My ToDo Lists

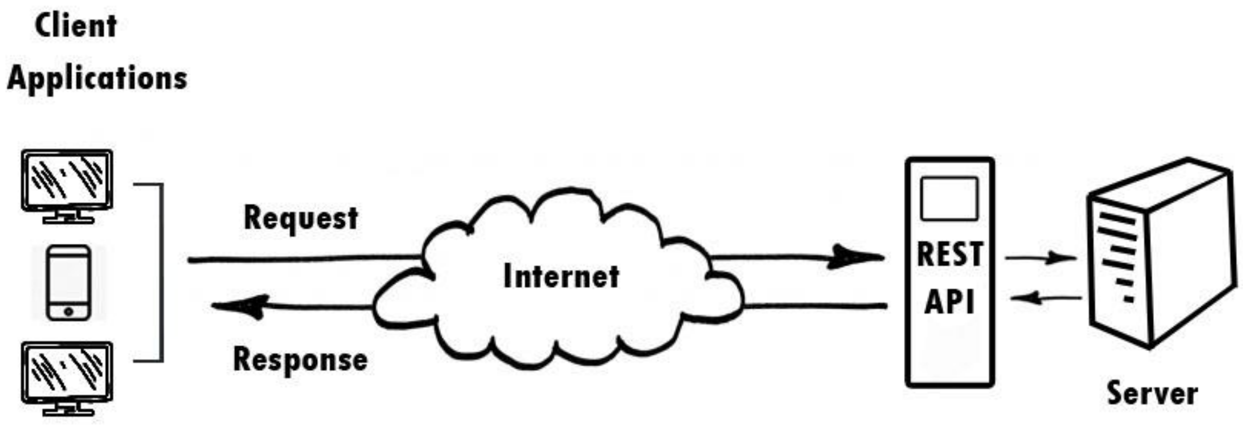
MyToDoList is a web application which allows users to manage their tasks efficiently. To do this, firstly they can register themselves and login into the homepage where they find their todo lists. Moreover, our web application is designed to be very intuitive and allows to the final user to interact and move from a section to another easily without having to get lost or confused.

Responsive design in home page and in dashboard
When the window is reduced and its width is less than 1000 pixels, the navigation side bar disappears and is switched into a burger menu. Also, images are put in the background in order to display the same content in the page but in a smart way. To do this, we used CSS properties like the Flexbox module to lay out, align and distribute space among items in their container.
For scalability purposes, we created a dashboard component, which is the same in a few pages. It corresponds to a fully sized section, which have a title, some icons, and data related to what we can find in the dashboard such as the number of displayed tasks, and the members who are participating. Inside this big component, we find other components depending on the page where the user is. For example, there is the Kanban model components (To do – Doing – Done) in the Kanban dashboard component.

All the front-end pages are done (including responsive). The only missing page is the account page (/dashboard/account).
The data are for the moment written in raw in the components datas and can be inconsistent.
Moreover, the permissions management and the user login/registration are functional on the front end with the VueX implementation.

As for the backend of the project, all the data models (User, ToDoList, Rubric, Task) and their relationship are implemented in the backend for the serialization of the objects and in the MySQL database.
Moreover, the hash of the password in the database and the generation of a token specific to the user with JWT is functional. The token is currently valid for one week. This will certainly be redefined for security reasons.

<u>The relationship between the back and the front is as follows:</u>


Client
Applications
Request
Response
Internet
REST
API
Server

FRONT END

Showcase page

First, on the front end, we have implemented a showcase and explanatory page of our project with a contact section (not yet functional). This showcase page is fully responsive. Moreover in the header of this page we find navigation tabs to be redirected to anchors of the same page (contact or about) or to the login page.

Login and Register

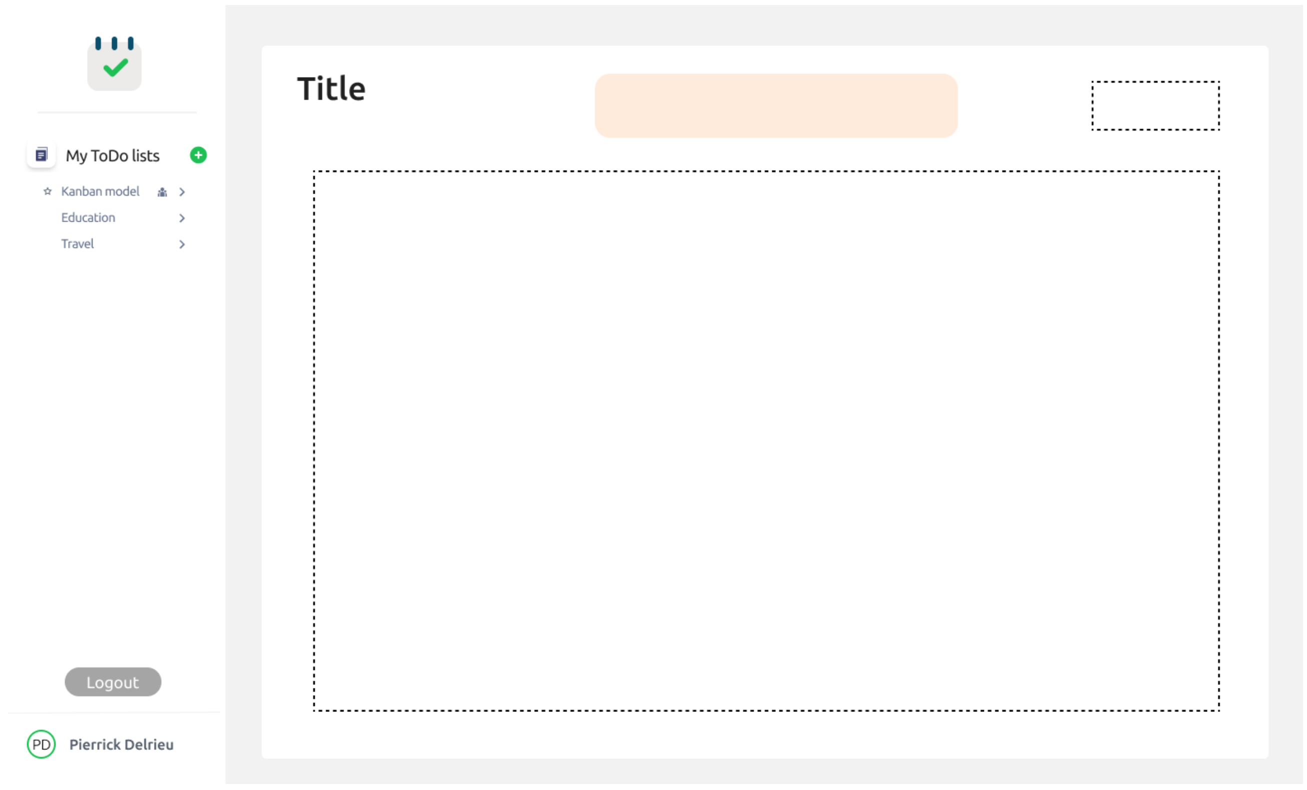
The login page contains a form. The login is done with two fields: email and password. We will implement later the functionality of password recovery by email. Moreover, the login page displays an error if the connection did not work. This page also contains a redirection to the registration page.
Similarly, the registration page contains a form but with additional fields: firstname, lastname, and password confirmation. This page contains a JavaScript animation that makes the user aware of the security of his password with 5 levels of security. In addition, a color bar located under the password entry indicates the level of robustness of the password. This indicator changes from a red level to a green level, which is easy for the user to understand. As the user enters more characters, the size of the colored bar increases to a higher level of strength. To reach the top level, the user must enter a password with at least 8 characters, including upper and lower case, numbers and special characters. An error is also displayed if the registration attempt failed (for example if the email is already in use).
These two pages are implemented at the following URLs: /, /login and /register. The code is structured as components for each reusable object like user input fields, form submit button, header ...

Authentication

Finally, once the authentication is done, the user can access all the urls of /dashboard. Before authentication, the attempt to access one of the urls of the Dashboard returns a 404 page. The authentication has been managed with VueX. Indeed, we store in the LocalStorage of the browser and use this token during our calls to the api in order to identify the user and send him his lists and information. For security reasons we do not want to store the user's token in the cookies because when sending a request, it can be intercepted.

Dashboard

For the dashboard part, we have created a component with the following architecture:



This allowed us not to duplicate the code between all the dashboard pages. Indeed, there will always be the sidebar on the left and a page on the left with content. It should be noted that the sidebar is itself a component.
Moreover, this component is responsive. The sidebar is displayed as a burger menu and the content page takes the full width of the page.

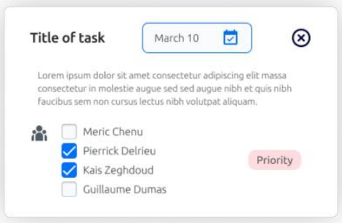
This component is called for /dashboard, /dashboard/list/:id and /dashboard/account (our only 3 dashboard url).

Rubrics and tasks

Lists and rubrics being repetitive elements, we created a task component and a rubric component.

Modal for CRUD

In addition, in order to implement the CRUD model for all our back end objects, we created modals (popUp page) on the front end.


Title of task
March 10
Lorem ipsum dolor sit amet consectetur adipiscing elit massa consectetur in molestie augue sed sed augue nibh et quis nibh faucibus sem non cursus lectus nibh volutpat aliquam.

- ☐ Meric Chenu
- ☑ Pierrick Delrieu
- ☑ Kais Zeghdoud
- ☐ Guillaume Dumas

These pages are closed when clicking outside or clicking on the closing cross. So we have a modal component and several subcomponents of modal as NewTask, NewRubric, EditTask ...
There is also the confirmModal subcomponent which allows to check the user action when deleting a task or a rubric and to make sure that it was not an error.

Linking the front-end and the back-end

In order to link our front-end and our database, it is necessary to have an intermediary: the back-end. In order to call the backend, we have created services for each of our models that allow to send post, get, update or delete requests to the backend server. These services are called during the assembly of the component, or the view so that the data is loaded in the datas before the display of the component.

Moreover, the call to the backend can take time. During our development phase we were not aware of this. But it is necessary to create a loading page which will have to be displayed during the loading of the data. Even if this time is often negligible.

The loading page is not yet implemented but the component itself is developed.

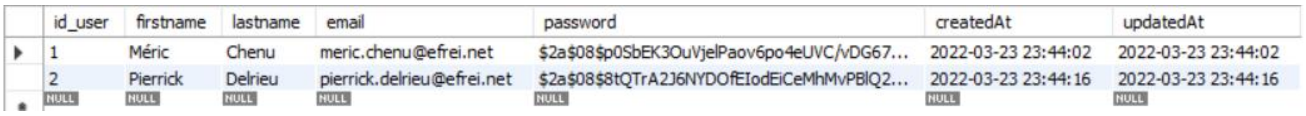
BACK END

During the development of our API, we installed several libraries. One of those we used is called nodemon, and it allows our server to display the changes made to it dynamically. Another important library is express. In the broad lines, express facilitates the development in node.js, accelerating the development. Finally, one of the most important is cors. This library allows us to implement a security on the incoming requests in our API. To do so, we have set the url of our frontend server on port 8080, in order to open it only to our frontend server.

Database

The data that we find on all applications, whether they are web or software, need data in order to function properly. This data itself needs a place to be stored. Where else could we store it but in a database?
2 types of databases exist, and many databases are derived from them. Here, we wanted to focus on the SQL model and the MySQL language.

In order to check the data in our database, we used the mysql workbench tool, whose data from our User model (we'll come to that in a moment) is displayed below:

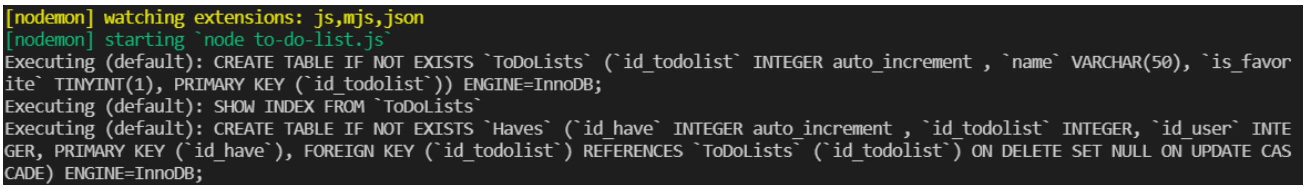


	id_user	firstname	lastname	email	password	createdAt	updatedAt
▶	1	Méric	Chenu	meric.chenu@efrei.net	$2a$08$p0SbEK3OuVjelPaov6po4eUVC/vDG67...	2022-03-23 23:44:02	2022-03-23 23:44:02
	2	Pierrick	Delrieu	pierrick.delrieu@efrei.net	$2a$08$8tQTrA2J6NYDOfEIodEiCeMhMvPBlQ2...	2022-03-23 23:44:16	2022-03-23 23:44:16
●	NULL	NULL	NULL	NULL	NULL	NULL	NULL

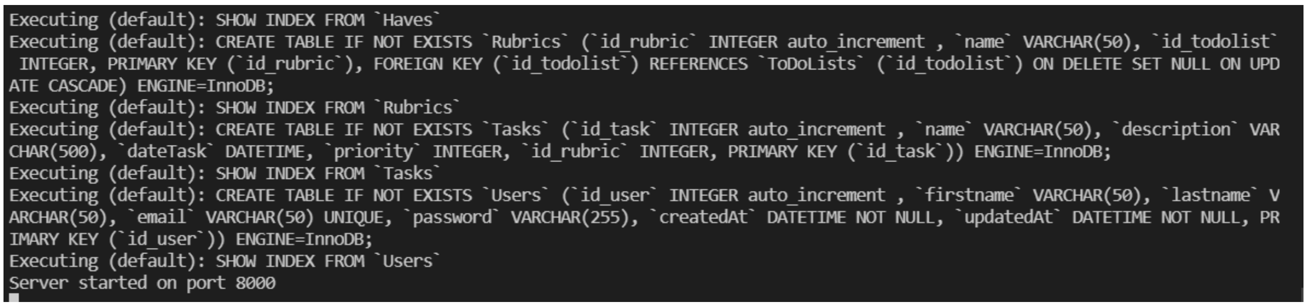
We can therefore observe several attributes of our model:
- Id, primary key of our model
- Firstname
- Lastname
- Email (which is unique)
- Password (which is hashed, we'll come to that later)
- Creation and modification time

Nevertheless, how to create templates from our API?
At the beginning, we were going to create SQL queries by hand, as it is possible in node.js. However, we quickly realized how long the task would take, and we decided to use the Sequelize library.

This library allows us to automate the sql queries, supported by Sequelize, by respecting its syntax. For example, when our server is launched, our different models, filled in our important javascript file sequelize, are created if they are not present in the database.

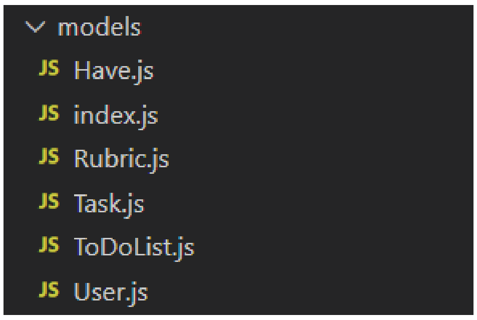


```
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node to-do-list.js`
Executing (default): CREATE TABLE IF NOT EXISTS `ToDoLists` (`id_todolist` INTEGER auto_increment , `name` VARCHAR(50), `is_favor
ite` TINYINT(1), PRIMARY KEY (`id_todolist`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `ToDoLists`
Executing (default): CREATE TABLE IF NOT EXISTS `Haves` (`id_have` INTEGER auto_increment , `id_todolist` INTEGER, `id_user` INTE
GER, PRIMARY KEY (`id_have`), FOREIGN KEY (`id_todolist`) REFERENCES `ToDoLists` (`id_todolist`) ON DELETE SET NULL ON UPDATE CAS
CADE) ENGINE=InnoDB;
```



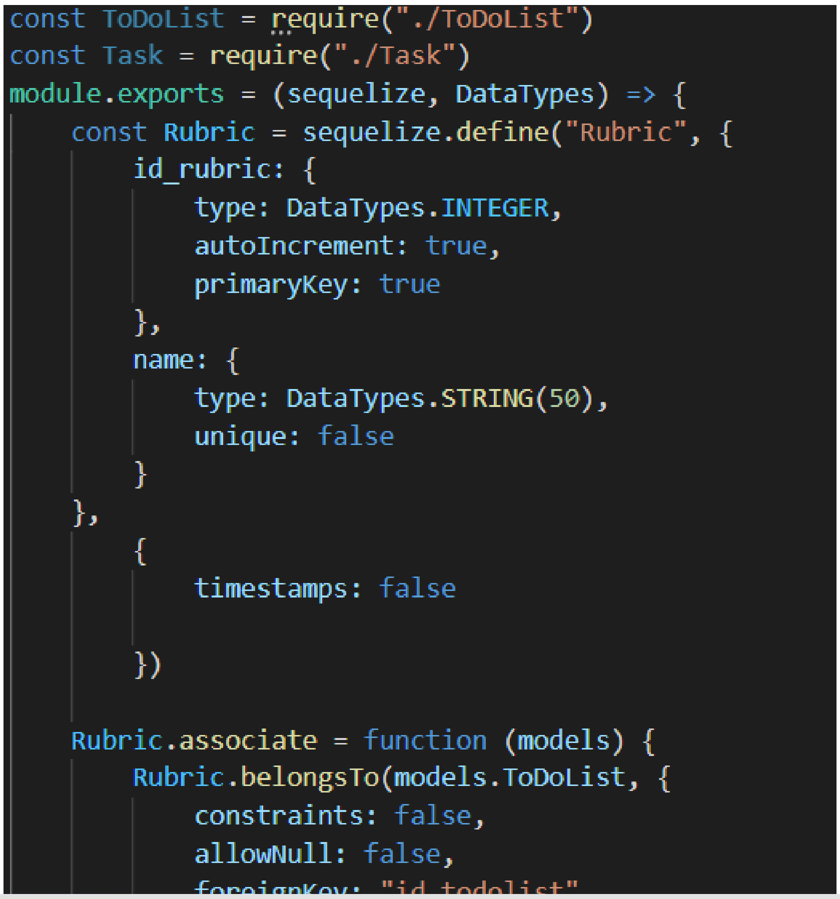
```
Executing (default): SHOW INDEX FROM `Haves`
Executing (default): CREATE TABLE IF NOT EXISTS `Rubrics` (`id_rubric` INTEGER auto_increment , `name` VARCHAR(50), `id_todolist`
 INTEGER, PRIMARY KEY (`id_rubric`), FOREIGN KEY (`id_todolist`) REFERENCES `ToDoLists` (`id_todolist`) ON DELETE SET NULL ON UPD
ATE CASCADE) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `Rubrics`
Executing (default): CREATE TABLE IF NOT EXISTS `Tasks` (`id_task` INTEGER auto_increment , `name` VARCHAR(50), `description` VAR
CHAR(500), `dateTask` DATETIME, `priority` INTEGER, `id_rubric` INTEGER, PRIMARY KEY (`id_task`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `Tasks`
Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id_user` INTEGER auto_increment , `firstname` VARCHAR(50), `lastname` V
ARCHAR(50), `email` VARCHAR(50) UNIQUE, `password` VARCHAR(255), `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PR
IMARY KEY (`id_user`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `Users`
Server started on port 8000
```

As we can see above, at the launch of our server, our User, Haves, Rubrics, Tasks and ToDoLists templates are created if they do not already exist. Moreover, their properties are filled in. At the code level, in order to fill in our models, we have created a models folder in our API that looks like the following:



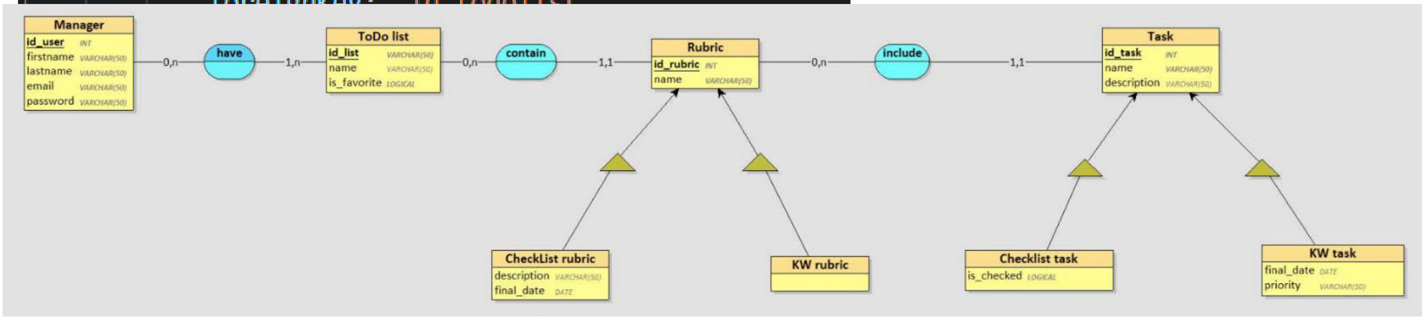
```
models
  Have.js
  index.js
  Rubric.js
  Task.js
  ToDoList.js
  User.js
```

Our index.js file is the one containing our different configurations needed to link sequelize and our API.



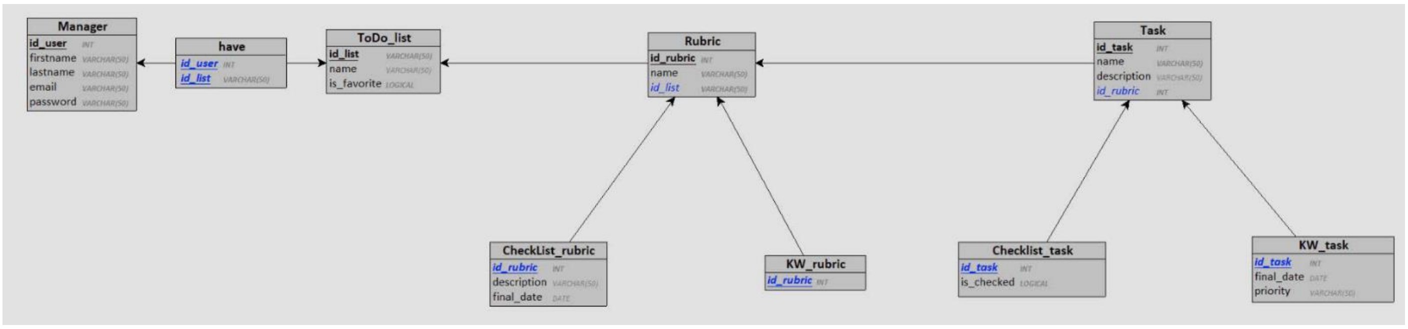
```
const ToDoList = require("./ToDoList")
const Task = require("./Task")
module.exports = (sequelize, DataTypes) => {
    const Rubric = sequelize.define("Rubric", {
        id_rubric: {
            type: DataTypes.INTEGER,
            autoIncrement: true,
            primaryKey: true
        },
        name: {
            type: DataTypes.STRING(50),
            unique: false
        }
    },
        {
            timestamps: false

        })

    Rubric.associate = function (models) {
        Rubric.belongsTo(models.ToDoList, {
            constraints: false,
            allowNull: false,
            foreignKey: "id_todolist"
```

If we take a deeper look at the organization of one of our models within our code, we notice that we define the name of our model and its properties via the "define" method of sequelize.
The type attribute, we fill in the type of the column.
We can also create ManyToMany and belongsTo relationships.
In our case, we notice that our Rubric model belongs to a Todolist.
Let's see this on our LDM and MCD to see it more clearly

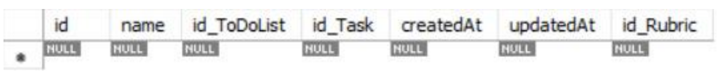


In our CMD, we have a Manager class (called Users in the database).
A manager has 0 or more ToDo lists, and a ToDoList belongs to 1 or more managers (we wanted to be able to share our todolist with other users, but this feature could not be implemented due to lack of time). A ToDo list contains 0 or more rubric, and a rubric is contained in a Todo list. Finally, a task is included in a rubric, and a rubric includes 0 or more tasks (Here, we don't have to take into account the inheritances on our CMD).

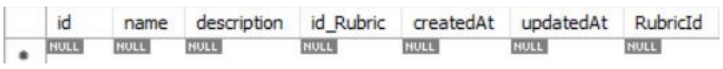
Below, we can see the organization of our MDL



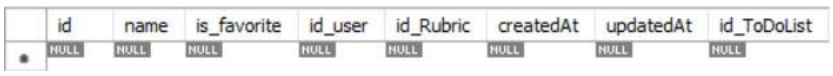
finally, we can have a short overview of the fields of our templates in our database, under mysql Workbench :



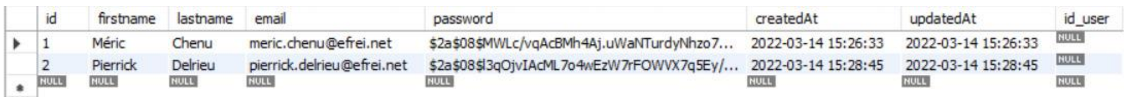
	id	name	id_ToDoList	id_Task	createdAt	updatedAt	id_Rubric
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL



	id	name	description	id_Rubric	createdAt	updatedAt	RubricId
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL



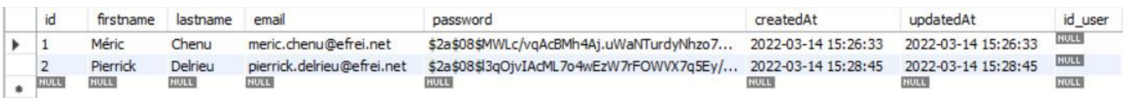
	id	name	is_favorite	id_user	id_Rubric	createdAt	updatedAt	id_ToDoList
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL



	id	firstname	lastname	email	password	createdAt	updatedAt	id_user
▶	1	Méric	Chenu	meric.chenu@efrei.net	$2a$08$MWLc/vqAcBMh4Aj.uWaNTurdyNhzo7...	2022-03-14 15:26:33	2022-03-14 15:26:33	NULL
	2	Pierrick	Delrieu	pierrick.delrieu@efrei.net	$2a$08$l3qOjvIAcML7o4wEzW7rFOWVX7q5Ey/...	2022-03-14 15:28:45	2022-03-14 15:28:45	NULL
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

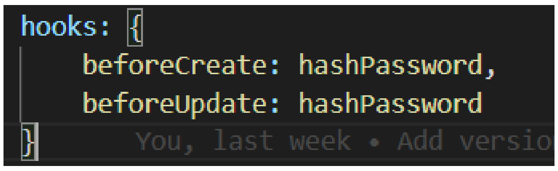
Authentication

When registering a user in our database, we hash his password for security reasons. As a result, as we can see below, the password appears unreadable:



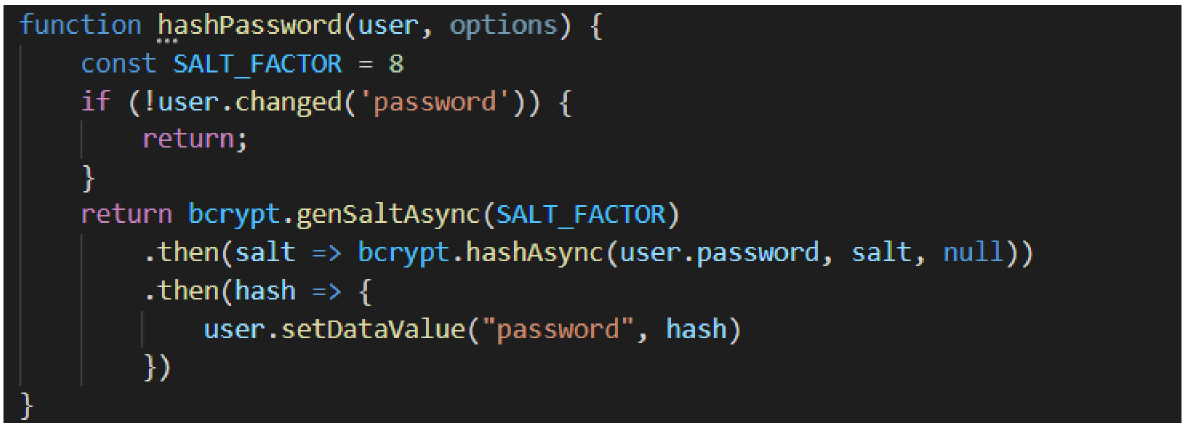
	id	firstname	lastname	email	password	createdAt	updatedAt	id_user
▶	1	Méric	Chenu	meric.chenu@efrei.net	$2a$08$MWLc/vqAcBMh4Aj.uWaNTurdyNhzo7...	2022-03-14 15:26:33	2022-03-14 15:26:33	NULL
	2	Pierrick	Delrieu	pierrick.delrieu@efrei.net	$2a$08$l3qOjvIAcML7o4wEzW7rFOWVX7q5Ey/...	2022-03-14 15:28:45	2022-03-14 15:28:45	NULL
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

The purpose of a hash function is to transform a readable text into an unreadable text, with an impossibility to find the readable form (unlike encryption). The only way to find a match is to hash a word in plain text and then compare the result.



```
hooks: {
    beforeCreate: hashPassword,
    beforeUpdate: hashPassword
}       You, last week • Add versio
```

when a user registers, we send his data to our user model, which, before being created, calls the hashPassword function.

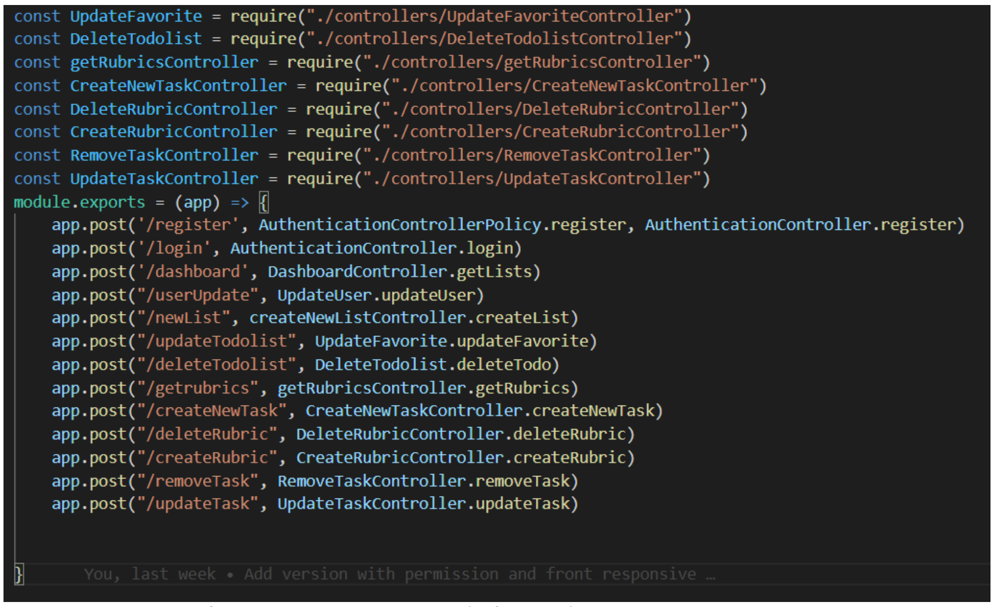


```
function hashPassword(user, options) {
    const SALT_FACTOR = 8
    if (!user.changed('password')) {
        return;
    }
    return bcrypt.genSaltAsync(SALT_FACTOR)
        .then(salt => bcrypt.hashAsync(user.password, salt, null))
        .then(hash => {
            user.setDataValue("password", hash)
        })
}
```

As we can see above, this function uses the bcrypt-nodejs library to hash passwords.
When the user registers, we generate a token, available for 7 days, which we store in the browser's localStorage, for page permission purposes. Indeed, if the user's token is generated, it means that he is logged in and therefore has access to pages restricted to logged-in users. We generate this token using the jsonwebtoken library.

Routes

Within our API there are many routes that we can see below in our routes.js file:



```
const UpdateFavorite = require("./controllers/UpdateFavoriteController")
const DeleteTodolist = require("./controllers/DeleteTodolistController")
const getRubricsController = require("./controllers/getRubricsController")
const CreateNewTaskController = require("./controllers/CreateNewTaskController")
const DeleteRubricController = require("./controllers/DeleteRubricController")
const CreateRubricController = require("./controllers/CreateRubricController")
const RemoveTaskController = require("./controllers/RemoveTaskController")
const UpdateTaskController = require("./controllers/UpdateTaskController")
module.exports = (app) => {
    app.post('/register', AuthenticationControllerPolicy.register, AuthenticationController.register)
    app.post('/login', AuthenticationController.login)
    app.post('/dashboard', DashboardController.getLists)
    app.post("/userUpdate", UpdateUser.updateUser)
    app.post("/newList", createNewListController.createList)
    app.post("/updateTodolist", UpdateFavorite.updateFavorite)
    app.post("/deleteTodolist", DeleteTodolist.deleteTodo)
    app.post("/getrubrics", getRubricsController.getRubrics)
    app.post("/createNewTask", CreateNewTaskController.createNewTask)
    app.post("/deleteRubric", DeleteRubricController.deleteRubric)
    app.post("/createRubric", CreateRubricController.createRubric)
    app.post("/removeTask", RemoveTaskController.removeTask)
    app.post("/updateTask", UpdateTaskController.updateTask)


}       You, last week • Add version with permission and front responsive …
```

When calling one of these routes, we go to our controller folder, in order to call our different files and their specific methods to perform actions on our database. These methods return a result that we get back on the front side as a response. Below you can find some routes, their responses and their data.
https://workflowy.com/s/routes-of-api-http-r/QrCW3EZ6oQg19y30

OPENING

Hosting of the site

During our project, we expressed the wish to publish our site on the web. However, due to lack of time, some features such as sharing todolist could not be implemented, preventing us from publishing it. Nevertheless, we aim to have all these features in place by the end of the week, and thus publish our site.