Master 1 Data sciences

Personal project [PROJ0011]:

Freezer manager

Loïc Lejoly s141123

Contents

1 Introduction

As part of the personal project course given at ULiège, I decided to develop a system to manage domestic freezers. This idea emerged to me because of a personal experience. Every two years, my family and I reorganize the home's freezers. I was a bit shocked when I saw the quantity of products that was in the freezers and for which we have completely forgotten that they were there. This is a problem that anyone who has freezers can potentially face someday.

The main problem of forgotten products is they cannot stay indefinitely in a freezer because of their physical properties. For example, meat kept in a freezer for a long time does not taste good. As results, some products are thrown to the rubbish. We could avoid throwing to the rubbish products if we were aware of the existence of them. The solution of that problem would be an application or a tool that gives the possibility to a user to manage the content of his freezers and more. At the moment, it seems that no free applications, except for smartphones, exist. Since it does not exist cross-platform applications, I decide to develop a cross-platform system that implements a freezer manager as subject of my personal project.

1.1 Project direction

The aim of the project is to provide a cross-platform freezer manager that gives to a user using this system some information about his products. This system also keeps a history of user's products. By cross-platform, I mean a system that does not depend on the operating system and that can be used on any system. Moreover, the main purpose of that manager is to give the possibility for third parts to develop their own applications and to use the freezer manager developed inside of them. It is personal choice and I took this option because I appreciate open-source applications and I would like to bring my personal contribution to the open-source-world.

1.2 Problems to face

To build a freezer manager. It is mandatory to identify the different aspects of the problem to solve. This first problem is to identify the different information to store in the freezer manager. These information must be stored in a database with some other data. The architecture of this database and data stored are explained in details in the section made for that purpose.

Since, the aim of the system manager is to give the possibility for a user using the system to find what he wants, where a product is located,... The system manager cannot be used directly because we want to give access to the freezer system manager for third applications. But we do not want that these third applications have the possibility to modify or destroy the system manager architecture. Moreover, to demonstrate the functionalities of the system, it is recommended to demonstrate the usage of the system on an external application.

1.3 Choices made to solve these problems

1.3.1 Useful information to collect

The information that the system must take into account are:

1. The user that uses the system manager.

2. A unique token linked to this user. This token is necessary to interact with the system.

3. A password and an identifier (email address) linked to that user. These two information are essential to obtain the token. The way to obtain a token will be explained later in the next sections.

4. The language that the user uses. This information is not used in the scope of the project but it is a useful information to store if we want to extend the project later and potentially acquire some knowledge about the user.

5. The freezers linked to this user. This information is fundamental because we need to locate a product.

6. The products linked to a user. A product is defined by several features:

   (a) A product name.
   (b) A description of the product given by the user.
   (c) A quantity in terms of people (e.g: a meal for 2 people).

(d) A product type (e.g; vegetable, soup,...). These types are predefined in the system and only the administrator of the system has the possibility to extend and/or reduces the product types list.

(e) A date of input and a date of output.

(f) A period during which a product may remain inside a freezer.

(g) An identifier that links the product to one of the user's freezers.

(h) A box number that allows to locate the box inside the freezer where the product is stored.

(i) An identifier inside the box where the product is stored to allow to retrieve the product asked. This identifier is unique inside the box. An example of a complete identification of a user product: the *product number 2* in *box number 3* of the *freezer number 1* for the user that has the token *x*.

1.3.2 Technical aspects

Since we want to give access to third applications, the system cannot be used directly. To work around this problem, interaction with the system manager is done through a REST API. REST API is commonly used by companies who want to give access to their data or a piece of their data to third applications without giving direct access to their databases. The main advantages of a REST API are multiple. First of all, it is a client-server that means the server evolves independently of the client and it is stateless. Moreover, a REST API allows to develop an application on any terminal. It is possible to develop an application for raspberry Pi, for a smartphone or any other connected object. Since a REST API is stateless, and we want to identify a user, we need a method to go around this problem. It is the aim of the token. The token allows detecting uniquely a user if it is correctly generated. A schema given Figure 1 shows how the REST API is set.
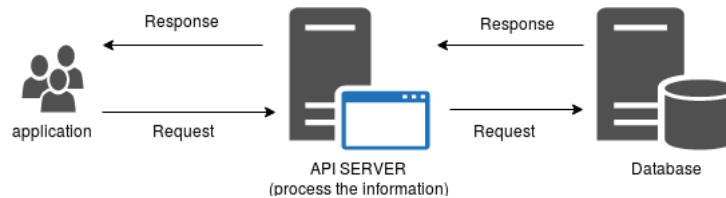

Figure 1: REST API interaction

As you can see, an application that uses the API has no direct access to the database. With this methodology it is also easier to apply some internal modifications on how data is processed. Moreover, it allows to abstract the way data is stored. It can be a good starting point for web semantics since the API is based on URI.

With a user token and the REST API, a user has the possibility to find what he wants. He can add new products, he can edit existing products, he can see what are the products he has in a specific freezer. He also has the possibility to see the history of a freezer. Moreover, the API can make suggestions for the user according to his previous consumptions and depending on what his freezers contain.

To give the possibility to third people to understand how the API works. I made a documentation for that purpose with examples that show how it works and what are the results obtained.

Finally, to demonstrate the utilization of a third application with the REST API. I built a website that only uses the API to manage the freezers of a specific user.

1.4 Overview

This report is divided into several sections. The first section discusses the way data is stored and choices made concerning the architecture. The second section discusses the creation of an internal website that allows to manage a user account. The aim of this website is not to manage user's freezers but more for managing account information. In this section an admin panel is also discussed to explain how to manage the freezer manager system. The *REST API* section explains how the REST API has been built, the different features available,... The *external website* section explains the external website built in order to use the REST API built. The *project challenges* section discusses the problems faced during the project and the improvements/future work that can be achieved on it. The last section discusses what the project brings me and the culmination of it.

The code related to this project follows this directory architecture:

```
freezer_manager
├── database
├── documentation
├── external_website
├── website
├── rest_api
├── flask_run.sh
├── run_server.sh
├── README.md
```

database directory contains a sample of the database created with the schema linked to it. *documentation* directory contains files related to the API documentation and the website built for that purpose. *external_website* directory contains files used to build the external website. *rest_api* directory contains files used to build the REST API server and communicate with the database. *flask_run.sh* and *run_server.sh* are scripts used to run severs(note that the settings used in these scripts are dependent of the system installation. If you want to use these scripts please refer to the *README.md* file. *README.md* explains the folder architecture in more details and how to install third programs. Note that the source code used for this project is available on *GitLab* at the following address: https://gitlab.com/lejoly/personal_project.

2 Content storing

2.1 Database model discussion

To store the different information about a user and the data related to his freezers a database is required. There exists several types of databases and it is not easy to make a choice that will not have bad impacts on the future if the choice made at the beginning was not good. The choice made is a relational database. This choice is motivated by several things. First of all, the problem is known and links to establish between data are known. Moreover, data would not change on the long-term that means the architecture of the database would not change a lot and the quantity of information to store will never be huge. In addition, data to store is essentially texts and numbers, thus no need to have a database that can manage video streams like *noSQL* databases. Finally, the relational database architecture is probably the architecture which is the easiest one to be ported into another database architectures like *Cassandra* or *MongoDB* or anything else architecture.

The relational database management system chosen is *MySQL* since it is an open-source software which is commonly use. Moreover, it does not require too much work to set it up.

2.2 Relational database structure

The information that the database needs to store will be explained in detailed.

First of all, we need to store some information about a user such as an email which is unique and a password that allows the user to establish a connection with the internal website. This website is an integral part of the freezer system manager and gives additional information about the user such as the token linked to the user account. This token is mandatory to have access to the freezer manager API and it is generated uniquely. It is also used to identify a user.

A user has also two additional information that are the language and learning fields. The language field is used to specify the language used by the user as explained earlier. It can be interesting for third applications to have some knowledge about the user. The learning field is used to save some parameters that could be useful for learning algorithm. For this project I did not use it because I did not have enough time to find and test some of these algorithms. So a default value *[0,0]* is simply put. To summarize a user is defined through 5 fields which are a token, a password, an email, a language, and a learning.

After defined a user, we need to define the other objects that will be stored and linked to a user.

The database contains several types of products. These product types can only be added by the administrator of the system to avoid inconsistencies that users can bring by adding weird product types or by defining several times the same type. We need to keep in mind that some users just want to break the system instead of using it. A product type is described by a unique identification number automatically generated by the database, the name of the product type in English and in French. The identification number allows to locate and retrieve the information of a specific product type. If we want to extend the number of languages we can add new columns in the database without affecting the architecture or using a translation tool to translate the

existing languages into the desired language. Since translation tools are very powerful, the second proposition seems to be more adequate. To summarize a product type is defined through 3 fields which are a type identification, an English type name, and a French type name.

To store user's products we need to formally define how a freezer is described. A freezer is described with three fields which are a unique identification number, the number of boxes that the freezer has, and the name given to this freezer. The identification number is used to locate and retrieve a specific freezer. It is automatically generated by the database. To summarize a freezer is defined through 3 fields which are the freezer identification, the number of boxes that the freezer contains, and a freezer name.

A freezer product from a user contains several fields. First of all, a unique identifier given by the system itself. A date of input, this date must be anterior to the actual day. In other words it is impossible to encode product with a date that refers to a place in the future. A date of output, by default this field is set to null. When this field is not null it means that the product is not any more in the freezer. The date of output must be greater or equal to the date of input. A product also contains a period field which is used to define the period in months for which the product should be in the freezer. Thus, third applications can display products that overcome this period. The quantity is also a piece of information of a product. It gives the quantity in terms of person for whom the product can be used. Finally, a product also has two additional fields to allow a user to locate more easily a product. These two additional fields are the box number which give the box for a specific freezer and the product identifier which allows to locate the product in this box. This identifier is unique for the specific box.

A product also receives a description. The product description contains three different fields which are the description identifier, the product name and a free description of the product. This description can be a structured description or a simple text.

Now that we know what the database will contain and the links that exist between the different elements, we can establish the entity-relationship model that will make it easier to visualize our database. This is explained in details in the following section.

2.3   Entity-relationship model

The Figure 2 shows the entity-relationship model of the freezer database. As we can see on this schema the relations *list_freezer* and *product_to_type* are not mandatory since the information that they linked can be retrieved from the rest of the database. These two relations have been inserted into the database for a simplicity since these information are often asked. Knowing that it is interesting for the backup part.
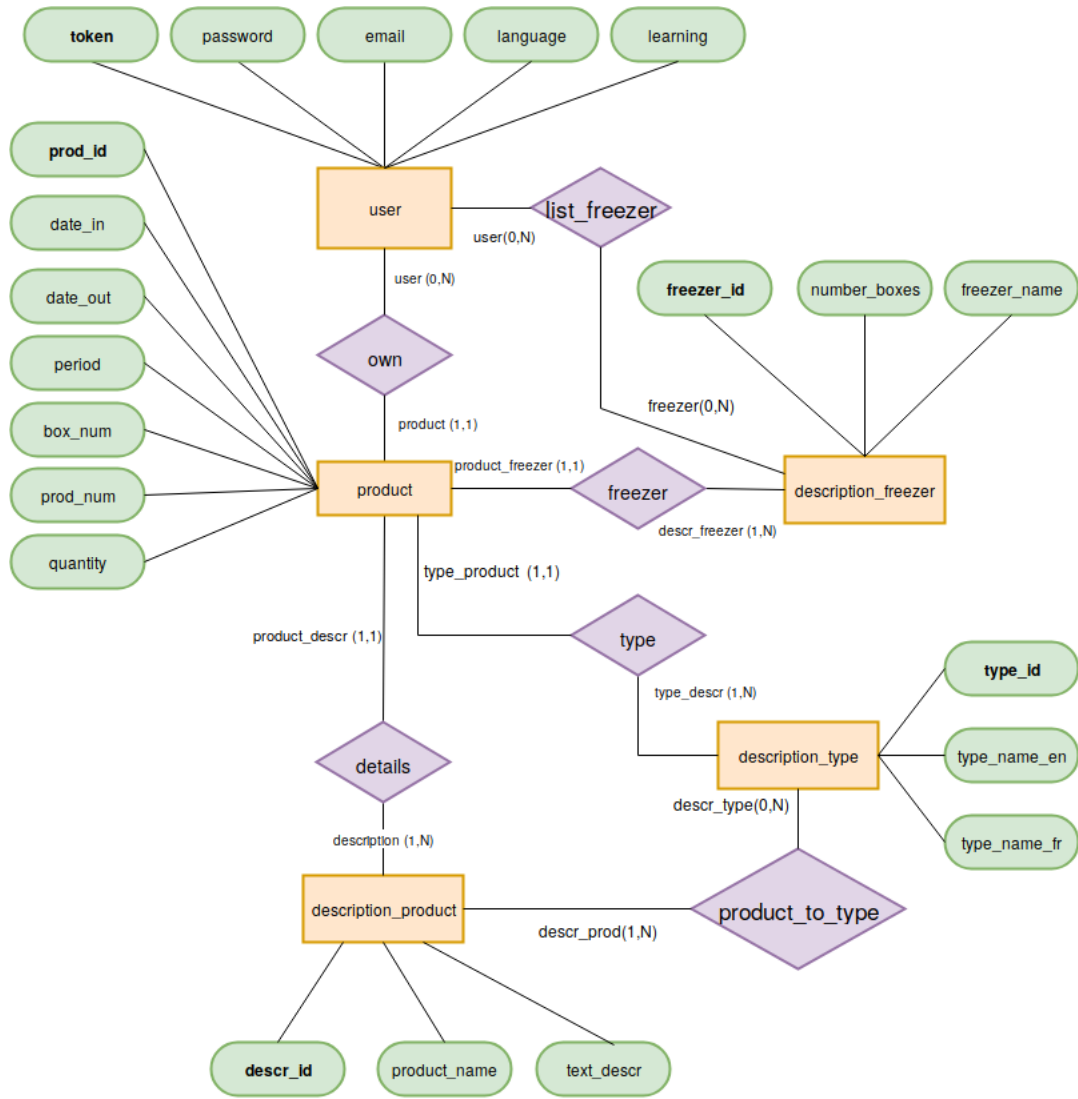
Figure 2: entity-relationship model of the database

2.4 Relational model

From the entity-relationship we can go into the relational format which allow to have a representation for integrating the database into a relational database management system.

1. User(**token**, password, email, language, learning)

2. Description_type(**type_id**, type_name_en , type_name_fr)

3. Description_product(**descr_id**, product_name, text_descr)

4. Description_freezer(**freezer_id**, number_boxes, freezer_name)

5. Product_to_type(**#descr_id, #type_id**)

6. List_freezers(**#freezer_id, #token**)

7. Product(**prod_id**, #token, #descr_id, #freezer_id, #type_id, date_in, date_out, period, box_num, prod_num, quantity)

The database uses *MySQL*. The engine used is *InnoDB* with utf8mb4_unicode_ci as collation. The choice of this engine has been done because is the default engine and manage ACID transactions and foreign keys. The Figure 3 depicts the *SQL* tables implemented and the types associated to each variable. The choices made for the length size of variables are personal choices. But the length size can be changed with no difficulties if the need arose. The encoding chosen allows to be flexible as much as possible.

6

Figure 3: slq tables of the database

As discussed above the architecture of the database allows to skip some tables to backup like *Product_to_type and List_freezers*. These two tables can be reconstructed with the others tables. Thus, it is not mandatory to backups these ones.

To use the database we need to fill it with some data. This filling part has been done manually. The reason of that is to represent something that has a sense and can represent a lambda user. Moreover, I made this choice because initially I wanted to develop some preference learning algorithm and in this optic fill a database with random objects has no sense.

3   Internal website

After the set up of the database, we need to give the possibility for new users to create an account to have access to the freezer API and manage their freezers. Without an account a user cannot do anything. It is the reason why an application to create and manage a user account is needed.

The type of application chosen to manage these things is a website. This choice has been made because we can have a tool that is running on a browser and it avoids the potential conflicts from an OS to another one. The languages used for the website development are *HTML/CSS* and *JavaScript* for the front-end and *PHP* for the back-end. *PHP* is a server-side language which is commonly used when we develop a website and it allows interacting with the *MySQL* database.

The internal website is divided into three distinct parts. The first one concerns the account creation, the second concerns a user panel that can be accessed when an account is created, and the last section is dedicated to the admin panel. This panel is only dedicated to the administrator of the freezer application.

The recommended protections for a website are not configured or use the default configuration files. To secure a website some files such as *.htaccess* or *php.ini* need to be correctly set in accordance with the server. The transactions made between a user and the server are done by using *HTTP* protocol but to be secure *HTTPS* should be used. Moreover, the redirection pages are not configured and/or personalized but for a production website it is required.

3.1 Account creation

To use the API a user need to create an account with an e-mail address and a password. A webpage (Figure 4) is built to achieve this goal(*website/register.php*). To have a the password encrypted, a *PHP* function called *crypt()* is used with a salt. By default, *crypt* function uses the DES algorithm. This algorithm was the most widely used algorithm during many years. Nowadays this algorithm is not secure to crypt data but in the case of the project is the algorithm chosen.

If the objective is to put in production the project. It is recommended to change the password encryption to use another type of encryption like SHA 512 or another secure encryption algorithm.

When the account has been created it is linked to a token. This token is generated on the server side with the PHP function *random_bytes* which is recommended to generate string of cryptographic random bytes that are suitable for cryptographic use. The length specified for the generation is 32 bytes. Thus, the number of possibilities is $(2^8)^{32} = 2^{256}$. After that, the random sequence generated is transformed into a hexadecimal string to be more readable by a novice user.
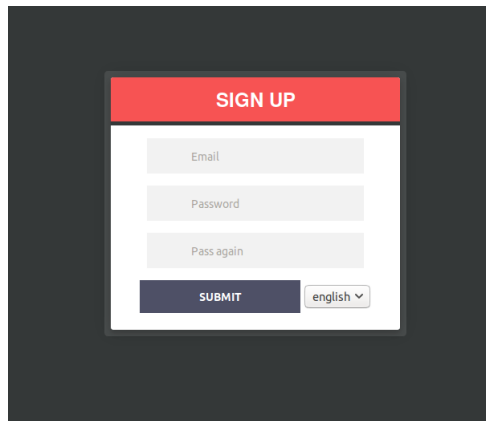

Figure 4: register page

Note that the token generated is static but in general a token has an expiration date. I did not go into that direction because it becomes complex to manage such token.

When the account is created a user can use the API to build its own software and manage his freezers with the help of The API and the token linked to its account or use an existing application that uses the API. Third applications should only ask for your token and nothing else.

3.2 User panel

When a user possesses an account. He has the possibility to access to a user panel of the freezer system manager with the *website/login.php* page Figure 5. If the information given are correct, he will be redirected to the user panel page *website/user/user_interface.php* Figure 6.
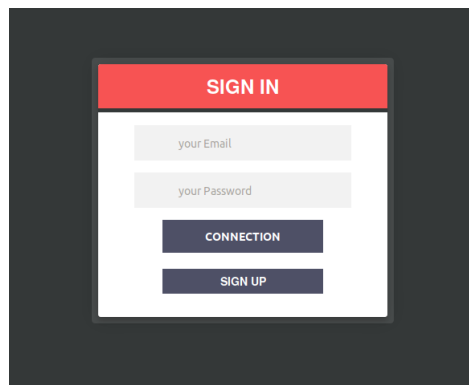

Figure 5: Login page

This user panel is used to give to the user more information about its account. i.e: the email address, the token linked to the account, the possibility to change the password and the language. For the scope of the project the user panel is not developed more than that but for production project it should be the case. We could add some features to manage our freezers, etc.

Figure 6: User panel

3.3 Admin panel

It is interesting for a system administrator to have access to an admin panel. In general, an admin panel is used to manage a system and more specifically the database linked to it. It allows to have a global view of the system. Analyse information about a specific user. Manage the system by applying some updates. Compute some statistics and more. In the scope of the project, I built a really simple panel that only allows to list and add a new product type. I would have liked to develop it in details but I did not have enough time to do that and it potentially is too much work for the scope this personal project.

The Figure 7 shows the admin login page. As we can see this page is really simple. To have the possibility to connect to the admin panel you need to give a identifier (admin) and a password (root). These two informations are hard-coded in the *website/admin/login.php*. For a practical case this approach is not recommanded. Instead of encoding these informations inside a php file it is recommanded to store it in the *.htaccess* file or inside a configuration file like *config.ini*. This allows to separate the code from the configuration and sensitive information.

Figure 7: admin login page

When the identifier and the password given are correct. You will be redirected to the admin panel Figure 9. As we discussed above, the admin panel is very simple. You have the possibility to display and hide the different product types, you can also add a new product type.
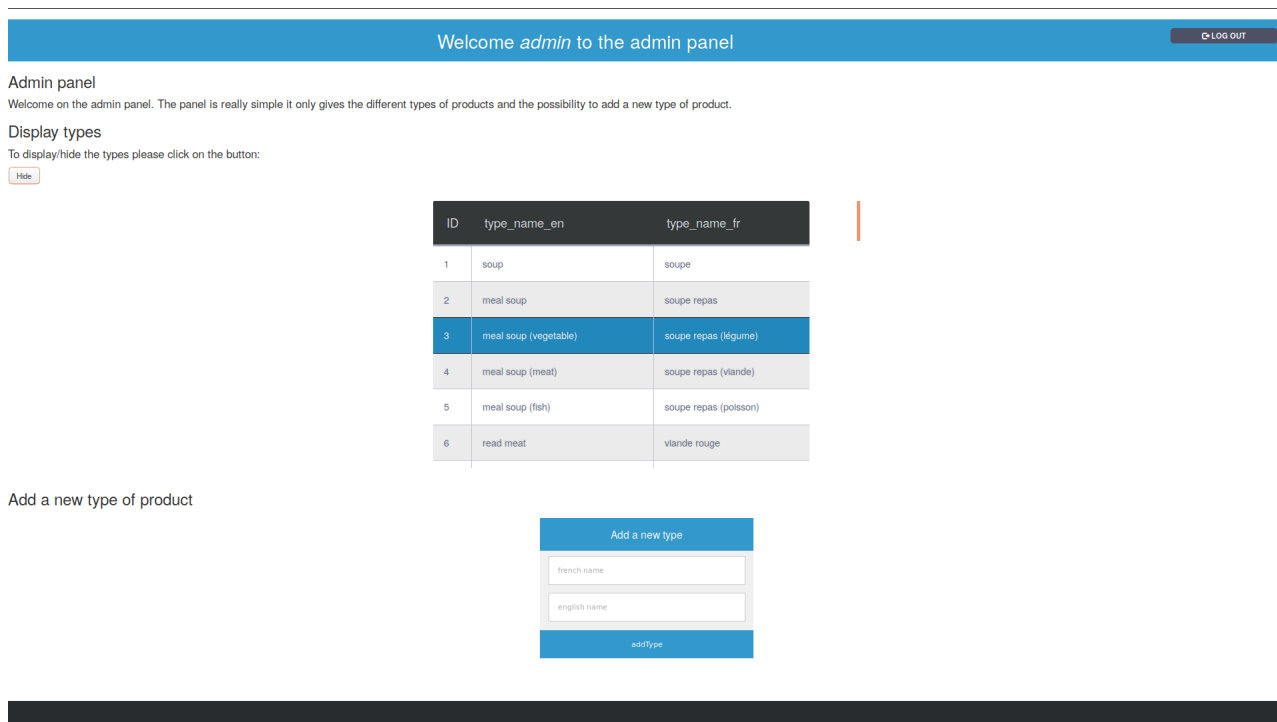
Figure 8: admin login page

4 REST API

A REST API is built on a server that establishes a communication between the database and the client. REST API is useful for several reasons. The first reason is that it is a client-server that means the server evolves independently of the client. Moreover, with a REST API you can develop applications without worrying about changes in queries since they are the same on any terminal. Finally, new functionalities can be added easily since the REST API is based on URI. We Just need to specify a new URI and specify what the request needs to achieve.

The REST API server implemented in the project scope cannot be really called a RESTful server because all specifications asked to be certified as REST are not fulfilled. A layered system and cacheability are not implemented because it was too complex to set up and I would have spent a lot of time to set up these parts.

Since the REST API needs to work with bowers that specify inside their *HTTP* request headers the field *Origin* for some type of requests like *POST*, the REST API needs to use cross-origin resource sharing (CORS). The CORS allows the REST API server to accept AJAX requests which are forbidden due to the same-origin security policy.

A limiter system has been set to limit the number of requests that a user can do by minute (100 requests) and by day (24000 requests). The user IP address is used to detect a user. The system that achieves this task is an extension of the flask micro-framework and it is called *flask-limiter*.

Also note that flask is single threaded with version 0.12.2. With the stable version 1.0 that comes up April 26th 2018, it is possible to use several threads in development mode. To put the flask server in production and gives the possibility to use several threads it is recommended to use *Gunicorn* [1] or *uWSGI*[2].

Finally, concerning the protocol used to achieve the different transactions it is the HTTP protocol. It is recommended to use a secure protocol like HTTPS if we want to use the API in production. But for the scope of the project I did not set up an HTTPS protocol since you need to set up a server like *gunicorn* to have a trust SSL certificate.

---

[1]http://gunicorn.org/
[2]https://uwsgi-docs.readthedocs.io/en/latest/

4.1 Folder architecture

The REST API follows the folder architecture given below. The folder *tests* is used to implement the different unit tests done on the server. The others files defined at the root of the REST_api folders are used to build the server itself.

```
REST_api
├── tests
│   └── tests.py
├── mysqlRequests.py
├── queryDB.py
├── responseMessage.py
├── sever.py
├── utils.py
└── validatorDB.py
```

1. *mysqlResquests.py*: This module defines the major part of requests done by the API. It only contains variables that are PURE sql strings. It is simpler to maintain SQL requests with a separate file where sql requests are defined instead of looking in the core of the server.

2. *queryDB.py*: This module implements a class *QueryDB* that is used to send SQL requests and retrieve data of them.

3. *responseMessage.py*: This module defines the detailed response messages that are send with an error status code. It only contains English strings. With this method it is possible to change the language of the messages sent simply by overriding these variables with the appropriate text.

4. *server.py*: This module is the server core it implements the server. It is the module to run to set the server up.

5. *utils.py*: This module implements some functions used by other modules.

6. *validatorDB.py*: This module implements a class *ValidatorDB*. This class is used to validate parameters that will be sent with MySQL requests. It avoids to bring inconsistencies in the database due to incorrectly done user requests.

4.2 Server

As you can see with the folder description above, the server is developed in python. I made this choice because I wanted to discover the possibilities offered by this language but others languages are also perfectly fit to achieve that such as *Java, PHP, JavaScript,...*

A micro-framework is used to develop the Python server with Python language. This framework is Flask [3] and it is build an *Werkzeug* and *Jinja 2*. The version of Flask used to develop this project is 0.12.2. The stable version of the micro-framework just comes up (April 26th 2018) with a number of changes.

To interact with the *MySQL* database I used the module *MysqlDB* [4]. This module is not the best known nor the most used. Nevertheless, this module allows to write SQL requests in a pure SQL fashion. I chose this module instead of others well-known modules based on a Object Relational Mapper (ORM) such as *sqlachemy* for two reasons. The first one is that an ORM like *sqlalchemy* makes abstractions of the SQL part. It can be an advantage in terms of optimizations, to manage cursors,... The drawback of ORM is that you cannot necessarily understand what is behind this module. The second reason is that the set up of ORM is no so easy to do correctly. It seems understandable since the ORM makes abstraction of sql part. Thus, I made the choice to take the *MysqlDB* instead of *sqlachemy* module. Nevertheless, for a production application, it is recommended to use an ORM because these tools are optimized to achieved requests, They give the possibility to change database type without changing all requests in the code. Moreover, the application can be maintained by people not necessarily mastering the SQL language.

4.3 Functionalities implemented

Several functionalities have been implemented. For each request done by a user, parameters sent with the requests are parsed to avoid SQL injection. Moreover, type parameters are also checked. A first check is automatically done by the flask micro-framework by checking the format of the URI.

---

[3]http://flask.pocoo.org/
[4]https://github.com/PyMySQL/mysqlclient-python

For instance the URI format specified in the server follows:

$$@app.route(”/myRequest/ < int : identifier > ”)$$

If a user gives a string *myIdentifier* as parameter flask automatically detects that the URI format is not correct and simply returns a status code 404 because the route with this URI format does not exist.

If all parameters have a correct type the server will check the consistencies of the data sent with the request. If data is consistent the sever can interact with the database otherwise an error status code is returned. An object with a detailed explanation is also returned with an error status code. This JSON object has two fields *details and status*. The *status* field contains the error status code sent and *details* field contains a text that explains the reason of that status code.

4.3.1 Check_token

This request implements a single method which is *GET* method. It gives the possibility to check the validity of a given token. If this one is correct a response with the status code 200 is sent. Otherwise, an error status code is sent with a JSON object as explained before.

The prototype of this request is the following:

$$< domainUrl > /check\_token/ < token >$$

Parameters:

1. <domailUrl>: The url of the server API. In the case of the project the server is run on the *localhost:5000*

2. <token>: A user's token that corresponds to an existing account.

4.3.2 Types

This request implements a single method which is *GET* method. It gives the possible types of products that can be encoded into the database. To obtain these types you need to give an existing token. If this token is correct a JSON object containing the different types of products is sent and the status code of the response is 200. Otherwise, an error status code is sent with a JSON object as explained before.

The prototype of this request is the following:

$$< domainUrl > /types/ < token >$$

Parameters:

1. <domailUrl>: The url of the server API. In the case of the project the server is run on the *localhost:5000*

2. <token>: A user's token that corresponds to an existing account.

The architecture of the JSON file returned is the following:

```
[
 {
  "type_id": 1,
  "type_name_en": "soup",
  "type_name_fr": "soupe"
 },
 {
  "type_id": 2,
  "type_name_en": "meal soup",
  "type_name_fr": "soupe repas"
 },
 ...
]
```

Explanation of the JSON object:

1. *type_id*: An 'integer' that represents the id of the product's type

2. *type_name_en*: The type name in English

3. *type_name_fr*: The type name in French

Note that it is not possible for a user to add a product type. It is the desired behaviour since we do not want a user with the possibility to add random product types or define several times the same product type. This functionality only given to the admin of the system. This functionality is implemented for the admin website.

4.3.3 Freezers

This request implements a four methods which are *GET,POST*, *PUT*, and *DELETE*. These methods are used to manage the user's freezers and they share a common prototype:

$$< domainUrl > /freezers/ < token >$$

Parameters:

1. <domailUrl>: The url of the server API. In the case of the project the server is run on the *localhost:5000*

2. <token>: A user's token that corresponds to an existing account.

GET method: This method is used to retrieve information about the different freezers that a user possesses. This information is a list that contains all user's freezers and each freezer is described by three elements which are the identifier, the name, and the number of boxes that the freezer contains. If this token is correct a JSON object containing the information about the user's freezers is sent and the status code of the response is 200. Otherwise, an error status code is sent with a JSON object as explained before.

The architecture of the JSON format is:

```
[
 {
  "freezer_id": 1,
  "freezer_name": "freezer 1",
  "number_boxes": 4
  },
  {
   "freezer_id": 3,
   "freezer_name": "freezer cave",
   "number_boxes": 6
  },
  ...
]
```

POST method: This method is used to add a new freezer for a specific user. To add a new freezer data sent must follow a certain data type. The content-type sent must be a *application/json*. If another content-type is sent, the request does not work and the status code 415 is returned in addition with two fields *details and status* described earlier. If the content-type is correct the JSON sent must respect the architecture given below. If the user token, the content-type, and the JSON object sent are correct the response status code 200 is received. Otherwise, an error status code is sent with a JSON object as explained before.

The architecture of the JSON object to send must be:

```
{
 "num_boxes": "4",
 "name": "my_name"
}
```

Explanation of the JSON object:

1. *num_boxes*: An 'integer' that represent the number of boxes.

2. *name*: An 'string' that represents the new name given to the freezer.

PUT method: This method is used to update an existing freezer which belongs to a specific user. To update a specific freezer data sent must follow a certain data type. The content-type sent must be a *application/json*. If another content-type is sent, the request does not work and the status code 415 is returned in addition with two fields *details and status* described earlier. If the content-type is correct the JSON sent must respect the architecture given below. If the user token, the content-type, and the JSON object sent are correct the response status code 200 is received. Otherwise, an error status code is sent with a JSON object as explained before.

The architecture of the JSON object to send must be:

```
{
 "freezer_id": the id of an existing freezer,
 "num_boxes": "4",
 "name": "my_name"
}
```

Explanation of the JSON object:

1. *freezer_id*: An 'integer' that represents the id of an existing freezer that belongs to the user with the token given.

2. *num_boxes*: An 'integer' that represents the number of boxes. Can be leaved empty if we do not want to update the number of boxes.

3. *name*: A 'string' that represents the new name given to the freezer. Can be leaved empty if we do not want to update the freezer name.

DELETE method: This method is used to delete an existing freezer which belongs to a specific user. To remove a specific freezer data sent must follows a certain data type. The content-type sent must be a *application/json*. If another content-type is sent, the request does not work and the status code 415 is returned in addition with two fields *details and status* described earlier. If the content-type is correct the JSON sent must respect the architecture given below. If the user token, the content-type, and the JSON object sent are correct, the response status code 200 is received. Otherwise, an error status is sent with a JSON object as explained before.

The architecture of the JSON object to send must be:

```
{
 "freezer_id": the id of an exisitng freezer
}
```

Explanation of the JSON object:

1. *freezer_id*: An 'integer' that represents the id of an existing freezer to delete.

4.3.4 Freezer_next_id

This request implements a single method which is *GET* method. It gives the possibility to obtain the next available identifiers of all boxes of a specific freezer. These available identifiers represent the number that will be attributed for a new product of the concerned freezer box. If the request is correctly done a JSON object containing the next available box identifiers for a specific freezer and the status code of the response is 200. Otherwise, an error status code is sent with a JSON object as explained before.

The prototype of this request is the following:

$$< domainUrl > /freezer\_next\_id/ < freezer\_id > / < token >$$

Parameters:

1. <domailUrl>: The url of the server API. In the case of the project the server is run on the *localhost:5000*

2. <freezer_id>: An integer that represents the id of an existing freezer that belongs to the user with the token given.

3. <token>: A user's token that corresponds to an existing account.

The architecture of the JSON file returned is the following:

```
{
 "1": 8,
 "2": 3,
 "3": 4,
 "4": 3
}
```

Where for instance, "1" represents the box number and 8 the next available id for the product in the box 1.

4.3.5 Get_product

This request implements a single method which is *GET*. It give the possibility to retrieve products of a specific user by specifying some parameters to the request. As for other get requests If the request is correctly done a JSON object containing information and the status code of the response is 200 are returned. Otherwise, an error status code is sent with a JSON object as explained before.

The prototype of this request is the following:

$$< domainUrl > /get\_product/ < param > / < freezer\_id > / < token >$$

Parameters:

1. <domailUrl>: The url of the server API. In the case of the project the server is run on the *localhost:5000*

2. <params>: A string that define the parameter that we want three parameters are possible:

    (a) *all*: Selects all products.
    (b) *inside*: Selects only products that are inside freezers.
    (c) *outside*: Selects only products that are outside freezers.

3. <freezer_id>: An integer that represents the id of an existing freezer that belongs to the user with the token given. If we want to select all freezers that belongs to a user the id 0 must be specified.

4. <token>: A user's token that corresponds to an existing account.

```
[
 {
  "box_num": 1,
  "date_formatted_in": "2016-12-31",
  "date_formatted_out": "2017-03-02",
  "descr_id": 2,
  "freezer_id": 1,
  "freezer_name": "frigo 1",
  "period": 6,
  "prod_num": 2,
  "product_name": "glace vanille",
  "quantity": 1,
  "text_descr": "glace maison au aromatis\u00e9 \u00e0 la vanille",
  "type_id": 24
 },
 {
  "box_num": 1,
  "date_formatted_in": "2017-12-26",
  "date_formatted_out": null,
  "descr_id": 4,
  "freezer_id": 1,
  "freezer_name": "frigo 1",
  "period": 6,
  "prod_num": 4,
  "product_name": "Soupe de No\u00ebl",
  "quantity": 4,
  "text_descr": "Soupe \u00e0 base de tomate, brocolli et poisson",
```

```
  "type_id": 1
 },
 ...
]
```

The JSON object returned is a list of element with the architecture shown above. Explanation of the different elements

1. *box_num*: Represents the freezer box where the product is stored.

2. *date_formatted_in*: Represents the input date of the product in the freezer. The format of the date is YYYY-MM-DD.

3. *date_formatted_out*: Represents the output date of the product in the freezer. The format of the date is YYYY-MM-DD. If there is no output the value is simply null.

4. *descr_id*: The identifier of the product description.

5. *freezer_id*: The identifier of the freezer.

6. *freezer_name*; The freezer name.

7. *period*: The period in months for which the product can stay in the freezer.

8. *prod_num*: The identifier of the product inside the freezer box.

9. *product_name*: The name of the product.

10. *text_descr*: The description of the product.

11. *type_id*: The type identifier of the product.

4.3.6 Add_product

This request implements a single method which is *POST*. It gives the possibility to add a new product into a existing freezer for a specific user. A JSON object must be sent with a specific architecture. This method follows the same behaviours as the preceding *POST* methods previously described.

The prototype of the request is:

$$< domainUrl > /add\_product/ < token >$$

Parameters:

1. <domailUrl>: The url of the server API. In the case of the project the server is run on the *localhost:5000*

2. <token>: A user's token that corresponds to an existing account.

The architecture of the JSON object to send must be:

```
{
 "product_name": "name",
 "text_descr": "description",
 "freezer_id": number,
 "type_id": number,
 "date_in" datetime YYY-MM-DD,
 "period":" number,
 "box_num": number,
 "prod_num": number,
 "quantity": number
}
```

Explanation of the JSON object:

1. *product_name*: A string that represents the name given to the new product.

2. *text_descr*: A string that represents the description of the product.

3. *freezer_id*: An 'integer' that represents the id of an existing freezer that belongs to the user with the token given.

4. *type_id*: An integer that represents the identifier of an existing product type. TO know the different product types you can refer to the request *types*.

5. *date_in*: A datetime with the format YYYY-MM-DD that represents the date when the product was added to the freezer.

6. *period*: An integer that represents the period in months for which the product can stay in the freezer.

7. *box_num*: An 'integer' that represents the freezer box where to add the new product.

8. *prod_num*: An 'integer' that represents the an available identifier inside the selected box to identify the product inside this box.

9. *quantity*: An 'integer' that represents the quantity in terms of people for which the product can be consumed.

4.3.7   Update_product

This request implements a single method which is *POST*. It gives the possibility to update an existing product into a existing freezer for a specific user. A JSON object must be sent with a specific architecture. This method follows the same behaviours as the preceding *POST* methods previously described.

The prototype of the request is:

$< domainUrl > /update\_product/ < freezer\_id > / < box\_num > / < prod\_num > / < inside > / < token >$

Parameters:

1. \<domailUrl\>: The url of the server API. In the case of the project the server is run on the *localhost:5000*

2. \<freezer_id\>: An integer that represents the id of an existing freezer that belongs to the user with the token given. If we want to select all freezers that belongs to a user the id 0 must be specified.

3. \<box_num\>: An integer that identifies the box where the product is stored.

4. \<prod_num\>: An integer that identifies the product inside the box previously referred.

5. \<inside\>: An integer that specifies if the product is still in the freezer or not. If the integer given is strictly higher than 0 than it means it is inside otherwise it is outside.

6. \<token\>: A user's token that corresponds to an existing account.

The architecture of the JSON object is the same as the add_product request except that here we also has date_out field that takes a datetime with the YYYY-MM-DD format or null if there is no output date. The output date given must be equal or higher than the input date.

```
{
 "product_name": "name",
 "text_descr": "description",
 "freezer_id": number,
 "type_id": number,
 "date_in": datetime YYYY-MM-DD,
 "date_out": datetime YYYY-MM-DD,
 "period":" number,
 "box_num": number,
 "prod_num": number,
 "quantity": number
}
```

4.3.8 General_tendency

This request implements a single method which is *GET*. It gives the possibility to obtain a global view on the products stored and consumed by all people using the API. It allows to list the occurrence of each product type in the database following a descending order manner. This request follows the same pattern as the others *GET* methods explained above. If the token is correct a status code 200 is retuned with a JSON. Otherwise, a status code 400 is returned with a JSON explaining the reason of that error.

The prototype of the request is:

$$< domainUrl > /general\_tendency/ < token >$$

Parameter:

1. <token>: A user's token that corresponds to an existing account.

Example of the JSON object returned:

```
{
 [
  {
   "freq": 12,
   "type_id": 1,
   "type_name_en": "soup",
   "type_name_fr": "soupe"
  },
  {
   "freq": 6,
   "type_id": 24,
   "type_name_en": "ice-cream",
   "type_name_fr": "glace"
  },
  ...
 ]
}
```

Explanation of the JSON object:

1. *freq*: An integer that represents the number of time the product type appears in the database.

2. *type_id*: An integer that represents the identifier of an existing product type. TO know the different product types you can refer to the request *types*.

3. *type_name_en*: The name of the product type in English

4.3.9 Custom_tendency

This request implements a single method which is *GET*. It gives the possibility to obtain a personalized tendency of a specific based on his previous consumptions and the products that are in his freezers. The results are sorted following the latest date the product type remove from a user's freezer and the Descending order on the frequency. This request follows the same pattern as the others *GET* methods explained above. If the token is correct a status code 200 is retuned with a JSON. Otherwise, an error status is sent with a JSON object as explained before.

The prototype of the request is:

$$< domainUrl > /custom\_tendency/ < token >$$

Parameter:

1. <token>: A user's token that corresponds to an existing account.

Example of the JSON object returned:

```
{
 [
  {
```

```
  "freq": 12,
  "latest": "2018-02-05",
  "type_id": 1,
  "type_name_en": "soup",
  "type_name_fr": "soupe"
 },
 {
  "freq": 6,
  "latest": "2017-05-02",
  "type_id": 24,
  "type_name_en": "ice-cream",
  "type_name_fr": "glace"
  }
  ...
 ]
}
```

Explanation of the JSON object:

1. *freq*: An integer that represents the number of time the product type appears in the database.

2. *latest*: The date of the last product of that type that left the user's freezers.

3. *type_id*: An integer that represents the identifier of an existing product type. To know the different product types you can refer to the request *types*.

4. *type_name_en*: The name of the product type in English

4.4 Documentation

To understand how to use the REST API a website with a documentation has been made. The tool used to build this documentation is *MkDocs*[5]. This tool is commonly used to build documentation and are used for instance by *Keras*. Markdown is used to write the documentation and the tools translates it into static html page. It is really easy to use if you know the Markdown language.

This documentation explain what are the available requests that can be done, how to achieve a correct request and what are the results sent back. For each request an example using the *client URL request library (cURL)* is given. The choice of this tool has been made because it is cross-platform and not difficult to use. But it can be used with any other tool that can achieve http requests.
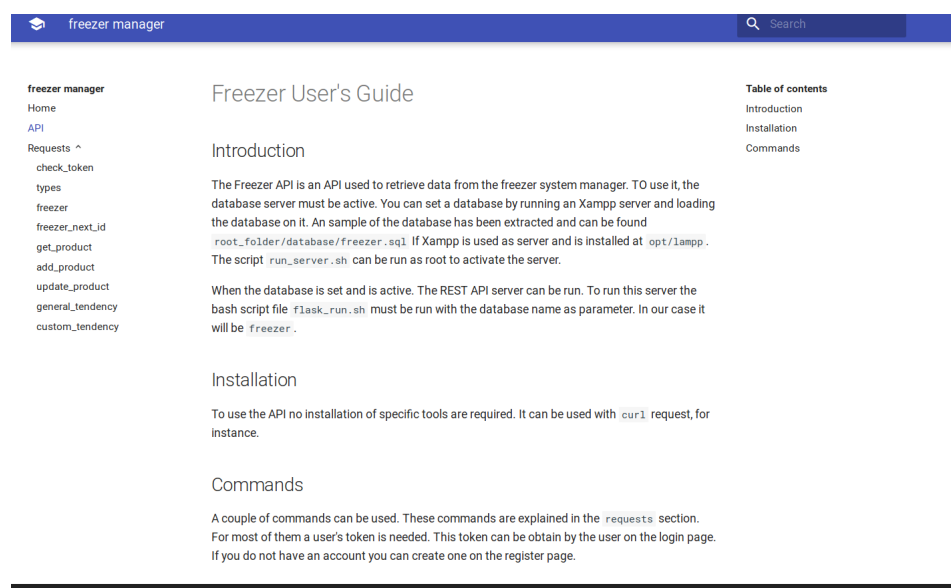
Figure 9: Documentation website

The documentation site can be found *documentation/api-documentation/site/index.html*

---

[5] https://www.mkdocs.org/

4.5 Unit tests

To test the different functionalities implemented with the REST API, some unit tests have been done. The list of tests done is not exhaustive and can be extended. The units tests essentially cover the routes of the requests, the status code returned depending on the request done, the JSON objects sent, the content-type sent, and the methods implemented.

The tests done are defined in the test folder *REST_api/tests/tests.py*. They are based on the *unittest* python module. The quantity of unit tests done is around 40 tests. The Figure 10 depicts the results of a portion of tests done. As you can see the time elapsed between the starting time of a request and the end time is fast. The unit tests *test_XX_XX_methods_not_implemented* are slower because several requests are done inside of them. Also note that the REST API is single threaded for this project and for production perceptive it will be multithreaded with the help of *gunicorn or uwsgi*. Then the performances observed will change.

Figure 10: Portion of unit tests done

5 External website

To demonstrate that the API can used by thirds applications. I developed a external application that only communicate with the REST API. The aim of this application is to provide to a user a user-friendly dashboard. To have access to the dashboard a user token is required and must be submitted here Figure 11.
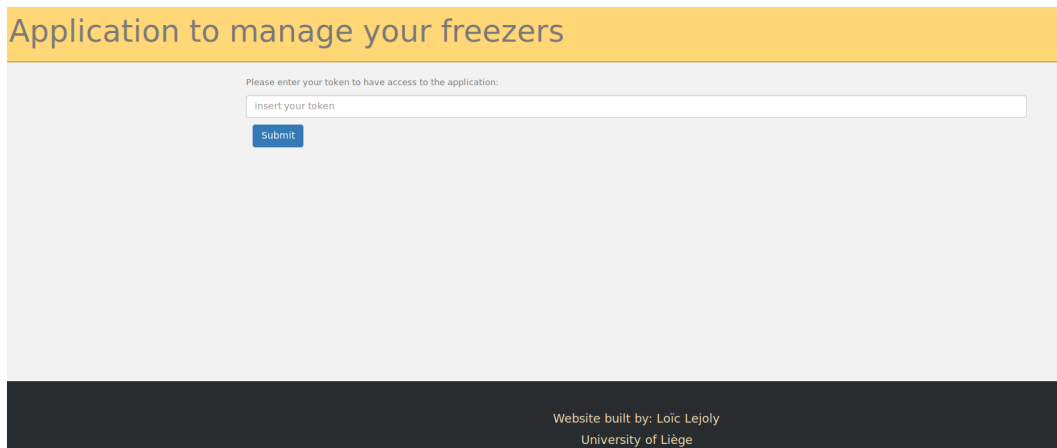
Figure 11: Dashboard of the external website before giving a token

If the token given is valid, four tiles will be displayed. Each tile refer to a specific section (*types of products, management of freezers, management of products, and trends*). The user has the possibility to display and hide each section. By default, all sections are hidden. If the user want to display one section it just needs to click on the display button and click again to hide the section.
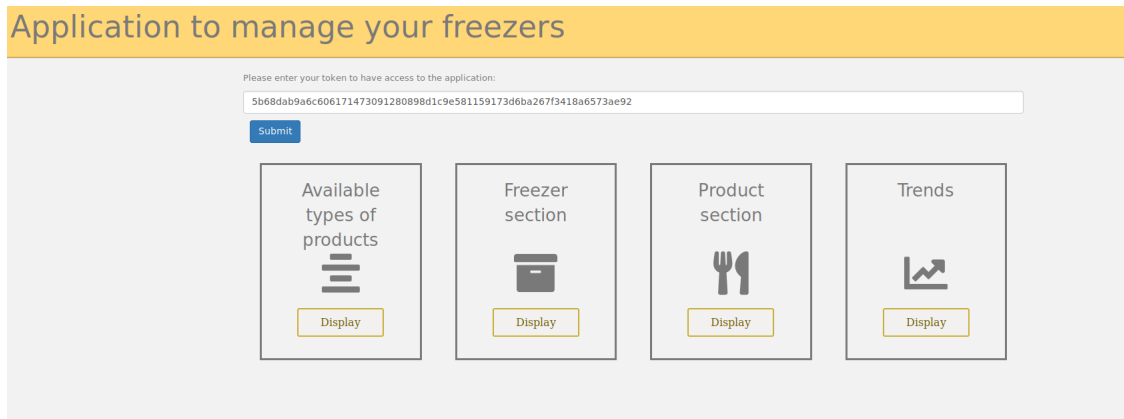
Figure 12: Dashboard of the external website for a valid token

5.1 Types tile

The first tile is dedicated to the different product types. When a user click on this tile he can see all possible types that can be encoded. He also have the possibility to see the name of product types in French and in English. The Figure 13 depicts the output displayed to the user when he clicks on types tile.

Figure 13: product types

5.2 Freezer tile

The freezer tile is dedicated to manipulations that can be done on user's freezers. The Figure 14 depicts the displayed to the user when he clicks on the freezer tile. As you can see, a user has the possibility to see the freezers that he has, the possibility to update an existing freezer, the possibility to add a freezer , and finally the possibility to remove a freezer.

Figure 14: freezers

5.3 Product tile

The product tile is dedicated to manipulations that can be done on user's products. The Figure 15 and Figure 16 depict the output to the user when he clicks on the freezer tile.

On the Figure 15 a user can select parameters for a specific freezer and the results are displayed. As you can see a row is in orange, this means that the product exceeded the period for which it could stay in the freezer. For this example the period was 1 month and the difference between the input date until now is bigger than one month.

Figure 15: Display products

The Figure 16 depicts the other available possibilities that a user has to manage his products. He can add new products to his freezers but he can also update existing products.

Figure 16: Functionalities on products

5.4   Trends tile

The trends tile is dedicated to display tendencies of users. The Figure 17 depicts the output to the user when he clicks on the trends tile. The general tendency simply reflects the global frequency of each product type. The personalized tendency reflects the personalized tendency of the user based on his previous consumptions and the products that he has on his freezers. As you can see on this figure there are some changes between the general tendency and the personalized one. It is because of several users are encoded then the general tendency is different.
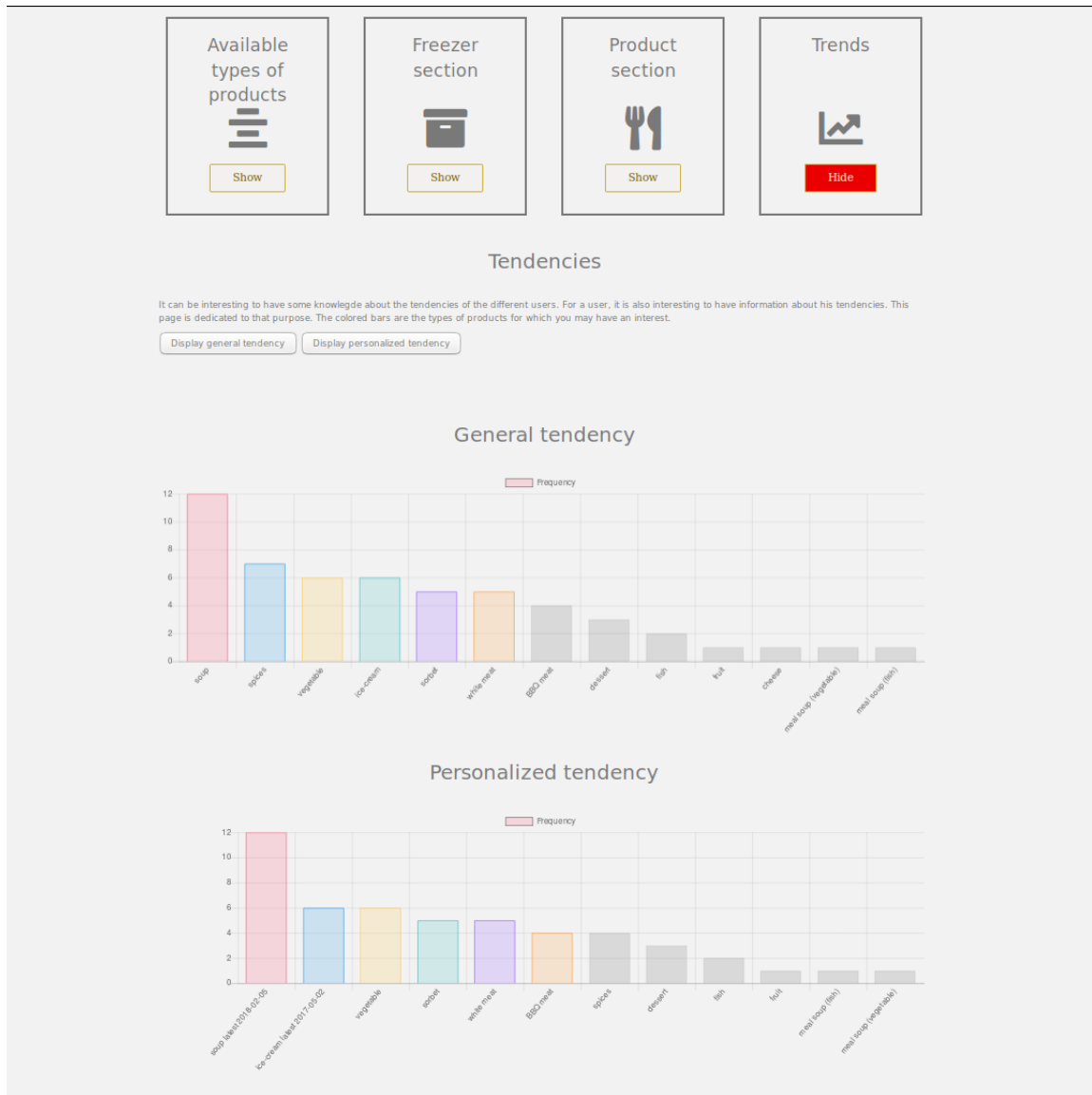
Figure 17: Trends tile

5.5 Website development

The external website is developed with several languages *CSS, HTML, JavaScript*. Moreover, some frameworks et tools are used. To have a responsive website, the containers of the *Bootstrap*[6] framework are used. *Bootstrap* is very well-known in the world of web development. It is one of the famous framework used to build the skeleton of a website. To have dynamic tables, I used a plug-in called *tablesorter*[7] developed with *JQuery* and finally for the plot generation I used *chart.js*[8]

The rest of the website is developed from scratch with pure *JavaScript* (no framework used). I used the pure *JavaScript* instead of framework like *JQuery* because I did not see the real benefits of a specific framework. Moreover, there are a plethora of *JavaScript* frameworks and most of them require some practice before using it.

---

[6]https://getbootstrap.com/
[7]https://mottie.github.io/tablesorter/docs/index.html
[8]https://www.chartjs.org/

6 Project challenges

6.1 Problems faced

I did not encounter big problems for this project. The most challenging parts were probably to constructs the different architectures. The database architecture took sometime to finally reach the one used in the project. I also thank Mr. Hiard for the advices he gave me concerning my database architecture. Concerning the architecture of the REST API built with flask, it is not an easy work to find the proper architecture since there is no a unique template for building it, it will depend on the problem to solve. With flask you can build an API with different manners. The direction that I took seems to be coherent but for a largest application I think working with *blueprints* decorators provided by flask can help because you can encompass routes that share the same pattern.

Concerning the external website the most challenging part is probably to work with *JavaScript* and the asynchronous behaviour brought by the AJAX requests. Moreover, working with pure *JavaScript* can help to understand how it is working but you need to write extra code to achieve the same job as a framework.

6.2 Work done

Concerning the work achieved for this project about 50 files have been written by myself and represent about 4000 lines of code (the third part libraries excluded). I spent lot of time to elaborate the database architecture since it is one of the most important part of the system developed.

Another major part of the system that took time is the REST API server. First of all, I needed to understand how to build a REST API and what are the different existing tools since it was the first time I used this technology. After that, I need to understand the flask framework and read the documentation given.

Concerning the development of the internal website It was not the most difficult part because I already used this technology. The development of the external website it is another story because I decided to only use JavaScript and I have rarely used this technology before. Thus, this part was also a challenging part that took time.

For the time spent on this project, it is really difficult to evaluate because I did not refer working hours spent on the project and it is probably a mistake. The only thing I can say is a spent a lot of time on it.

6.3 Future work

The future of this freezer manager system is probably to use an ORM to do the requests instead of using *MysqlDB* to allow an easier maintenance that could be done by people how do not know SQL language. The reason for not having chosen this method directly is justified by the fact that *MysqlDB* module gives me the possibility to use SQL language and it was a goal that I was looking for.

Another point is to migrate the system on the stable version of flask and mix it with *gunicorn or uwsgi*. It is also interesting to leave the localhost area and test the system on a real environment.

Another interesting point to develop in the future is to set a logger system that saves everything done on the server. In case of a Ddos attack or if a malicious people Who stole a user token and tries to destroy the data that the victim saved on its account. With the logger we can see where the requests come from.

The enrichment of the internal website *user and admin* part must developed by adding extra features for the user. Concerning the admin panel this one must be developed deeper by giving the possibility to the administrator to obtain useful information about users and obtain statistics (e.g: number of products that are store in the database, the average number of products stored by user,...).

An interesting thing, perhaps too optimistic, would be to put the project available for everyone and put the project as an open project where every one can contribute to the project and also to have feedback from others people.

7 Conclusion

This personal project is a good personal experience and it should be strongly advised to students because it allows to learn a number of things. First of all, it forces you to establish your own road map and to establish yourself a formulation of your project, the direction that will be explored and how you will get there.

As far as I am concerned, this personal project gives me the possibility to think about the different aspects of a project in general. The bounds to establish to avoid leaving the scope of the project and focuses all your time on a specific aspect of your project which is essential for a production project but does not really matter in the scope of a personal project.

In my case, I decided to not explore some technical aspects such as establishing an accurate data security (expiration on the token, reliable encryption of password, configuration of the different servers to be secure,...). The aim was not to neglect these technical aspects but to provide a functional project. The time allowed for the personal project course is limited, thus you need to make some choices concerning the aspects to develop. Moreover, this personal project gives me the possibility to use what I have learned in some courses and discover new things. For instance, how to build a server with flask. Finally, I think I will continue to develop this project for personal enrichment.