



## Master 1 Data sciences

Personal project:

Freezer manager

*Loïc Lejoly s141123*

Academic year 2017-2018

*June 2018*

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Explanation of the project itself . . . . .	2
<b>2</b>	<b>Content storing</b>	<b>3</b>
2.1	Database model discussion . . . . .	3
2.2	Relational database structure . . . . .	3
2.3	Entity-relationship model . . . . .	4
2.4	Relational model . . . . .	5
<b>3</b>	<b>Internal website</b>	<b>6</b>
3.1	Account creation . . . . .	6
3.2	User panel . . . . .	7
3.3	Admin panel . . . . .	8
<b>4</b>	<b>API REST</b>	<b>9</b>
4.1	Folder architecture . . . . .	9
4.2	Sever . . . . .	10
4.3	functionalities implemented . . . . .	10
4.3.1	check_token . . . . .	10
4.3.2	types . . . . .	11
4.3.3	Freezer . . . . .	11
	GET method: . . . . .	12
	POST method: . . . . .	12
	PUT method: . . . . .	12
	DELETE method: . . . . .	13
4.3.4	freezer_next_id . . . . .	13
4.3.5	get_product . . . . .	13
4.3.6	add_product . . . . .	15
4.3.7	update_product . . . . .	15
4.3.8	general_tendency . . . . .	16
4.3.9	custom_tendency . . . . .	17
4.4	Documentation . . . . .	17
<b>5</b>	<b>External website</b>	<b>18</b>
<b>6</b>	<b>unit tests</b>	<b>18</b>
<b>7</b>	<b>problemes rencontrés</b>	<b>18</b>
<b>8</b>	<b>future work</b>	<b>18</b>
<b>9</b>	<b>Conclusion</b>	<b>18</b>

# 1 Introduction

In the context of the personal project course given at ULiège. I decided to develop a smart freezer. This idea emerges due to a personal experience. Every year my family and I reorganize the home's freezers. I was a bit shocked when I saw the quantity of products that were in the freezers and for which I was not aware. This is a problem that anyone can potentially face one day.

The main problem of forgotten products is they cannot stay indefinitely in a freezer due to their physical properties. For instance after some time a meat staying for a long time in a freezer will not have a good taste. As a result, some products were thrown to the rubbish. We could avoid throwing products to the rubbish if we are aware of the existence of those products. The solution of that problem would be an application or a tool that gives the possibility to manage the content of your freezers and more. It is the direction that I took for my personal project.

## 1.1 Explanation of the project itself

The aim of the project is to provide a freezer system manager that gives to a user using this system some information about his products. This system also keeps an history of a user's products. Moreover, the main purpose of that system manager is to give the possibility for a third person to develop its own application and use the freezer system manager inside of it. It is a 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-world.

The information that the system manager needs to 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 informations are essential to obtain the token.
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.
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,...).
  - (e) A date of input and output
  - (f) A period during which a product may remain inside a freezer.
  - (g) An identification inside a freezer (e.g: the product number 2 in box 3 of the freezer number 1).

These informations 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.

As explained above, 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 freezer 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.

To get around this problem the interaction with the freezer 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 is that it is a client-server which means that 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 or for a smartphone. Since 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 to detect uniquely a user if it

is correctly generated. A schema given Figure 1 shows where the REST API is set.

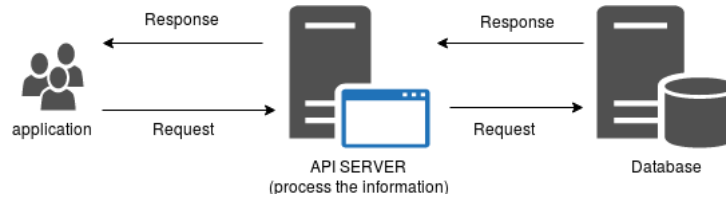


Figure 1: REST API interaction

The token generated is static but in general a token has an expiration date. I do not go into that direction because it becomes complex to manage such token with an API.

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 historic 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.

## 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 which means that 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 database. Finally, the relational database architecture is probably the architecture which is the easiest one to be ported into another database architecture like *Cassandra* or *MongoDB* or anything else.

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 need 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 system manager API and it is generated uniquely. It is also used to identify a user.

A user has also two additional information which 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 do no use it because I did not have enough time to find and test some of these algorithms. 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 defined the other objects that will be stored and linked to a user.

The database contains several types of products. These types of products 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 time 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 news columns in the database without affecting the architecture or using a translation tool to translate the existing 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 an 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 date of its entrance in the freezer. 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.

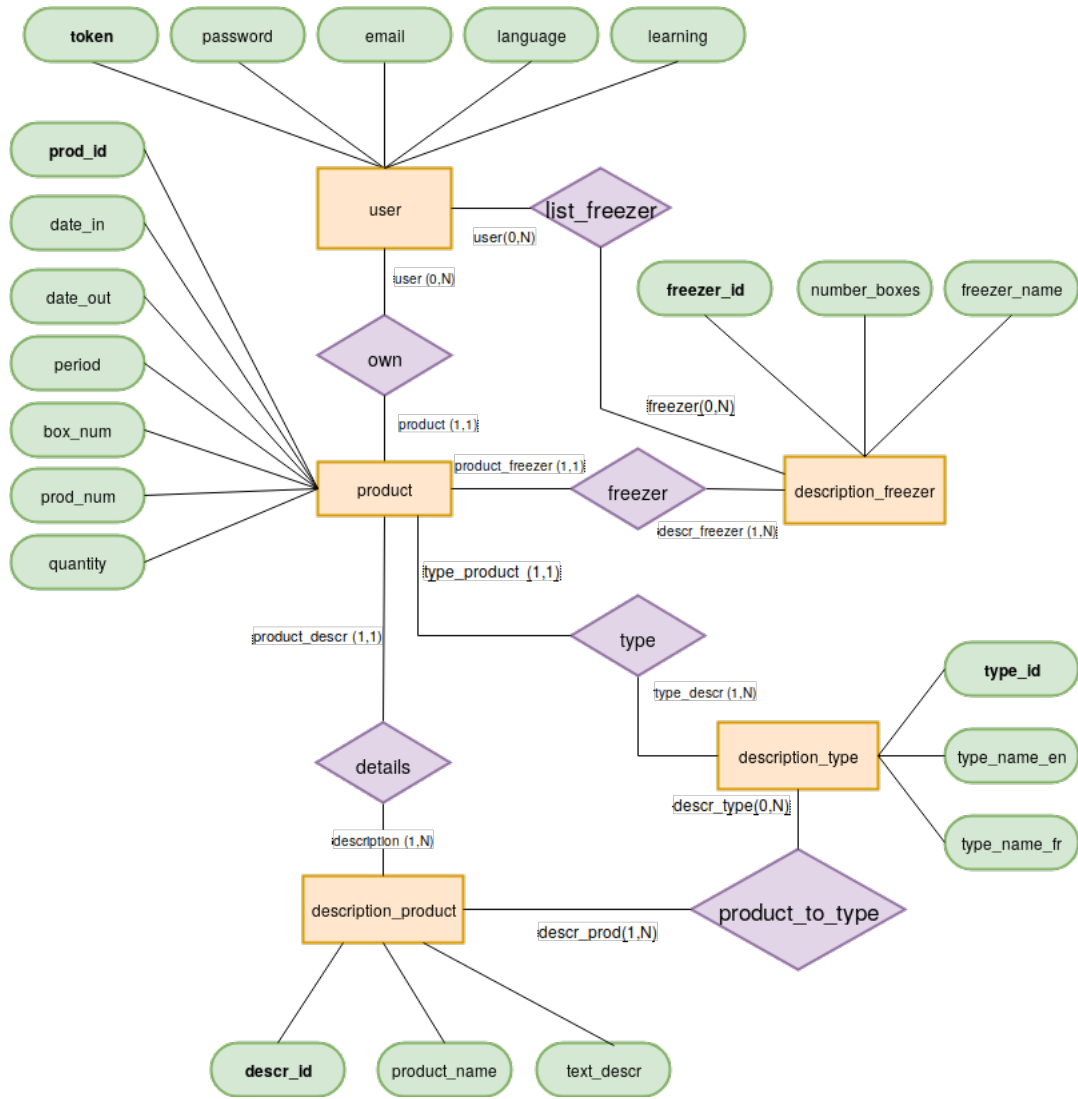


Figure 2: entity-relationship model of the database

## 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 is encoded in *MySQL* language. 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 *MySQL* 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.

User	
PK	<b>token</b> VARCHAR 128
	password VARCHAR 64
	email VARCHAR 64
	language VARCHAR 32
	learning LONGTEXT

User	
PK	<b>prod_id</b> INT(32)
FK	token VARCHAR 128
FK	descr_id INT(32)
FK	freezer_id INT(32)
FK	type_id INT(32)
	date_IN DATE
	period INT(32)
	date_out DATE
	box_bum INT(32)
	quantity INT(32)
	prod_num INT(32)

Description_type	
PK	<b>type_id</b> INT(32)
	type_name_en VARCHAR 128
	type_name_fr VARCHAR 128

Description_product	
PK	<b>descr_id</b> INT(32)
	product_name VARCHAR 256
	text_descr MEDIUMTEXT

Description_freezer	
PK	<b>freezer_id</b> INT(32)
	number_boxes INT(32)
	freezer_name VARCHAR 256

Product_to_type	
PK, FK	<b>descr_id</b> INT(32)
PK, FK	<b>type_id</b> INT(32)

List_freezers	
PK, FK	<b>freezer_id</b> INT(32)
PK, FK	<b>token</b> VARCHAR 128

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 table can be reconstruct with the others tables. Thus it is not mandatory to back-ups 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 startup 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 your 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. 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 to interact 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. To secure a website some files such as *.htaccess* or *php.ini* need to be correctly set in accordance with the server. 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 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.

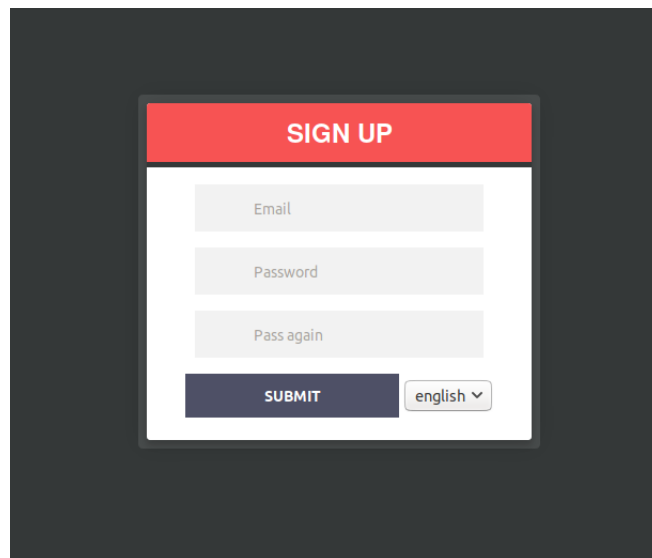
The image shows a registration form titled "SIGN UP" in a red header. Below the header are three input fields labeled "Email", "Password", and "Pass again". At the bottom of the form is a dark blue "SUBMIT" button and a language dropdown menu currently set to "english". The entire form is centered on a dark gray background.

Figure 4: register page

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. The third part applications should only asked your token and anything 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 6). If the information given are correct, he will be redirected to the user panel page *website/user/user\_interface.php*.

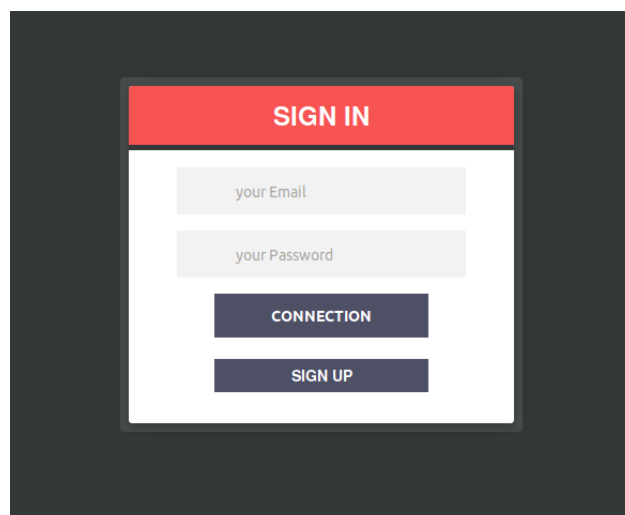
The image shows a login form titled "SIGN IN" in a red header. Below the header are two input fields labeled "your Email" and "your Password". At the bottom of the form are two dark blue buttons: "CONNECTION" and "SIGN UP". The entire form is centered on a dark gray background.

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.

Welcome *loiclejoly@gmail.com* to the user panel

**User panel**  
Welcome on the user panel. This panel is used to give you some information about your account. This interface is very simple at the moment and gives you just the minimum information that you need to use the freezer API with third part applications.

**Account information**  
Login: *loiclejoly@gmail.com*  
Token: *5b68dab9a6c606171473091280898d1c9e581159173d6ba26713418a6573ae92*  
Language: *fr*

**Change language**  
English  
language

**Change Password**  
current password  
new password  
new password again  
password

Site réalisé par: Loïc Lejoly  
Université de Liège

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 be too much work for the scope this course.

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 recommended. Instead of encoding these informations inside a php file it is recommended 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.

**Admin panel**

admin \*\*\*\* Connection

Figure 7: admin login page

When the identifier and the password given are correct. You will be redirected to the admin panel Figure 8. 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.

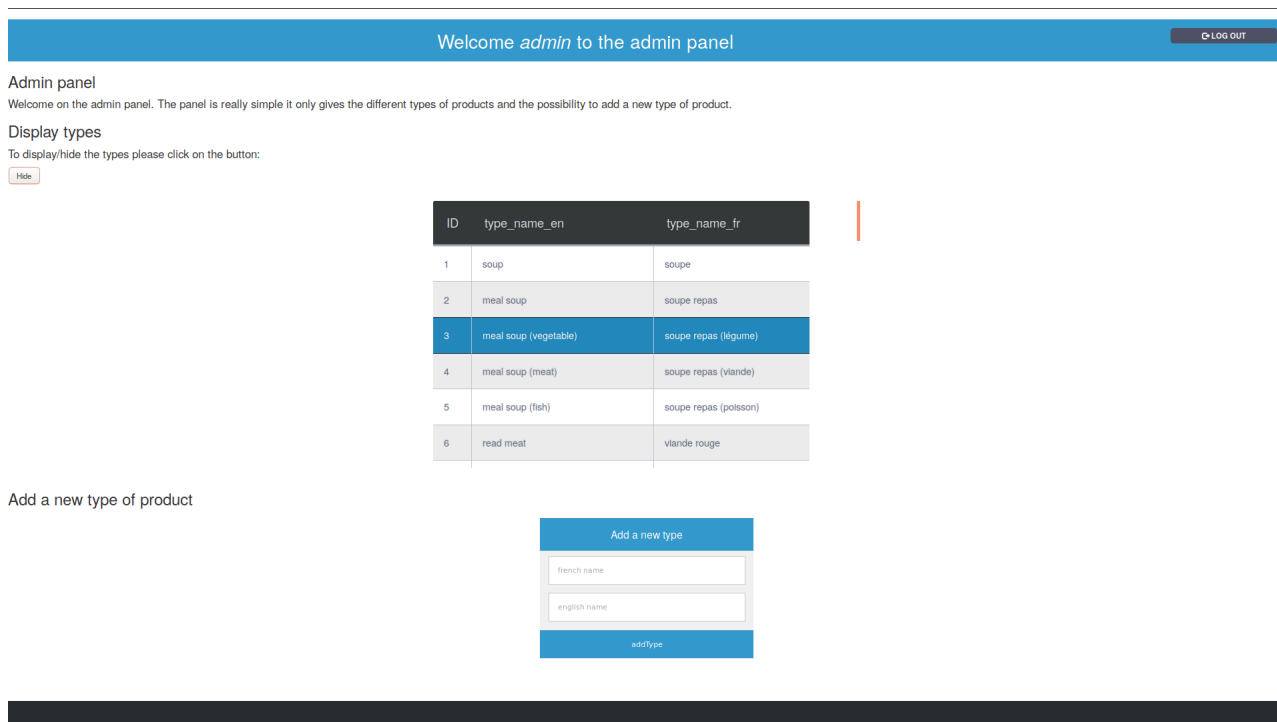


Figure 8: admin login page

## 4 API REST

A REST API is built on a server that establishes a communication between the database and the 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 need 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 browsers which specify in request headers the field *Origin* for some type of requests *POST*, the REST API needs to use cross-origin resource sharing (CORS). The CORS allows the REST API server to accept the ajax requests which are forbidden due to the same-origin security policy.

### 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. *mysqlRequests.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 sent 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 the mysql request. It avoids to bring inconsistencies in the database due to incorrectly done user requests.

## 4.2 Sever

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, Javascripts,...

A micro-framework is used to develop the Python server with Python language. This framework is Flask 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 (26th 2018) with a number of changes.

To interact with the mysql database I used the module MySQLDB <sup>1</sup>. 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 sqlalchemy 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 sqlalchemy 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.

For instance the URI format specified in the 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.

---

<sup>1</sup><https://github.com/PyMySQL/mysqlclient-python>

The prototype of this request is the following:

$\langle domainUrl \rangle /check\_token/ \langle token \rangle$

Parameters:

1.  $\langle domainUrl \rangle$ : The url of the server API. In the case of the project the server is run on the *localhost:5000*
2.  $\langle token \rangle$ : 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:

$\langle domainUrl \rangle /types/ \langle token \rangle$

Parameters:

1.  $\langle domainUrl \rangle$ : The url of the server API. In the case of the project the server is run on the *localhost:5000*
2.  $\langle token \rangle$ : 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 Freezer

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:

$\langle domainUrl \rangle /freezer/ \langle token \rangle$

Parameters:

1.  $\langle domainUrl \rangle$ : The url of the server API. In the case of the project the server is run on the *localhost:5000*
2.  $\langle token \rangle$ : 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 exisitng 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. *<domainUrl>*: 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. *<domainUrl>*: 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\u00e9e 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 give 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. *<domainUrl>*: 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 YYYY-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 give 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\_i > / < box\_num > / < inside > / < token >*

Parameters:



1. <domainUrl>: 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. <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.
5. <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 expect 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 requests follows the same pattern as the others *GET* methods explained above. If the token is correct a status code 200 is returned 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 > /global\_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"
    },
    {
      "freq": 6,
      "type_id": 24,
      "type_name_en": "ice-cream"
    },
    ...
  ]
}
```

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 requests follows the same pattern as the others *GET* methods explained above. If the token is correct a status code 200 is returned 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*. 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.

## 5 External website

To demonstrate that the API can be used by third applications. I developed an external application that only communicates 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.

If the token given is valid, four tiles will be displayed. Each tile refers 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 wants to display one section it just needs to click on the display button and click again to hide the section.

concernant la section produits parler de la détection des produits ayant dépassé la période recommandée

The external website is developed with several languages *css, html, javascript*. Moreover, some frameworks and tools are used. To have a responsive website, the containers of the bootstrap framework are used. Bootstrap is very well known in the website world. It is one of the famous frameworks used to build the skeleton of a website. To have dynamic tables I used a plug-in called tablesorter developed with jquery and finally for the plot generation I use chart.js

The rest of the website is developed from scratch with pure javascript (no framework used). I used the pure javascript instead of a 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.

Ce veut être une application "externe" à l'API REST et à la DB. Peut être vu comme une société externe utilisant un service

Le but de cette application est d'avoir une sorte de dashboard bcp de javascript pur javascript pur bootstrap utilisé essayé de le rendre convivial et intuitif tablesorter utilisé pour table interactive 4 panel différent qui peuvent être caché ou affiché permet d'afficher en rouge dans les produits ceux qui ont dépassé la date limite

## 6 unit tests

## 7 problèmes rencontrés

l'architecture n'est pas la plus adéquate pour faire les unit tests. de plus on peut lire que l'instauration d'une API est assez compliquée à mettre en place car il n'existe pas de template spécifique cela dépend du problème que l'on a traité. sur le site flask une autre approche d'architecture est proposée

## 8 future work

Faire passer l'API sur un support ORM comme sqlalchemy qui permettra à l'API d'être maintenue plus facilement etc. Le fait de ne pas directement avoir choisi cette méthode m'a permis d'utiliser le langage SQL est c'était un but que je recherchais.

faire passer l'application sur la version stable 1.0 tester l'API sur des terminaux comme les smartphones et autres.

URI template des requêtes pourrait être uniformisé pour respecter au mieux les requêtes.

mettre en place le logger system qui permet de garder une trace de tout ce qui a été fait sur le serveur. En cas d'attaque du serveur par exemple ou si une personne mal intentionnée et ayant dérobé un token on peut avoir une trace d'où a été émise la requête mettre en place le système sur un serveur hors du localhost

Mettre en place l'application pour qu'elle puisse tourner en multithreads

Peut être fort optimiste mais lancer l'api en production ou premièrement la rendre accessible aux autres et avoir leur feedbacks.

## 9 Conclusion