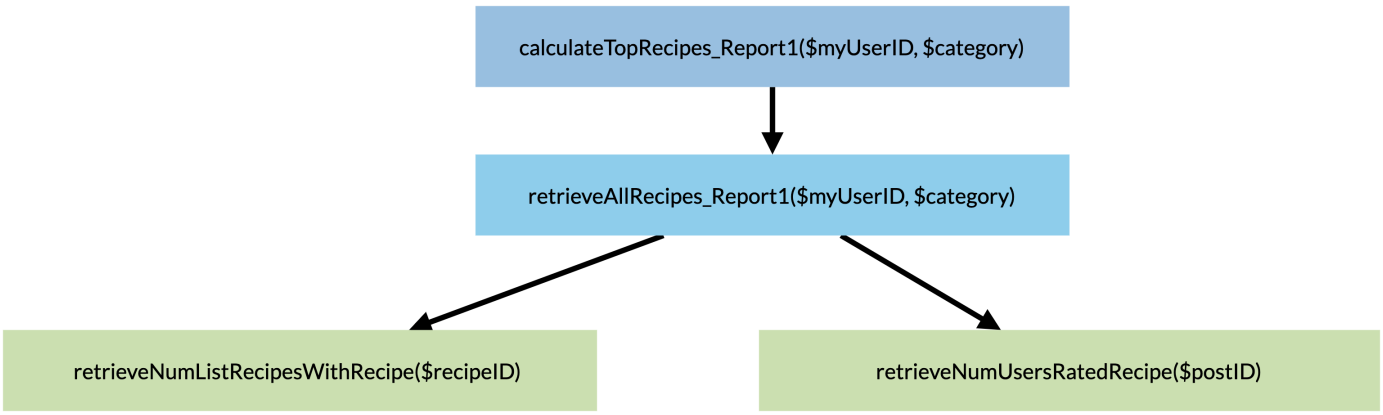


Fig. 9. call graph for report 1 in MySQL

3.3.3 Procedure for MongoDB

For MongoDB there is also a method **calulateTopRecipeReport1()** which sorts the result by the first two parameters, however all the calculations are made with one method **retrieveAllRecipesReport1()** needing method **definerecipeName()** to determine if the recipe is owned by current user and method **retrieveAllRecipesFromMongoDB()** to get all recipes from MongoDB. There is also method **retrieveAllCategories()** as in SQL part to retrieve all the categories for filtering the result of the report.

```
1   public function calulateTopRecipe_Report1($actualUser, $category){
2               $resultArray = $this->retrieveAllRecipes_Report1($actualUser, $category);
3               usort($resultArray, function($a, $b) {
4                   // Compare the first field of the inner array
5                   $firstComparison = $b[0] <=> $a[0];
6                   // If the first fields are equal, compare the second field
```

```
            if ($firstComparison === 0) {
                return $b[1] <=> $a[1];
            }
            return $firstComparison;
        });
        return $resultArray;
    }

    private function retrieveAllRecipes_Report1($actualUser, $category) {
        $resultObjects = array();
        $allRecipes = $this->retrieveAllRecipesFromMongoDB();
        if($category != null) {
            $allRecipesByCategory = array();
            foreach($allRecipes as $recipe) {
                if($recipe->category == $category) {
                    array_push($allRecipesByCategory, $recipe);
                }
            }
            $allRecipes = $allRecipesByCategory;
        }
        foreach($allRecipes as $recipe){
            $newRating = 0;
            $recipeToResult = array();
            $rating = $recipe->rating;
            if($rating == 0){
                $ratesNumber = 0;
            }
            else {
                $ratesNumber = count($recipe->userRated);
            }
            $recipeName = $recipe->recipeName;

            $newRating = ($rating * $ratesNumber + 5 + 1) / (5 * $ratesNumber + 5 + 5);
            $listRecipeNumber = 0;
            $listRecipeArray = $recipe->listRecipe;
            if (!empty($listRecipeArray)) {
                $listRecipeNumber = count($listRecipeArray);
            }
            array_push($recipeToResult, round($newRating, 2));
            array_push($recipeToResult, $listRecipeNumber);
            array_push($recipeToResult, $this->defineRecipeName($actualUser, $recipe));
            $resultObjects[] = $recipeToResult;
        }
        return $resultObjects;
    }

    private function defineRecipeName($actualUser, $comparedRecipe){
        $definedRecipeName = $comparedRecipe->recipeName;
        foreach($actualUser->recipes as $recipe){
            if($recipe->recipeName == $comparedRecipe->recipeName &&
            $recipe->recipeDescription == $comparedRecipe->recipeDescription &&
            $recipe->category == $comparedRecipe->category &&
```

```
            $recipe->rating == $comparedRecipe->rating &&
            $recipe->creationDate == $comparedRecipe->creationDate &&
            $recipe->editedDate == $comparedRecipe->editedDate ){
                $definedRecipeName .=  " <font color=#f56342><b> (myRecipe) </b></font>";
            }
        }
        return $definedRecipeName;
    }

    public function retrieveAllCategories() {
        $result = array();
        $allRecipes = $this->retrieveAllRecipesFromMongoDB();
        foreach($allRecipes as $recipe) {
            $category = $recipe->category;
            array_push($result, $category);
        }
        return array_unique($result);
    }

    private function retrieveAllRecipesFromMongoDB() {
        $manager = new MongoDB\Driver\Manager("mongodb://user:pwd@recipeWebsite_MongoDB:27017");
        $query = new MongoDB\Driver\Query([]);
        $result = $manager->executeQuery($this->dbnameMongo.'.recipe_collection', $query);
        $recipes = [];
        $recipeDocuments = [];
        foreach ($result as $document) {
            $recipeDocuments[] = $document;
        }
        return $recipeDocuments;
    }
```

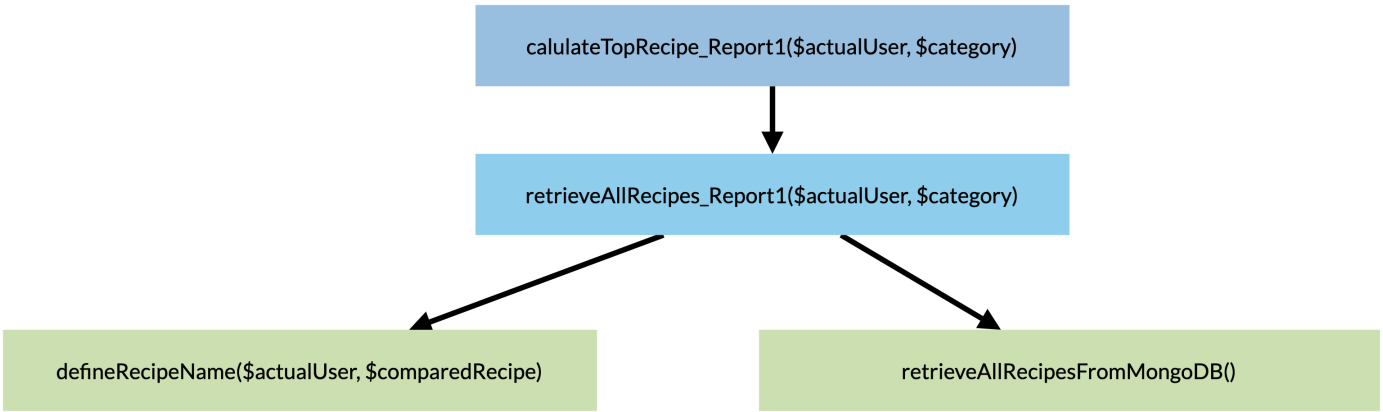


Fig. 10. call graph for report 1 in MongoDB

3.4 Most successful ListRecipe so far (Malte K.)

3.4.1 Theoretical outline

[A] **Description:** The report finds the top-ranked ListRecipes so far or up to a deadline ("most successful" refers to "successNumberN") (6):

$$
\begin{aligned}
\#likesPerListRecipe &:= \text{likesAmount of a ListRecipe} && (1)\\
recipeRating &:= \text{rating of one recipe (1 } - \text{ 5)} && (2)\\
\#comments &:= \text{amount of comments for one recipe} && (3)\\
commentSize &:= \text{length of one comment x} && (4)\\
recipeCommentRating &= (recipeRating + (\#comments * commentSize)) && (5)\\
\\
\boldsymbol{successNumberN} &= (\#likesPerListRecipe * 0.75) + (recipeCommentRating * 0.25) && (6)
\end{aligned}
$$

[B] **Entities:** ListRecipe, Recipe, Comment, userLikesListRecipe, listContainsOfRecipe
[C] **Sorted by:** creationDate, successNumberN
[D] **Filtered by:** creationDate (listRecipes created before are included)
[E] **Columns:** "creationDate", "listID", "listName", "rating Recipe", "postID Recipe", "recipeIDReference Comment", "commentContent"

3.4.2 Procedure for MySQL

This elongated and already shortened part is intended to symbolize the MySQL procedure for calculating all the necessary data for calculating the successNumberN explained above. The **calculateTopListRecipe_Report2()** method calls the **retrieveAllLists_Report2()** method to iterate over each listRecipe and retrieve all the data. This first calls the **retrieveAndProcessRecipesByListID()** method (to calculate each individual recipe). However, this method must in turn call **retrieveRatingForGivenRecipe()** to get all ratings for a recipe, and it must call **retrieveAndProcessAllCommentNumberForARecipe()** to get and calculate the respective comments of each recipe. Finally, the method **retrieveAndProcessAllLikesForOneListRecipe()** has to be called in the main method **retrieveAllLists_Report2()** in order to get all likes for each listRecipe.

```
1  public function calulateTopListRecipe_Report2($myUserID, $selectedDate){
2      //* [A] retrieve all calculations for all listRecipes
3      $resultArray = $this->retrieveAllLists_Report2($myUserID, $selectedDate );
4      //* [B] sort them descending order
5      usort($resultArray, function($a, $b) {
6          return $b[0] - $a[0];
7      });
8
9      return $resultArray;
10 }
11
12 public function retrieveAllLists_Report2($myUserID, $selectedDate ){
13     $resultObjects = array();
14     $dateFrom = $selectedDate;
15     $sql = "SELECT * FROM listRecipe WHERE creationDate <= '$dateFrom'";
16     $dataBaseConnection= new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB", "user", "pwd");
17     $stmt = $dataBaseConnection->prepare($sql);
18     $stmt->execute();
19
20     //* hier AusgangsPunkt, weil man durch alle Listen geht
```

```
    while ($userData = $stmt->fetch(PDO::FETCH_ASSOC)) {
        $successNumberN = 0;
        $listRecipeBundle = array(); //? (listName, successNumberN)
        //* [a] collect and calculate "comments" and "recipes rating"
        $recipeCommentRating = $this->retrieveAndProcessRecipesByListID($userData['listID']);
        //* [b] collect and calculate all "likes" for a listRecipe
        $likesPerListRecipe =  $this->retrieveAndProcessAllLikesForOneListRecipe($userData['listID']);
        ...calculation...
        ...calculation...

        ...
    }
    return $resultObjects;
}

public function retrieveAndProcessRecipesByListID($listID){
    //* [A] setup all variables
    $recipeCommentRating = 0;

    //* collect all recipes which belongs to given list (listID)
    $sql = "SELECT * FROM listContainsOfRecipe WHERE listID = :listID";
    $dataBaseConnection= new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB", "user", "pwd");
    $stmt = $dataBaseConnection->prepare($sql);
    $stmt->bindParam(':listID', $listID);
    $stmt->execute();

    //* [C] process for all recipes
    while ($userData = $stmt->fetch(PDO::FETCH_ASSOC)) {
        ...
        $finalRecipeCommentsNumber = $this->retrieveAndProcessAllCommentNumberForARecipe
        ($userData['recipeID']);
        ...
    }

    //* [D] put it the weithing
    return $recipeCommentRating;
}

public function retrieveRatingForGivenRecipe($recipeID)  {
        $sql = "SELECT * FROM recipe WHERE postID = :recipeID";
        $dataBaseConnection= new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB", "user", "pwd");
        $stmt = $dataBaseConnection->prepare($sql);
        $stmt->bindParam(':recipeID', $recipeID);
        $stmt->execute();

        return ($userData = $stmt->fetch(PDO::FETCH_ASSOC)) ? $userData['rating'] : 0;
}

public function retrieveAndProcessAllCommentNumberForARecipe($postID) {
    //* [A] collect all necessary data
    $finalRecipeCommentsNumber = 0;
    $sql = "SELECT * FROM comment WHERE recipeIDReference = :postID";
```

```
    $dataBaseConnection = new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB", "user", "pwd");
    $stmt = $dataBaseConnection->prepare($sql);
    $stmt->bindParam(':postID', $postID);
    $stmt->execute();

    //* [B] AMOUNT OF COMMENTS in ONE recipe
    ...calculation...
    //* [C] length of each comment multiplied by the AMOUNT OF COMMENTS
    ...calculation...
    return $finalRecipeCommentsNumber;
}

public function retrieveAndProcessAllLikesForOneListRecipe($listID) {
        //* [A] collect all necessary data
        $likesPerListRecipe = 0;
        $sql = "SELECT * FROM userLikesListRecipe WHERE listID = :listID";
        $dataBaseConnection = new PDO("mysql:host=dbSQL;dbname=RecipeWebsiteDB",
        "user", "pwd");
        $stmt = $dataBaseConnection->prepare($sql);
        $stmt->bindParam(':listID', $listID);
        $stmt->execute();


        //* [B] AMOUNT OF COMMENTS in ONE recipe
        ...calculation...

        return $likesPerListRecipe;
}
```

The following call tree illustrates the call depth that occurs when collecting data for this report with MySQL:

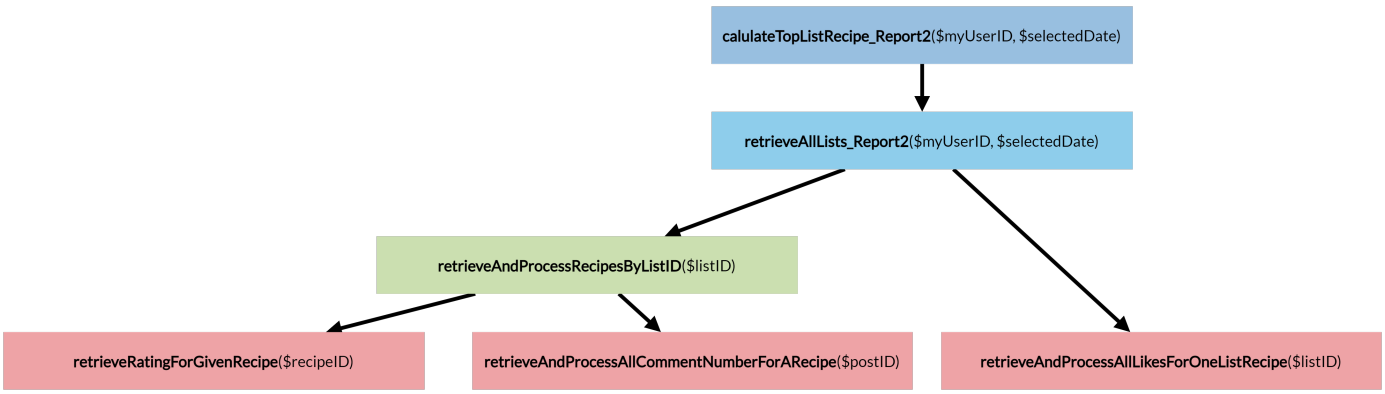


Fig. 11. call graph for report 2 in MySQL

3.4.3 Procedure for MongoDB

The calculation for this report is much leaner with MongoDB, with **calculateSuccessNumberN()** being the controlling method and essentially needing the other two methods to get the respective listRecipe names with **defineListName()** on the one hand and **getAllListRecipesFromMongoDB**() on the other hand to get all lists from MongoDB at all. The **calculateRecipeCommentRating()** method is more object-oriented and could in principle be included in the controlling main method.

```
public function calulateTopListRecipe_Report2($actualUser, $selectedDate){
    //* [A] retrieve all calculations for all listRecipes
    $resultArray = $this->calculateSuccessNumberN($actualUser, $selectedDate);

    //* [B] sort them descending order
    usort($resultArray, function($a, $b) {
        return $b[0] - $a[0];
    });

    return $resultArray;
}


private function calculateSuccessNumberN($actualUser, $selectedDate){
    $resultObjects = array();
    $allListRecipes = $this->getAllListRecipesFromMongoDB();

    foreach($allListRecipes as $listRecipe){
        if($selectedDate >= $listRecipe->creationDate){

            $likesPerListRecipe = $listRecipe->likes;
            //* every recipe in a listRecipe
            $recipeCommentRating = 0;
            foreach($listRecipe->recipes as $recipe){
                $recipeCommentRating += $this->calculateRecipeCommentRating($recipe);
            }
            ...calculation...
        }
    }
    return $resultObjects;
}


private function calculateRecipeCommentRating($recipe){
    ...calculation...
}

private function defineListName($actualUser, $comparedListRecipe){
    ...calculation...
}

private function getAllListRecipesFromMongoDB(){
    ...
}
```

The leaner and simpler procedure for collecting all the data for the report is also illustrated here again by the call graph of the methods one below the other. There is really only one main method from which calls are branched off, basically only two, considering that *calculeRecipeCommentRating()* could also be included in *calculateSuccessNumberN()* method.

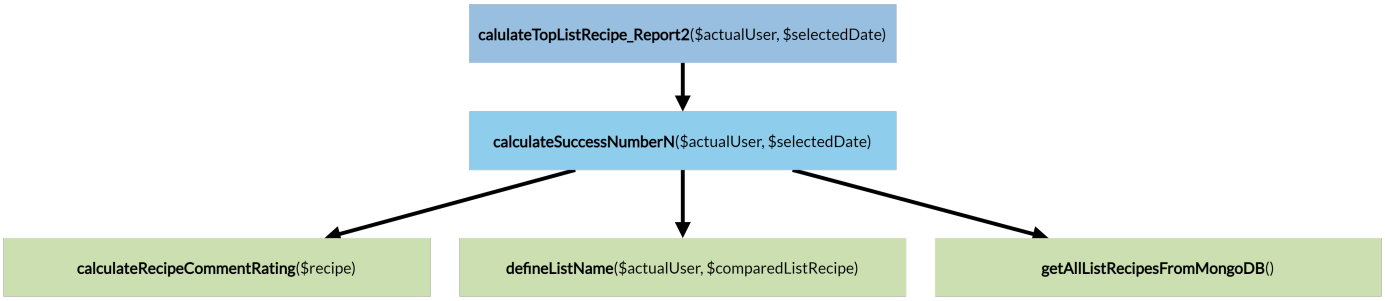


Fig. 12. call graph for report 2 in MongoDB

3.5 Comparison between MySQL and MongoDB for processing data

In order to be able to compare the database management system better with one another in the following, the addition of data on the one hand and the querying of data on the other should be differentiated from one another.

(1) **Adding data:** As for adding data, both database management systems are almost the same and have no real advantage. The only advantage in terms of security are the prepared statements for SQL insertions, which could be rebuilt for MongoDB with further service logic.

(2) **Queries:** However, there are clear differences in the queries. Not only that with MongoDB you have all the data directly together without first using further joins, the code structure is also automatically greatly simplified. This is particularly evident in the call graphs of Report 2.

4 Summary and conclusion

Overall, looking back on this project, it is difficult to say which of the two database management systems is better or worse. MySQL has "old-fashioned" procedures that one is used to, such as thinking in tables, consciously using IDs or outsourcing relations to tables. In this respect, the NoSQL approach is newer, but also intuitive in its way (JSON or BSON structured data without a certain defined schema), in that everything that is required is available directly in one document.
However, both database management systems also have negative sides that often require specially adapted solutions. In MySQL it is sometimes necessary to drop normalization and denormalize for the sake of simplicity. Sometimes you also have to put up with more complicated mechanisms in order to be able to collect data together. However, there are also dangers with MongoDB, such as somewhat recursive data structures, for which one often has to decide on suboptimal solutions. For this part for example, we are not quite sure if we designed the user-MongoDB-structure the right way, when we not allow that "follow"-user can also have "follow"-user on their own. This decision was made to deliberately prevent recursion, but may need to be corrected in the future if you want to make user-centric data usable on this website - so it's not a final decision, but more of a pragmatic trade-off.
Overall, the direct comparison and migration from MySQL to MongoDB shows that the document-based database management system type is much easier to handle, MongoDB also seems to be better for scalable applications with large amounts of data, since flexible data modeling can be used. This flexibility has also simplified the direction of migration from MySQL to MongoDB, in our view.