Real Time Systems – Bus project

Introduction

My project was to implement a website to manage a bus company, with the possibility for clients to reserve bus seats.

The main difficulty of this project is the number of languages it requires : PhP, SQL, JavaScript (js), Html, CSS, and finally, Cpp.

To implement this website, I used the Laravel framework. It allows greater possibilities than coding all the website from scratch. It facilitates the communication between html, php, js, css and sql.

Up to this point, the actual difficulty is to handle real time system with Cpp. In fact, it could have been achieved by Php, that has tools similar to Cpp mutex. But, for this project, the aim was to implement it in Cpp. And it is at this moment that comes the higher difficulty of this project: made the website and the Cpp code communicate with each other. I will explain how I managed to do that later in this report.

☞ Observation 1

The pictures can be a bit small. You can find the original ones in the following folder :
bus_project/Report/Figs
They are named *fig{x}*, in the order of the report.

☞ Observation 2

This website needs a server to run, the Cpp part and the database part too. I can't let them run all the time because I have an other website I'm working on runnning on my serser, and it does not like when both are launched... If you want to visit the website by yourself on your browser, please feel free to send me an email, and I will turn it on a few hours. I hope you'll understand.

☞ Observation 3

I've tried to put the code into the report, but sometimes, it is not very easy to read. It might be easier for you to open the good file in parallel.

☞ Observation 4 — Important – Link to the github repository

You can find all the files here: click here

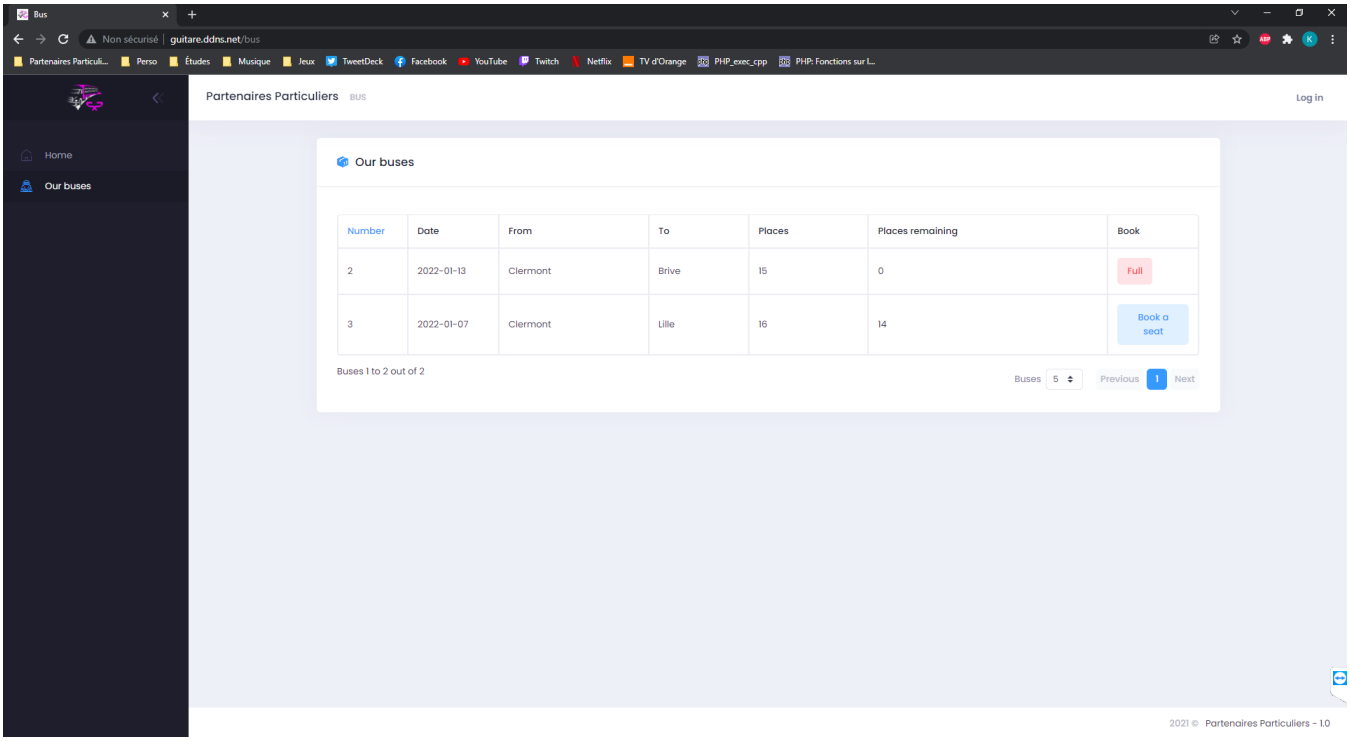
It should be public so that you can read and download files. If you don't, feel free to email me.

CONTENTS

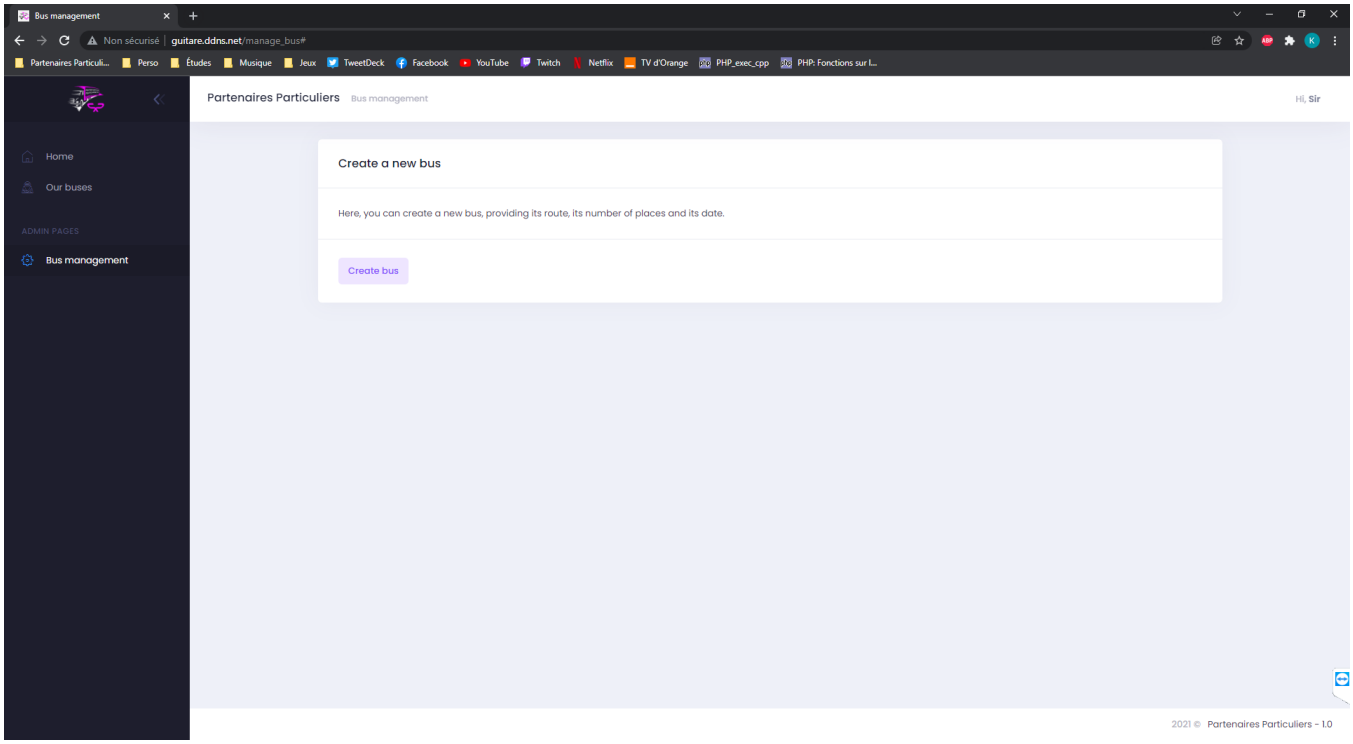
I) The global architecture of the website

1) A website...

The website is composed of two public pages : the home page, and the bus page. The home page is not very useful. It is just to present the company. However the bus page is the main page of the website: it allows the user to see the current buses and to reserve seats.

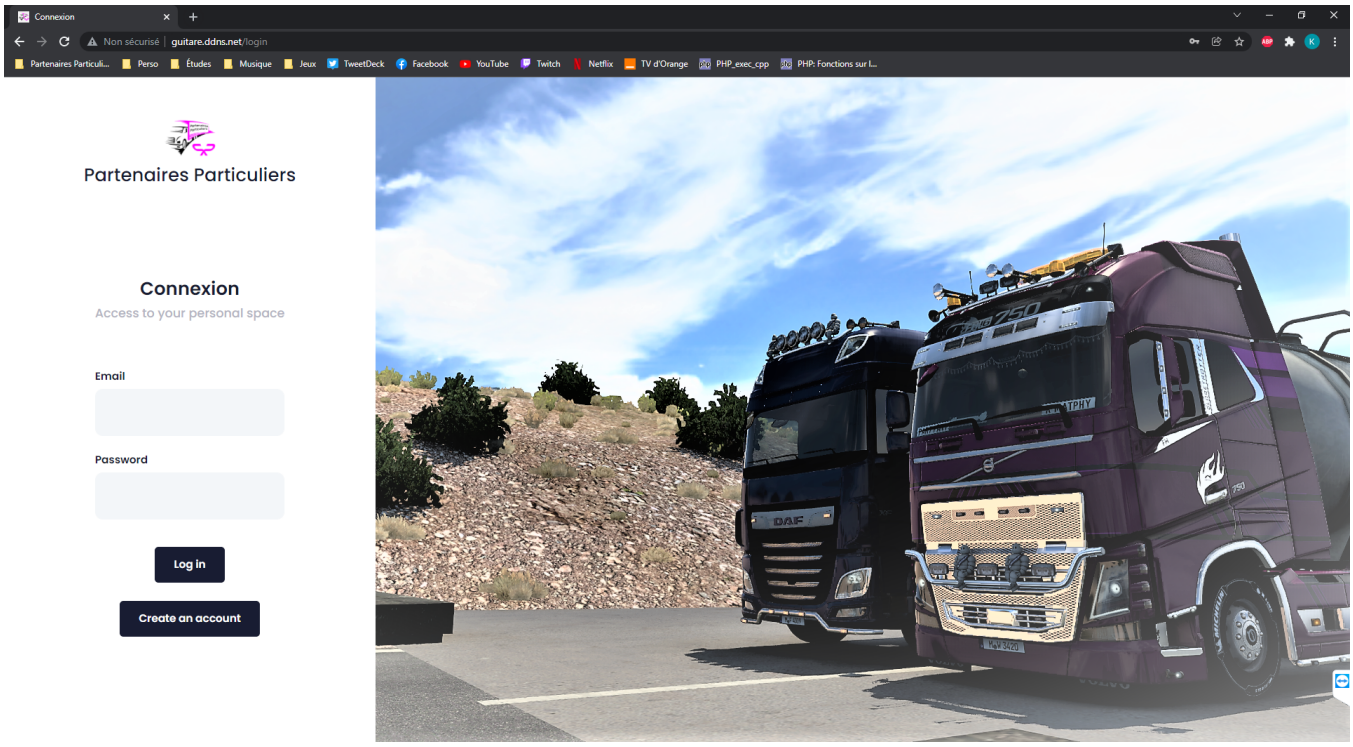


There is also a other page called *Manage bus*, that is for the owner of the company (that I will call Admin). This page allows the admin to create a bus.



As you may have noticed, there is a difference in the navigation list between the both images: the manage bus page is not visible if your are not an admin. It is handled by a system of permissions.

But to know if you are an admin or not, you need to have an account. So, there is a page dedicated to login and registration, accessible by the Log in button, on the right.



In case you don't have any account yet, you can register here :

☞ Observation 5

You may ask yourself why there are trucks here, instead of buses. The fact is that I already have a working website about a virtual trucks company. I've been working on this website for one year now, so I reuse some pieces of my code to save some time for this project. It why there are Trucks, or why the name is a French one.

2) ...but not alone

To make this website work well, it requires some other things. Indeed, we talked about accounts, permissions, buses... All of these have to be stored somewhere. It is why we need a database. This database will store all the data required by the website.
The other important thing to have is the Cpp server, that will handle the synchronisation part.

II) The database

The database is a MySQL database.

1) Create the database

I have worked a lot with database (db), and although there are some softwares than provide way to work on db without any SQL commands, I think that is more efficient and rigorous to use only commands. So to create the database, I used a script that you can find here:
bus_project/DB/Creation_Script.sql

The first part of the script is the following:

```
drop table if exists bus;
drop table if exists tokens;
drop table if exists permissions;
drop table if exists operations;
drop table if exists user_details;
drop table if exists users;
```

In fact, it is the last part that we write when you are doing this script, because it is a kind of protection part in case of failure when you execute the script. What is it doing? The word *drop* means delete, so the first line will delete the table bus. But there is *if exists*. So, it checks if the table bus exists. If yes, it drops it, if no, it does nothing. This part ensures you that you are not messing up your tables and that there are clean.

The next part is the creation of the tables:

```
CREATE TABLE users (
    iduser int not null,
    email varchar(100),
    grade int not null,
    password varchar(200)
);

CREATE TABLE user_details (
    iduser_detail int not null,
    users_iduser int not null,
    firstname varchar(50),
    lastname varchar(50)
);

CREATE TABLE operations (
    idoperation INT NOT NULL,
    name VARCHAR(255),
    description VARCHAR(255),
    route VARCHAR(255)
);

CREATE TABLE permissions (
    idpermission INT NOT NULL,
    grade INT,
```

```
    operations_idoperation INT
);

CREATE TABLE tokens (
    idtoken INT NOT NULL,
    users_iduser INT,
    token VARCHAR(255)
);

CREATE TABLE bus (
    idbus int not null,
    date date not null,
    from_city varchar(50) not null,
    to_city varchar(50) not null,
    place_number int not null
);

CREATE TABLE books (
    idbook int not null,
    bus_idbus int not null,
    users_iduser int not null
);
```

SQL is quite transparent. *CREATE TABLE* means that we create the table bus. Then, between the parenthesis, we define all the attributes of the table. I will explain for the bus table:

- idbus, which is an integer and cannot be null.
- date, which is saved as a date format and cannot be null.
- from_city, which is a varchar(50), that is a string that can be 0-50 long and cannot be null.
- to_city, which is a varchar(50), that is a string that can be 0-50 long and cannot be null.
- place_number, which is an integer and cannot be null.

Up to this point, we have our tables. But if we want them to work efficiently, we need to declare two things:

- the primary key of each table: the attribute that is unique for each row, and allows to identify the data.
- the foreign key if necessary: it is when a table refers to the primary key of another table, like for exemple, in the table books, bus_idbus that refers to idbus of the table bus. Foreign keys are very helpful to do joins between tables.

We do that this way:

```
ALTER TABLE users ADD CONSTRAINT pk_users PRIMARY KEY (iduser);

ALTER TABLE user_details ADD CONSTRAINT pk_user_details PRIMARY KEY (iduser_detail);
ALTER TABLE user_details ADD CONSTRAINT fk_user_details_users FOREIGN KEY (users_iduser
    ) REFERENCES users(iduser);

ALTER TABLE operations ADD CONSTRAINT pk_operations PRIMARY KEY (idoperation);

ALTER TABLE permissions ADD CONSTRAINT pk_permissions PRIMARY KEY (idpermission);
ALTER TABLE permissions ADD CONSTRAINT fk_permissions_operations FOREIGN KEY (
    operations_idoperation) REFERENCES operations(idoperation);

ALTER TABLE tokens ADD CONSTRAINT pk_tokens PRIMARY KEY (idtoken);
```

```
67 ALTER TABLE tokens ADD CONSTRAINT fk_tokens_users FOREIGN KEY (users_iduser) REFERENCES
       users(iduser);
68
69 ALTER TABLE bus ADD CONSTRAINT pk_bus PRIMARY KEY (idbus);
70
71 ALTER TABLE books ADD CONSTRAINT pk_book PRIMARY KEY (idbook);
72 ALTER TABLE books ADD CONSTRAINT fk_books_bus FOREIGN KEY (bus_idbus) REFERENCES bus(
       idbus);
73 ALTER TABLE books ADD CONSTRAINT fk_books_users FOREIGN KEY (users_iduser) REFERENCES
       users(iduser);
```

In this script, the declarations are done table by table. The first line declares the primary key, and then, if necessary, the foreign keys.

2) Tables glossary

a) Table users

In this table, we will have the registration data for each user:

- iduser, the id of the user,
- email, the email of the user,
- grade, the level of rights (0: normal user, 1: admin),
- password, the password of the user.

☞ Observation 6

For security purpose, the password is encoded before being stored. For exemple, if the password is "real-time systems", in the db, we will have a hash version of the password.

b) Table user_details

In this table, we will have the personal data of each user.

- iduser_detail, the primary key.
- users_iduser, the foreign key that refers to the primary key of the table users.
- firstname, the firstname of the user.
- lastname, the lastname of the user.

☞ Observation 7

I could have put this data into the users table, because it's a little website with not so much data. But I learned from my other project that is better to keep the registration data together and to add another table with personal data.

c) Table operations

This table is used to define the actions people can do on the website. We have:

- idoperation, the primary key.
- name, name of the operation.
- description, description of the operation (not mandatory but useful with huge projects).
- route, the route that is concerned by the operation (can be null). I'll explain what a route is later.

d) Table permissions

This table says which grade can do which operation. We have:

- idpermission, the primary key.
- grade, the concerned grade.
- operations_idoperation, the foreign key refering to operations.

e) Table tokens

When you are logged into the website, you have a unique token. It is stored here:

- idtoken, the primary key.
- users_iduser, the foreign key that refers to users.
- token, the token itself.

f) Table bus

The table that has all the data of a bus:

- idbus, the primary key.
- date, the date of the bus.
- from_city, the city the bus comes from.
- to_city, the city the bus will go.
- place_number, the number of places in the bus.

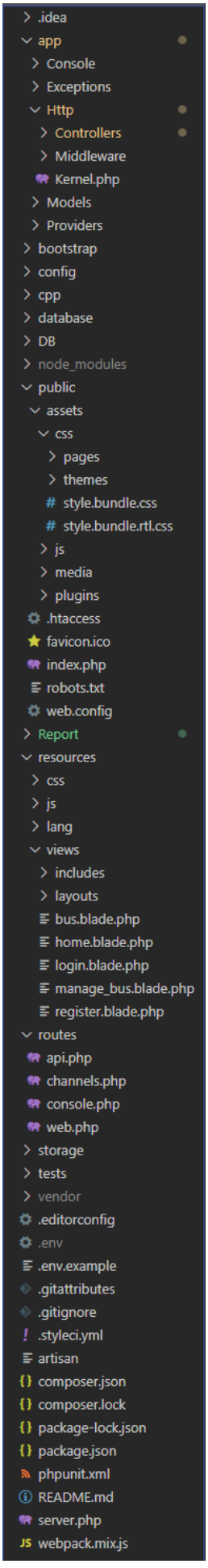
g) Table books

This table will contains all the seats bought by the users. Each time the user by a seat, it will add a new line. We have:

- idbook, the primary key.
- bus_idbus, the foreign key that refers to bus.
- users_iduser, the foreign key that refers to users.

III) The website part

1) Global architecture using Laravel Framework



As you can see, there are many folders and files to make all of this work.

There are some important aspects about developing this kind of application:

- the pages themselves: there are stored in *bus_project/resources/views*. They are .blade.php files.
- the routes: it's like an itinerary for each data, each command, each function. The ones I've done are in two files : *bus_project/routes/api.php* and *bus_project/routes/web.php*
- the CSS file: it contains all the parameters about the graphic interface. *bus_project/public/assets/css/style.bundle.css*
- the controllers: there are php files where are the main functions I've implemented about my application. The idea is to call them from the page file using AJAX. I'll talk about it later. *bus_project/app/Http/Controllers*
- the middleware: there are like controllers but more deeper in the application. While controllers do things like reserve a seat, the middleware will take care about connections, permissions, stuffs like this. *bus_project/app/Http/Middleware*
- the models: there are files which represent the tables of the db. There are used to help the website communicate with the db. *bus_project/app/Models*

☞ Observation 8

You can find some scripts about the db in *bus_project/DB*.
You can find the cpp file in *bus_project/cpp*.

2) The login system

a) The login page

To understand the login part, the important part of the login.blade.php is the following:

```
57                      <!--begin::Signin-->
58                      <div class="login-form login-signin py-11">
59
60                          @if(isset(Auth::user()->pseudo))
61                              <script>window.location="/auth/success";</script>
62                          @endif
63
64                          @if ($message = Session::get('error'))
65                              <div class="alert alert-danger alert-block">
66                                  <button type="button" class="close" data-dismiss="alert
    ">x</button>
67                                  <strong>{{ $message }}</strong>
68                              </div>
69                          @endif
70
71                          @if (count($errors) > 0)
72                              <div class="alert alert-danger">
73                                  <ul>
74                                      @foreach($errors->all() as $error)
75                                          <li>{{ $error }}</li>
76                                      @endforeach
77                                  </ul>
78                              </div>
79                          @endif
80
81                          <!--begin::Form-->
82                          <form class="form" method="post" action="{{ url('/auth/check')
    }}">
83                          {{ csrf_field() }}
84                              <!--begin::Title-->
85                              <div class="text-center pb-8">
86                                  <h2 class="font-weight-bolder text-dark font-size-h2
    font-size-h1-lg">Connexion</h2>
87                                  <span class="text-muted font-weight-bold font-size-h4">
    Access to your personal space</span>
88                              </div>
89                              <br>
90                              <br>
91                              <!--end::Title-->
92                              <!--begin::Form group-->
93                              <div class="form-group">
94                                  <label class="font-size-h6 font-weight-bolder text-dark
    ">Email</label>
95                                  <input class="form-control form-control-solid h-auto py
    -7 px-6 rounded-lg" type="text" name="email" autocomplete="off" />
96                              </div>
97                              <!--end::Form group-->
98                              <!--begin::Form group-->
99                              <div class="form-group">
100                                 <div class="d-flex justify-content-between mt-n5">
101                                     <label class="font-size-h6 font-weight-bolder text-
    dark pt-5">Password</label>
102                                 </div>
103                                 <input class="form-control form-control-solid h-auto py
    -7 px-6 rounded-lg" type="password" name="password" autocomplete="off" />
104                             </div>
105                             <!--end::Form group-->
```

```
106                             <!--begin::Action-->
107                             <div class="text-center pt-2">
108                                 <input type="submit" name="login" class="btn btn-dark
    font-weight-bolder font-size-h6 px-8 py-4 my-3" value="Log in" />
109                             </div>
110                             <div class="text-center pt-2">
111                                 <a href="{{route('register')}}" class="btn btn-dark
    font-weight-bolder font-size-h6 px-8 py-4 my-3" id="Create_account">Create an
    account</a>
112                             </div>
113                             <!--end::Action-->
114                         </form>
115                         <!--end::Form-->
116                     </div>
117                     <!--end::Signin-->
```

The first part (l.60 - l.79) is checking if the user has already been logged. If yes, it redirects to the home page, with as a logged user. Otherwise, it displays the error (for example, wrong password).
Then, we can find the form group, that is the part where we enter our email and password. Then, we can submit it (l. 108).
If we do not have account now, we can use the button Create an account (l. 111).

☞ **Observation 9**

About the link to the creation of account, we can say that we call the route register, instant of calling directly a page.

b) Login management

The main part of the login system is directly handled by Laravel. So, I won't talk about it. We just have to know that we can access to the connected user by *Auth::user()*.

c) Creating an account

This process was completly done by hand, so I will explain it. Once you arrive to the register page, you can see a form. In Php/Html, we have:

```
62                         <div class="text-center pb-8">
63                             <h2 class="font-weight-bolder text-dark font-size-h2 font-
    size-h1-lg">Registration</h2>
64                             <span class="text-muted font-weight-bold font-size-h4">
    Create your account</span>
65                         </div>
66                         <br>
67                         <br>
68                         <!--end::Title-->
69                         <!--begin::Form group-->
70                         <div class="form-group">
71                             <label class="font-size-h6 font-weight-bolder text-dark">
    First Name</label>
72                             <input class="form-control form-control-solid h-auto py-7
    px-6 rounded-lg" type="text" id="first_name" autocomplete="off" />
73                         </div>
74                         <!--end::Form group-->
75                         <!--begin::Form group-->
76                         <div class="form-group">
77                             <label class="font-size-h6 font-weight-bolder text-dark">
    Last Name</label>
78                             <input class="form-control form-control-solid h-auto py-7
    px-6 rounded-lg" type="text" id="last_name" autocomplete="off" />
```

```
79                          </div>
80                          <!--end::Form group-->
81                          <!--begin::Form group-->
82                          <div class="form-group">
83                              <label class="font-size-h6 font-weight-bolder text-dark">
    Email</label>
84                              <input class="form-control form-control-solid h-auto py-7
    px-6 rounded-lg" type="text" id="email" autocomplete="off" />
85                          </div>
86                          <!--end::Form group-->
87                          <!--begin::Form group-->
88                          <div class="form-group">
89                              <div class="d-flex justify-content-between mt-n5">
90                                  <label class="font-size-h6 font-weight-bolder text-dark
     pt-5">Password</label>
91                              </div>
92                              <input class="form-control form-control-solid h-auto py-7
    px-6 rounded-lg" type="password" id="password" autocomplete="off" />
93                          </div>
94                          <!--end::Form group-->
95                          <!--begin::Action-->
96                          <div class="text-center pt-2">
97                              <button class="btn btn-dark font-weight-bolder font-size-h6
     px-8 py-4 my-3" id="Create_account" onclick="Create_user()">Create an account</
    button>
98                          </div>
```

As you can see, every form is implemented as an html <input>. Each of them has an id. It's important for them to have a different id.

At the end, we have the Create an account button. What is important with it is the *onclick = "Create_user()"*. It means that when you click on the button, it will call the js function declared in the script part at the end of the file, which is:

```
128 <script>
129
130     function Create_user(){
131
132         $('#Create_account').addClass('spinner spinner-white spinner-right');
133
134         $.ajax({
135             url: "{{url('user/create')}}",
136             type: 'POST',
137             headers: {
138                 'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
139             },
140             data:{
141                 'first_name': $('#first_name').val(),
142                 'last_name': $('#last_name').val(),
143                 'email': $('#email').val(),
144                 'password': $('#password').val(),
145             },
146             success: function (response) {
147                 console.log(response);
148                 window.location.replace("{{route('login')}}")
149             }
150         });
151
152     }
153
154 </script>
```

Here we are! The first AJAX function. What is it exactly? It's a way to send data from the client side (the js function) to the server side (the php controller).

The first line add a spinner to the button, so that the user sees something is happening.

Then, we define the AJAX itself:

- the url is the route to the controller function.
- the type is what we are doing: POST => send to the server, GET => receive from the server.
- headers: the token. Here it's a little bit tricky because the user does not have a token yet.
- data: all the attributes we want to pass to the php function.
- the success function: what does the js function after the ajax function responded ? Here it redirects to the login page.

☞ Observation 10 — Warning!

An AJAX function is an asynchronous function!

☞ Observation 11

The js (aka the client) does not have any access to the database. It's only the php (aka the server). So, we need to transfer the data to the php.

☞ Observation 12

Let's see the code of the route, in the web.php file:

```
51 Route::post('user/create', 'App\Http\Controllers\Auth\CreateUserController@Create');
```

How is it built? It's a route so Route::. Then, it's a post type, so post. Then the name of the route ('user/create'), then the address of the controller. Finally, after the @, it is the name of the function inside the controller file.

Let's see the controller function now, in *bus_project/app/Http/Controllers/Auth/CreateUserController.php*:

```
3  namespace App\Http\Controllers\Auth;
4
5  use App\Http\Controllers\Controller;
6  use Illuminate\Http\Request;
7  use Illuminate\Support\Facades\Auth;
8  use Illuminate\Support\Facades\Hash;
9  use App\Models\Users as Users;
10 use App\Models\UserDetails as UserDetails;
11 use App\Models\UserToken as UserToken;
12
13 class CreateUserController extends Controller
14 {
15     function Create(Request $request)
16     {
17
18         $first_name = $request->input('first_name');
19         $last_name = $request->input('last_name');
```

```
20          $email = $request->input('email');
21          $password = $request->input('password');
22
23          if(!ctype_alnum($password)) return [
24              "error" => "Password not alphanumerical",
25              "err_message" => "Error: password should countain A-Z, a-z, 0-9 characters
    only."
26          ];
27
28          $login = Users::where('email', $email)->first();
29
30          if ($login) {
31              return [
32                  "error" => "Already exists",
33                  "err_message" => "Error: this user already exists."
34              ];
35          }
36
37          {
38              $lastIdUser = Users::orderBy('iduser', 'desc')->first()->iduser;
39
40              $user = new Users;
41              $user->iduser = $lastIdUser + 1;
42              $user->email = $email;
43              $user->grade = 0;
44              $user->password = Hash::make($password);
45              $user->save();
46              $user->refresh();
47
48              $lastIdDetails = UserDetails::orderBy('iduser_detail', 'desc')->first()->
    iduser_detail;
49
50              $userdet = new UserDetails;
51              $userdet->iduser_detail = $lastIdDetails + 1;
52              $userdet->users_iduser = $lastIdUser + 1;
53              $userdet->firstname = $first_name;
54              $userdet->lastname = $last_name;
55              $userdet->save();
56              $userdet->refresh();
57
58              $lastIdToken = UserToken::orderBy('idtoken', 'desc')->first()->idtoken;
59
60              $token = $this->gen_uuid();
61
62              $userToken = new UserToken;
63              $userToken->idtoken = $lastIdToken + 1;
64              $userToken->users_iduser = $lastIdUser + 1;
65              $userToken->token = $token;
66              $userToken->save();
67              $userToken->refresh();
68
69          }
70
71      }
72
73      function gen_uuid() {
74          return sprintf( '%04x%04x-%04x-%04x-%04x-%04x%04x%04x',
75              // 32 bits for "time_low"
76              mt_rand( 0, 0xffff ), mt_rand( 0, 0xffff ),
77
78              // 16 bits for "time_mid"
79              mt_rand( 0, 0xffff ),
80
```

```
81              // 16 bits for "time_hi_and_version",
82              // four most significant bits holds version number 4
83              mt_rand( 0, 0x0fff ) | 0x4000,
84
85              // 16 bits, 8 bits for "clk_seq_hi_res",
86              // 8 bits for "clk_seq_low",
87              // two most significant bits holds zero and one for variant DCE1.1
88              mt_rand( 0, 0x3fff ) | 0x8000,
89
90              // 48 bits for "node"
91              mt_rand( 0, 0xffff ), mt_rand( 0, 0xffff ), mt_rand( 0, 0xffff )
92          );
93      }
94
95  }
```

The first part, from line 3 to 11, is the equivalent of the #include in Cpp.

Then in the controller part, we have two functions, the function Create, that is called by the AJAX function, and the function gen_uuid, that will be called by the create function.

So, about the Create function:
The first part of the function is to get the data sent by the AJAX function. There are in the Request object. So we need the 18 to 21 lines to store them into php variables.

☞ Observation 13

In PHP, there is a $ before a variable name.

Then, we assess that the password is alphanumerical, because Laravel just knows how to handle alphanumerical ones.

The next part is to get from the table users the last id.

☞ Observation 14 — About SQL with Laravel

Usually, to get the last id, we would have execute:
SELECT MAX(iduser) FROM users.
But Laravel provides a Query Builder, that is more secure. The difficulty is that instead of executing a request, it is more like a table was an object. So, the request becomes:
Users::orderBy('iduser','desc')->first->iduser
The SQL equivalent is :
SELECT iduser FROM users ORDER BY iduser LIMIT 1.
The three commands give the same result.

Then, we will create a new line into the table users (l. 40 to 46).

☞ Observation 15

As you can see, we hash the password before save it, in order to have an encoded one in the db in case of hacking.

We do the same with the user details and finally we create the token for the user, using the gen_uuid function. This function was provided by the Laravel document in order to fit with Laravel requirements.

Your user is finally created. And we now can log in.

3) Permissions management

a) In the database

If you remember well, two tables are dedicated to the permissions management: permissions and operations. Here they are:

	idoperation	name	description	route
1	0	ADMIN	(null)	/manage_bus
2	1	SEE_ADMIN_PAGES	(null)	(null)

	idpermission	grade	operations_idoperation
1	0	1	0
2	1	1	1

So we have two operations:

- ADMIN: that allows a user to go to the page manage_bus.
- SEE_ADMIN_PAGES: that allows a user to see the admin pages.

As you can see, the permissions are not the same: one is for a page, and the other is more general.

And, in the second table, we can see that both permissions are granted to the grade 1, the admin.

b) In the website

All the permissions are handled in the Permission middleware:
bus_project/app/Http/Middleware/Permission.php

About the first one related to a page:

```
20      public function handle($request, Closure $next, $operationName)
21      {
22
23          if(Auth::check()) {
24
25              $id_user = Auth::user()->iduser;
26              $grade = Auth::user()->grade;
27
28              try {
29                  $operation = Operation::where('name', $operationName)->first();
30                  $operationAttributes = $operation->getAttributes();
31
32                  if(isset($operationAttributes['route'])) {
33
34                      $rankPermission = \App\Models\Permission::where('grade', $grade)
35                          ->where('operations_idoperation', $operation->idoperation)
36                          ->get();
37
38                      if($rankPermission->count() < 1) {
39                          return redirect('home');
40                      }
41                      else
42                          return $next($request);
43
44                  }
45
46              }
47              catch (\Exception $exception) {
```

```
48                  // Opération on trouvée
49                  dd($exception);
50              }
51
52              return $next($request);
53
54          }
55          else return redirect('login');
56
57      }
```

The idea is to get the user id, then, we check his grade, and go pick all the operations is allowed to do, according to the $operationName input. If it's allowed, it return $next($request), but if it is not, it redirect the user to the home page.

But something is missing here... What is this variable $operationName? Where does it come from? It comes from the routes.php file:

```
55 Route::middleware('auth')->group( function () {
56     Route::get('/manage_bus', function () {
57
58         return view('manage_bus');
59     })
60         ->name('manage_bus')
61         ->middleware('permission:ADMIN');
62 });
```

Here we are! Our first route to a page!

As you can see, the first thing is to declare it into a Route::middleware('auth') object. Why? Because it will test if the user is logged or not. If it's not, it will redirect it to the login page without checking any other permission.

Then, there is the route to the manage_bus page. And here, we can find the line 61: *->middleware('permission:ADMIN');*. That is your $operationName variable from the middleware.

Let's talk about the second one:
This permission is to see or not the page in the navigation bar. So to see where does it come from, we need to see the file where the navigation bar is done. It is in:
bus_project/resources/views/includes/aside.blade.php

```
26      <!--begin::Aside Menu-->
27      <div class="aside-menu-wrapper flex-column-fluid" id="kt_aside_menu_wrapper">
28          <!--begin::Menu Container-->
29          <div id="kt_aside_menu" class="aside-menu my-4" data-menu-vertical="1" data-
    menu-scroll="1" data-menu-dropdown-timeout="500">
30              <!--begin::Menu Nav-->
31              <ul class="menu-nav">
32                  <li class="menu-item @if(Route::currentRouteName() == 'home') menu-item
    -active @endif" aria-haspopup="true">
33                      <a href="{{ route('home') }}" class="menu-link">
34                          <i class="menu-icon flaticon-home"></i>
35                          <span class="menu-text">Home</span>
36                      </a>
37                  </li>
38                  <li class="menu-item @if(Route::currentRouteName() == 'bus') menu-item-
    active @endif" aria-haspopup="true">
39                      <a href="{{ route('bus') }}" class="menu-link">
40                          <i class="menu-icon flaticon-bus-stop"></i>
```

```
41                          <span class="menu-text">Our buses</span>
42                      </a>
43                  </li>
44                  @if (\App\Http\Middleware\Permission::hasOperation('SEE_ADMIN_PAGES'))
45                      <li class="menu-section">
46                          <h4 class="menu-text">Admin pages</h4>
47                          <i class="menu-icon ki ki-bold-more-hor icon-md"></i>
48                      </li>
49                      <li class="menu-item @if(Route::currentRouteName() == 'manage_bus')
    menu-item-active @endif" aria-haspopup="true">
50                          <a href="{{ route('manage_bus') }}" class="menu-link">
51                              <i class="menu-icon flaticon-settings"></i>
52                              <span class="menu-text">Bus management</span>
53                          </a>
54                      </li>
55                  @endif
56              </ul>
57              <!--end::Menu Nav-->
```

As it can be seen here, from line 44 to 55, there is an if clause. This clause calls a function from the middleware file, the *hasOperation* function, with the parameter 'SEE_ADMIN_PAGES'.

```
59      public static function hasUserOperation($iduser, $operationName)
60      {
61
62          $userRankID = Users::where("iduser", $iduser)->first()->grade;
63
64          $operation = Operation::where('name', $operationName)->first();
65          if(!$operation)
66              return false;
67
68          $permission = \App\Models\Permission::where('grade', $userRankID)->where('
    operations_idoperation', $operation->idoperation)->first();
69          if(!$permission)
70              return false;
71
72          return true;
73
74      }
75
76      public static function hasOperation($operationName)
77      {
78
79          if(Auth::check()) {
80
81              $user = Auth::user();
82              return self::hasUserOperation($user->iduser, $operationName);
83
84          }
85          else return false;
86
87      }
```

It's done in two functions: the hasOperation function will first check if the user is logged or not. If he is not, it will return false. Else, it will call the function hasUserOperation(iduser, 'SEE_ADMIN_PAGES').

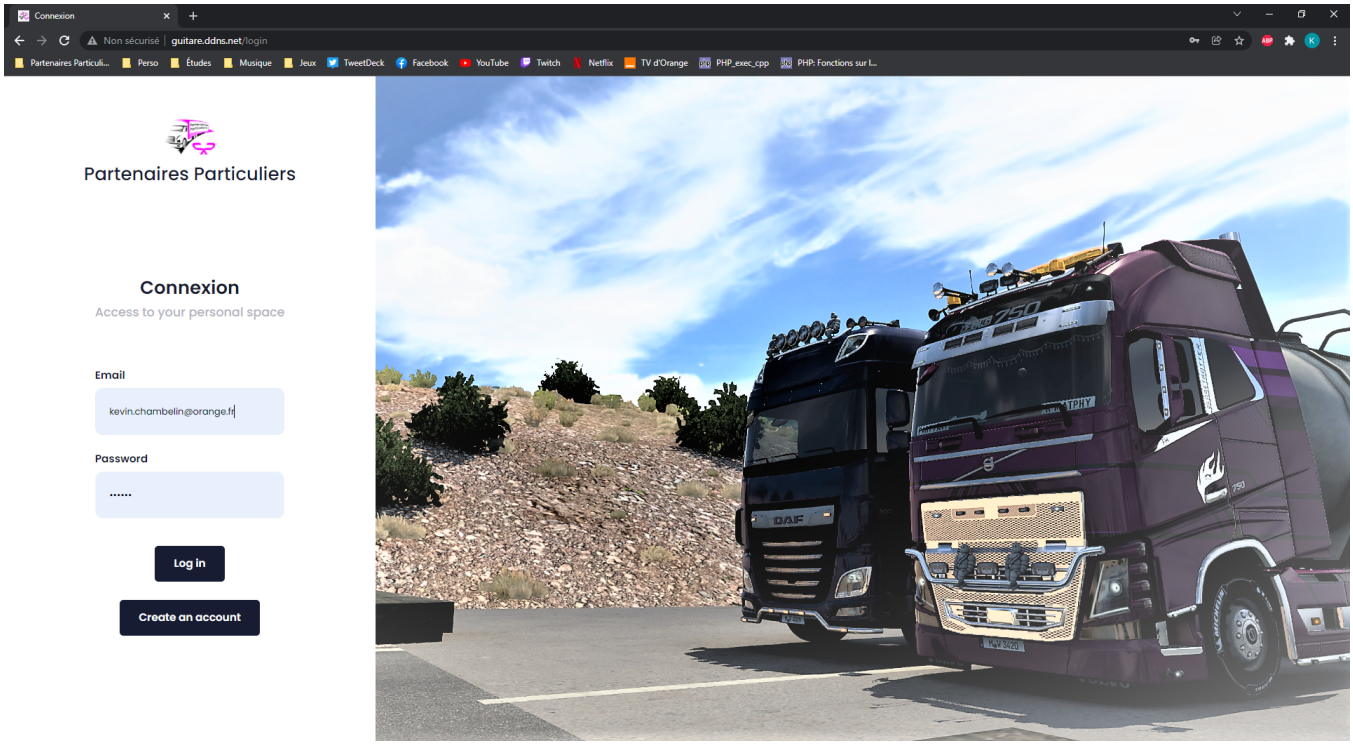
This second function will work like the one about page permission: it will check if the user has the right to perform this operation. If yes, it returns true, else it returns false.

We now have a working permissions handler.

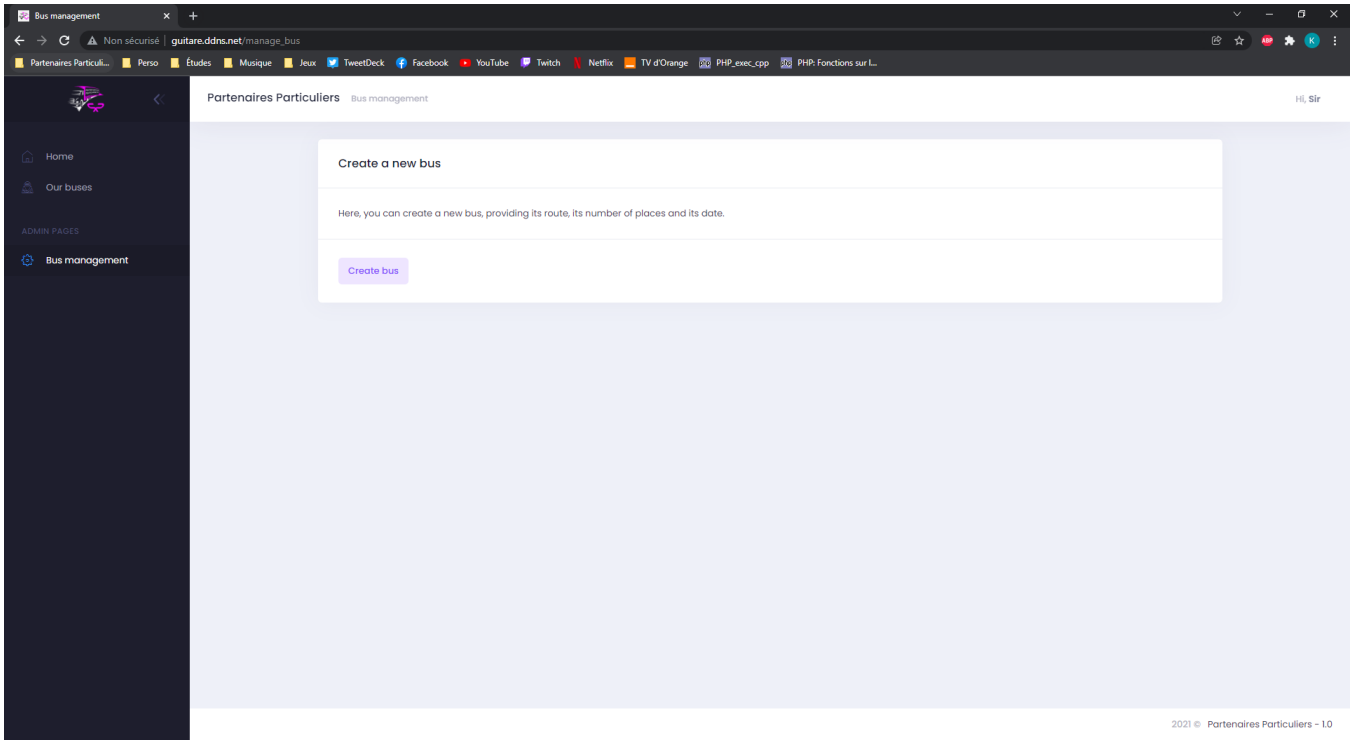
4) Create a bus

Let's go back to the website and create a bus as an admin:

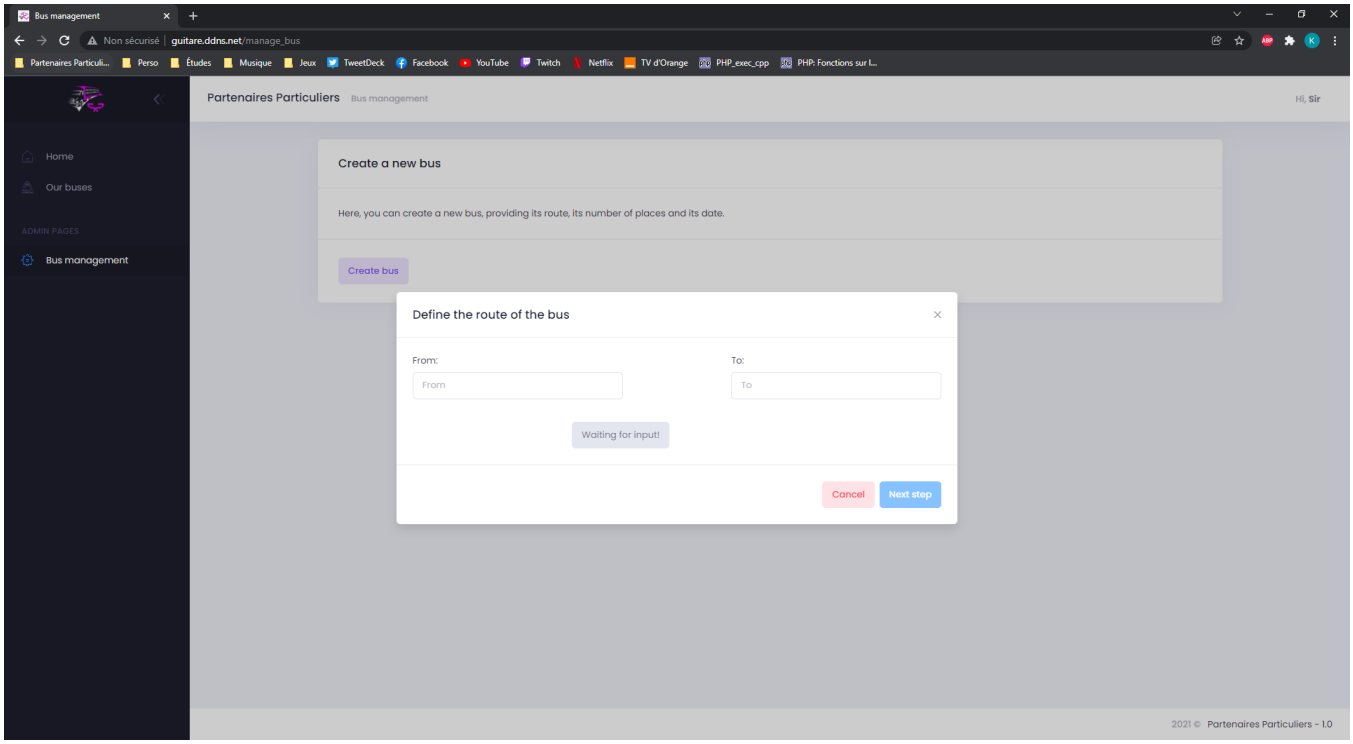
First, let's log with my admin account:



Now, I can see the manage bus page:



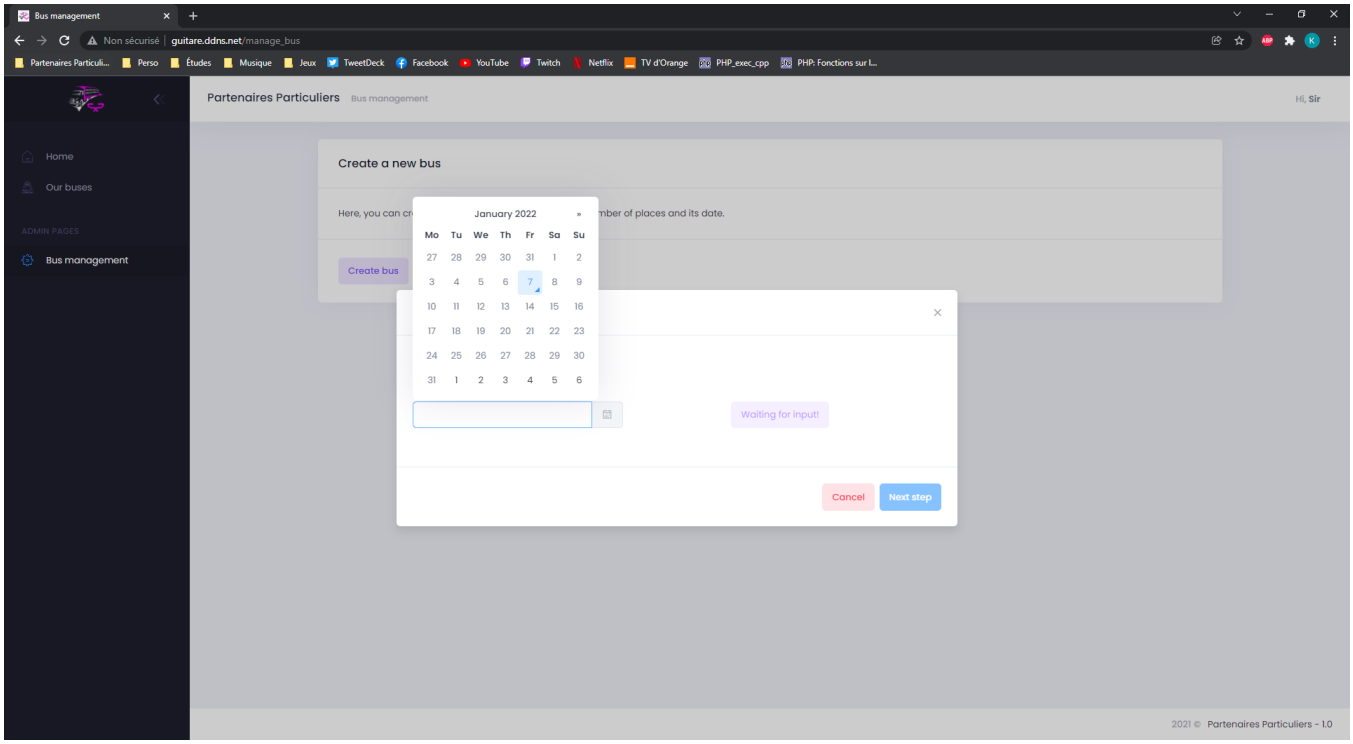
Let's begin the process by clicking on the Create bus button. It will open what we call a modal, that is a little window above the screen:



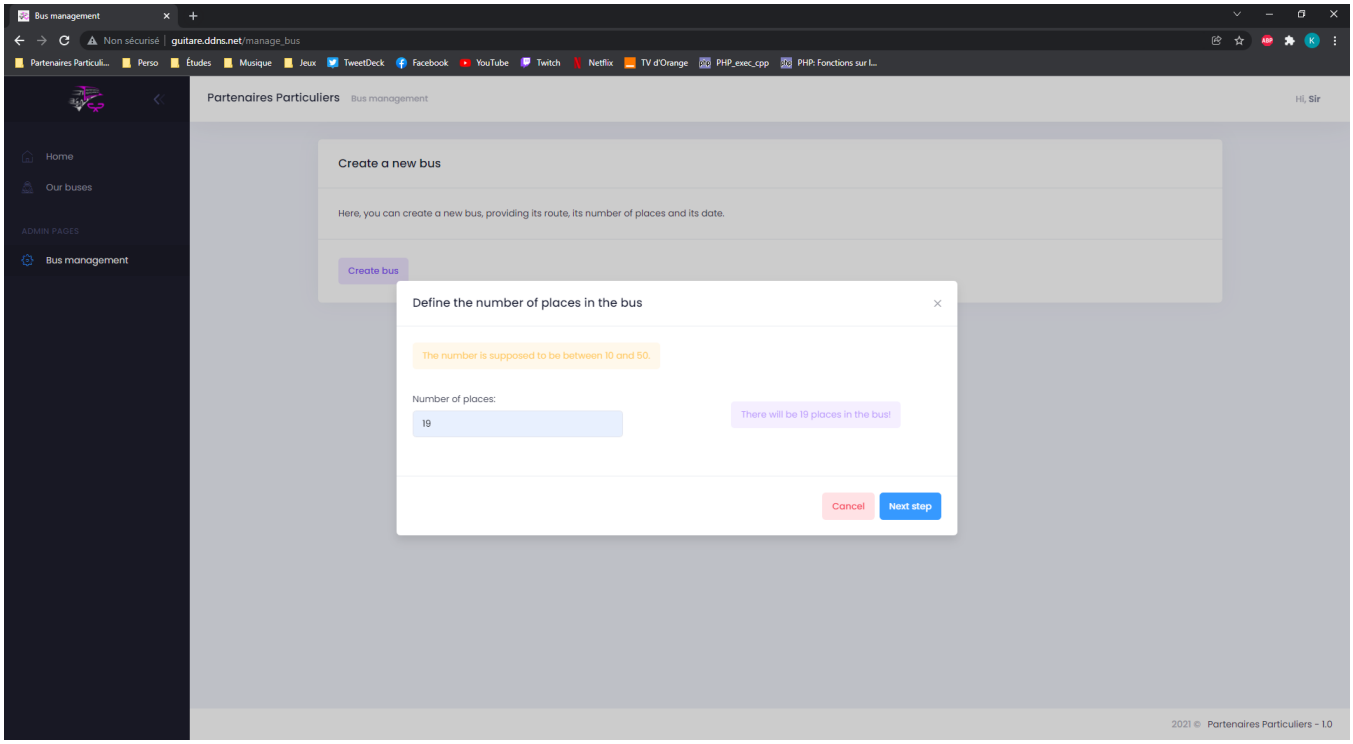
Here we can set the two cities for the bus. Then click on Next step to go to the next modal.

☞ **Observation 16**

The next step button won't available until both inputs are correct.

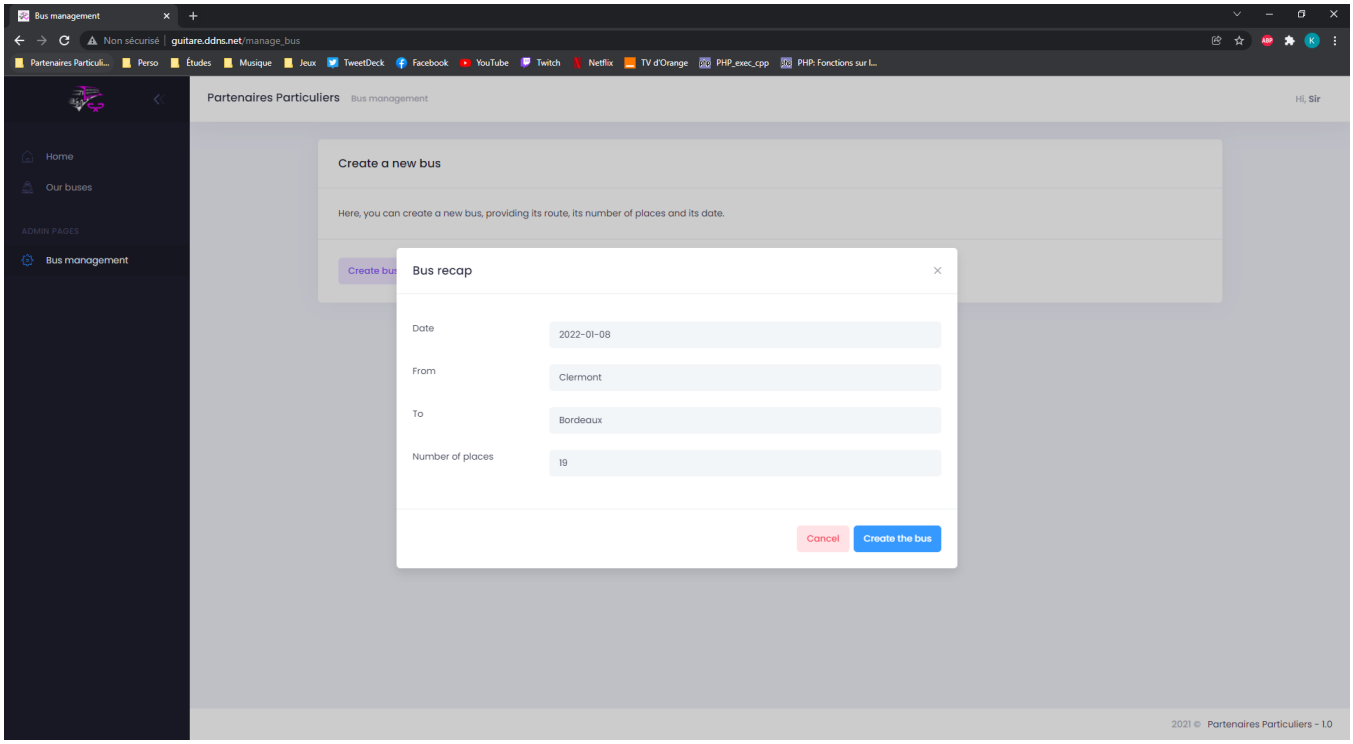


Then we can set the date of the bus. I put in place a protection to prevent the user to set a date before today.



Here, we can set the number of places in the bus.

We just need to check if the data are correct. So there is a recap:



Now, if we click on the Create the bus button, it will trigger a JS function, as it is done for the registration. AS a reminder, JS does not have any access to the db, so we need to post the variables so that php can read and process them.

Here is the JS function, that you can find in:
bus_project/resources/views/manage_bus.blade.php

```
319     function Create_bus() {
320
321
```

```
322
323         $.ajax({
324             url:  "/api/v1/admin/create_bus",
325             type: 'POST',
326             headers:{
327                 'token':token,
328             },
329             data:{
330                 'date' : $('#datepicker').val(),
331                 'from' : $('#from_city').val(),
332                 'to' : $('#to_city').val(),
333                 'nb_places' : $('#nb_places').val(),
334             },
335             success: function (response) {
336                 console.log(response);
337                 window.location.replace("{{route('manage_bus')}}")
338             }
339         })
340     }
```

As you can see, it is exactly the same process as the register one.

Let's see the function in the Bus controller:
bus_project/app/Http/Controllers/BusController.php

```
12     public function Create(Request $request) {
13 
14         $date = $request->input('date');
15         $from = $request->input('from');
16         $to = $request->input('to');
17         $nb_places = $request->input('nb_places');
18 
19         $max_id_bus = Bus::selectRaw('coalesce(max(idbus)) + 1 as maxid')->first()->
    maxid;
20 
21         Bus::insert(
22             [
23                 'idbus' => intval($max_id_bus),
24                 'date' => $date,
25                 'from_city' => $from,
26                 'to_city' => $to,
27                 'place_number' => $nb_places
28             ]
29         );
30 
31     }
```

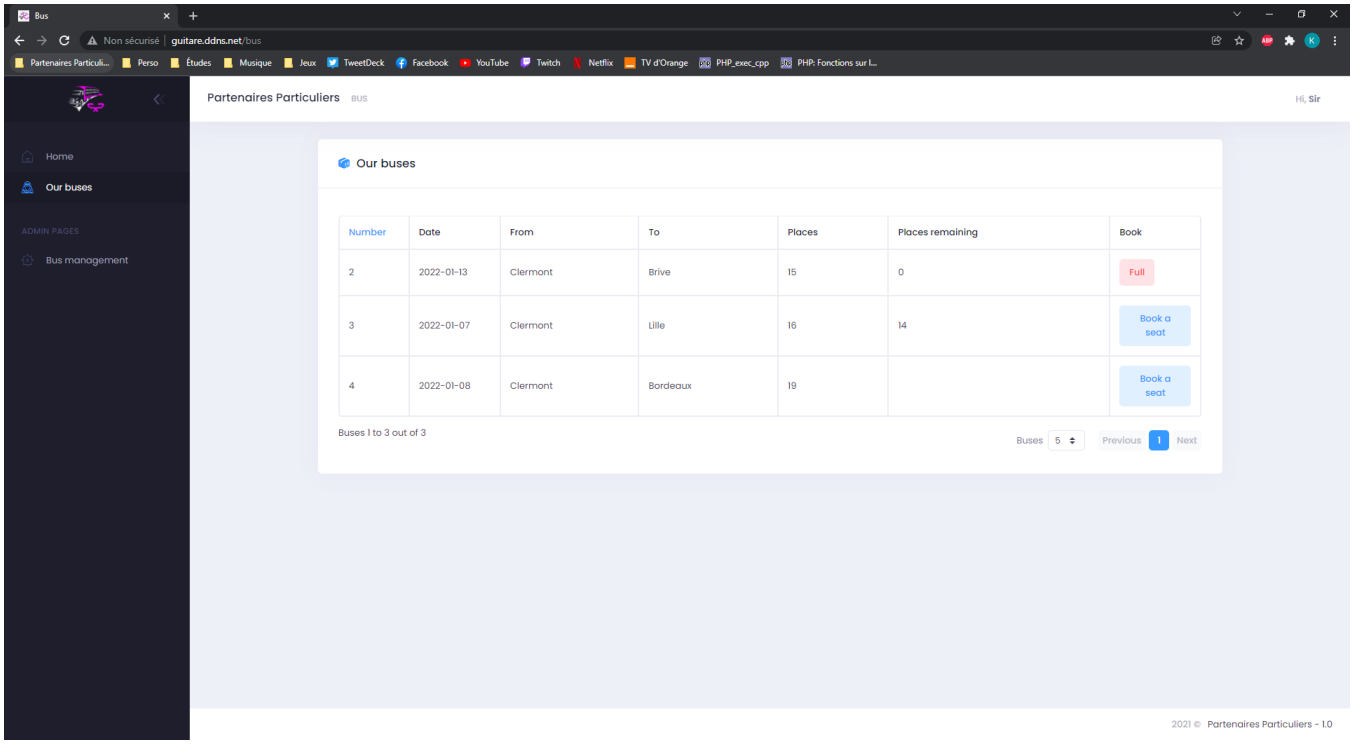
Once again, it's the same idea than for the registration: we save the data from AJAX in php variables, then, we get the max id of buses, and add a new record in the table.

Up to this point, we have achieve all the requirements in order to make our booking system. Let's do it!

5) The booking system – website side

a) Display all the available buses

To reserve a seat, we need to see the bus that are available, as here:



Here, we can all the buses that are scheduled and not yet gone. We can see how many places a bus has and how many places remain in each bus. To know that, we more "complicated" request on the db. In order to do that, I've created a SQL view. It's like a new table that is created as a result of a request. Let's see it:
bus_project/DB/display_buses.sql

```sql
create or replace view display_buses as with purchaised_places as (
    select
        bus_idbus,
        coalesce(
            count(
                idbook
            ), 0
        )as nb_places
    from
        books
    group by
        bus_idbus
) select
    b.*,
    b.place_number - pp.nb_places as places_remaining
from
    bus b
    left join purchaised_places pp on pp.bus_idbus = b.idbus;
```

The request itself begins at "with purchaised..." and ends at the end of the file.

In the with clause we return all the places that are already reserved for each bus. The two items returned are:

- bus_idbus, the id of the bus that will be used to the incoming join.

- coalesce(count(idbook),0) as nb_places, that represents the number of reserved places.

☞ Observation 17

The coalesce keyword is used the next way: *coalesce(smtg, 0)* will return smtg if smtg is not null, 0 otherwise.

Next, we do the select on the join of the table bus (b), and the table purchaised_places (pp) created in the with clause, on the attribute idbus. The selected items are:

- b*: all the attributes of the table bus.
- b.place_number - pp.nb_places: the number of places that remain.

With a view, Laravel will consider this as a table. So, to have the result, we just need to execute: *select * from display_buses.*

On the website, the display of the bus is done by an AJAX request from the JS. This time, AJAX is receiving data from PHP, but it works exactly the same way as before. It uses a library called ktdatatable to display the result:
bus_project/resources/views/bus.blade.php

```
<script>
    "use strict";
    var KTDatatablesSearchOptionsAdvancedSearch = function() {

        $.fn.dataTable.Api.register('column().title()', function() {
            return $(this.header()).text().trim();
        });

        var initTable1 = function() {
            // begin first table
            var table = $('#kt_datatable').DataTable({
                autoWidth: false,
                responsive: true,
                columnDefs: [{
                    className: 'control',
                    orderable: false,
                    target: 0,
                }],
                language: {
                    processing:     "Loading...",
                    lengthMenu:     'Buses _MENU_',
                    info:           "Buses _START_ to _END_ out of _TOTAL_",
                    infoEmpty:      "Buses 0 to 0 out of 0",
                    loadingRecords: "Loading...",
                    emptyTable:     "No bus available",
                    paginate: {
                        first:      "First",
                        previous:   "Previous",
                        next:       "Next",
                        last:       "Last"
                    },
                },

                dom: `<'row'<'col-sm-12'tr>>
                <'row'<'col-sm-12 col-md-5'i><'col-sm-12 col-md-7 dataTables_pager'lp
    >>`,

                lengthMenu: [5, 10, 25, 50],
```

```
140                 order: [[0, 'asc']],
141                 ajax: {
142                     url: "{{url('display_buses')}}",
143                     type: 'GET',
144                     dataSrc: "",
145                 },
146                 columns: [{
147                         data: 'idbus',
148                         width: '100px'
149                     },
150                     {
151                         data: 'date',
152                         width: '130px'
153                     },
154                     {
155                         data: 'from_city',
156                     },
157                     {
158                         data: 'to_city',
159                     },
160                     {
161                         data: 'place_number',
162                     },
163                     {
164                         data: 'places_remaining',
165                     },
166                     {
167                         data: function (row, type, val, meta) {
168                             if (row.places_remaining == 0) {
169                                 return '<button type="button" class="btn btn-light-
    danger font-weight-bold">Full</button>';
170                             } else {
171                                 return '<button type="button" class="btn btn-light-
    primary font-weight-bold" onclick="book(' + row.idbus + ')">Book a seat</button>';
172                             }
173                         },
174                         width: '130px'
175                     }
176                 ],
177
178             });
179
180         };
181
182
183
184         return {
185
186             //main function to initiate the module
187             init: function() {
188                 initTable1();
189             },
190         };
191
192     }();
193
194     jQuery(document).ready(function() {
195     KTDatatablesSearchOptionsAdvancedSearch.init();
196 });
197 </script>
```

The little thing that is different is for the last column, because we need either to have the book button, or to have the full tag. To do so, I made an if statement on the number of remaining places in the bus (l.

167 - 175).

b) Booking function

When we click on the book button, it send to the bus controller, using AJAX, the bus id and the user id. Then, all is handled by the controller: *bus_project/app/Http/Controllers/BusController.php*

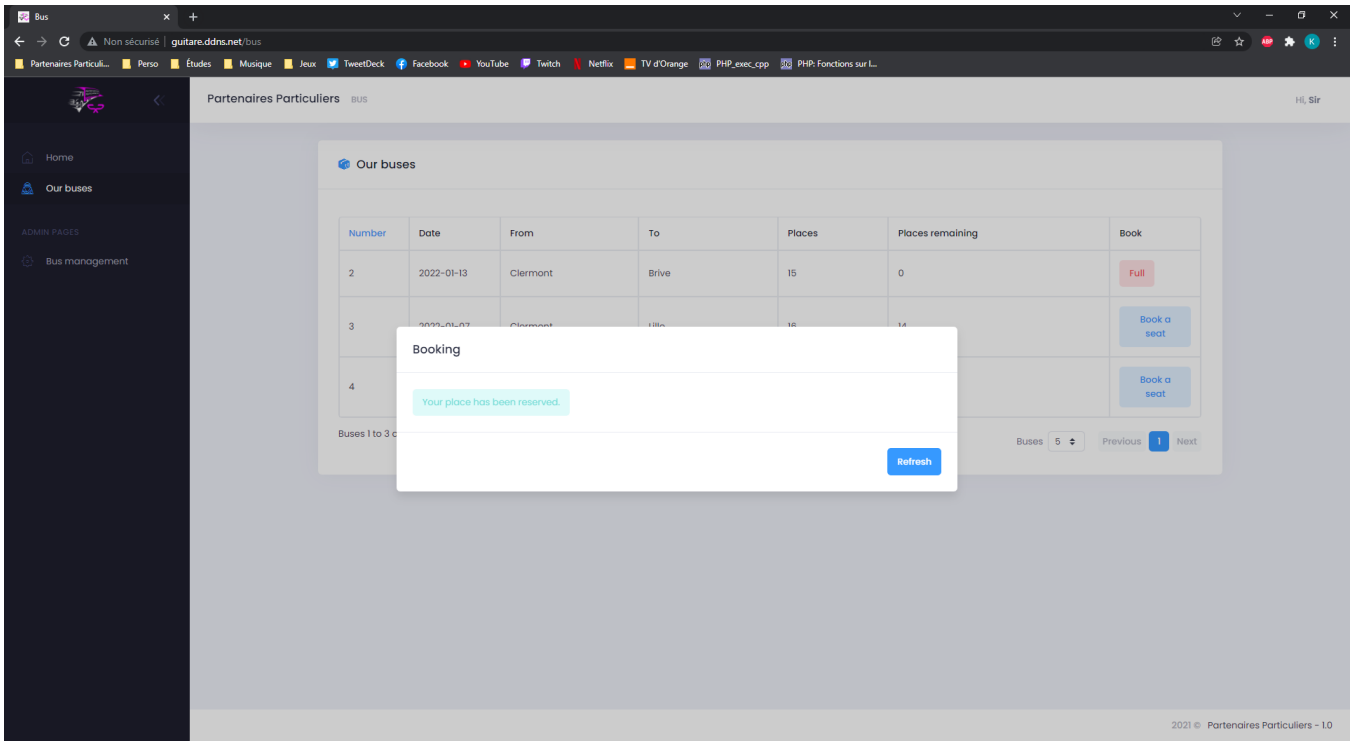
```
40      public function Book(Request $request) {
41
42          $id_bus = $request->input('idbus');
43          $id_user = $request->input('iduser');
44
45          $fp = fsockopen("localhost", 27015, $errno, $errstr);
46          if (!$fp) {
47              return -1;
48          }
49          $is_ok = 0;
50          while ($is_ok != 1) {
51              $is_ok = fread($fp, 10);
52          }
53
54          fclose($fp);
55
56          $purchased_places = Books::selectRaw('count(idbook) as places')
57                                      ->where('bus_idbus', '=', $id_bus)
58                                      ->first()->places;
59
60          $remaining_places = (Bus::select('place_number')
61                                  ->where('idbus', '=', $id_bus)
62                                  ->first()->place_number) - $purchased_places;
63
64          if ($remaining_places < 1) {
65              return -1;
66          }
67
68          $max_id_book = Books::selectRaw('coalesce(max(idbook)) + 1 as maxid')->first()
    ->maxid + 1;
69
70          Books::insert(
71              [
72                  'idbook' => intval($max_id_book),
73                  'bus_idbus' => $id_bus,
74                  'users_iduser' => $id_user
75              ]
76          );
77
78          return (0);
79
80      }
```

Here we are: the heart of the project!

First, we store in php variable what AJAX sent. Then begin the tricky part: calling the cpp server using sockets. The fact is that with php, the use of socket is very simple: it's handled like a file. So, in the $fp variable, we have the socket connection. We check then if the connection worked. If it does not, return -1. Else, it wait for the Cpp server to respond.

When the Cpp server responds, we verified the remaining places, in case somewhere else would have book while we were waiting for the cpp server. If there is still remaining places, it adds the record in the db, and returns 0 to the AJAX. Otherwise it returns -1.

Then, after the execution, the client pops a modal, with the response of the server: if the php response was 0 => success, otherwise => error, try again.



IV) The Cpp server

Here is the most difficult part of the project. The mutex part, you will see, is very simple. But I had to deal with sockets multithreading, and I didn't have any idea of how this thing was working. The fact is that there is a lot of explanations and support for linux users, but no so much for windows ones... So I had to try on my own and I thing I did something not so bad.

To implement it, I used the winsock2 library.

☞ **Observation 18**

In all this part, I will pick up code from the cpp file: *Bus_project/cpp/Server.cpp*

The first thing is to define my server class:

```
23 class server {
24     private:
25         int port;
26         SOCKET ListeningSocket;
27         bool running;
28         SOCKADDR_IN ServerAddr;
29         DWORD ClientThread(SOCKET);
30     public:
31         server(int);
32         int init();
33         int start();
34         int pause();
35         static DWORD WINAPI ThreadLauncher(void *p) {
36             struct thread_param *Obj = reinterpret_cast<struct thread_param*>(p);
37             server *s = Obj->ser;
38             return s->ClientThread(Obj->soc);
```

```
        }
};
```

AS attributes, we have the address, the port, the state of the server and the socket. As functions, I implemented an init, a start and pause functions. I also did a function to create a thread for each connection: ThreadLauncher.

The constructor of the server:

```
server::server(int p) {
    port = p;
    running = false;
}
```

The init() function:

```
int server::init() {
    struct in_addr MyAddress;
    struct hostent * host;
    char HostName[100];
    WSADATA wsaData;

    if(WSAStartup(MAKEWORD(2,2), &wsaData ) != 0 ){
        std::cerr << "WSAStartup failded" << std::endl;
        return 1;
    }

    if( gethostname( HostName, 100 ) == SOCKET_ERROR ){
        std::cerr<< "gethostname() failed: "<< WSAGetLastError() << std::endl;
        return 1;
    }

    if( (host = gethostbyname( HostName ) ) == NULL){
        std::cerr <<"gethostbyname() failed "<< WSAGetLastError()<< std::endl;
        return 1;
    }

    memcpy( &MyAddress.s_addr, host->h_addr_list[0], sizeof( MyAddress.s_addr ) );

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons( port );
    ServerAddr.sin_addr.s_addr = inet_addr( inet_ntoa( MyAddress ) );

    std::cout <<"server initialized" << std::endl;
    return 0;
}
```

The aim of this function is to initialize the serv by setting up the host, the port and all the necessary elements to make the server work fine.

One important aspect was to make it easy to debug, by adding a lot of if statements so that I know what goes wrong.

The start() function:

```
int server::start() {
    SOCKADDR_IN ClientAddr;
    int ClientAddrLen;
    HANDLE hProcessThread;
    SOCKET NewConnection;
    struct thread_param p;

```

```
85      if( ( ListeningSocket = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP ) ) ==
    INVALID_SOCKET ){
86          std::cerr <<"Can't create the socket. Error number " << WSAGetLastError()<< std
    ::endl;
87          WSACleanup();
88          return 1;
89      }
90
91      if( bind( ListeningSocket, (SOCKADDR *)&ServerAddr, sizeof( ServerAddr ) ) ==
    SOCKET_ERROR ){
92          std::cerr<<"bind failed: "<< WSAGetLastError()<< std::endl;
93          std::cerr<<"Port already used "<< std::endl;
94          closesocket( ListeningSocket );
95          WSACleanup();
96          return 1;
97      }
98
99      if( listen( ListeningSocket, 5 ) == SOCKET_ERROR ){
100         std::cerr<<"listen failed "<< WSAGetLastError()<< std::endl;
101         closesocket( ListeningSocket );
102         WSACleanup();
103         return 1;
104     }
105
106     std::cout <<"server started: listening "<<port<< std::endl;
107     running = true;
108     ClientAddrLen = sizeof( ClientAddr );
109
110     while(running){
111         if((NewConnection = accept( ListeningSocket, (SOCKADDR *) &ClientAddr, &
    ClientAddrLen)) == INVALID_SOCKET){
112             std::cerr  <<"accept failed: "<< WSAGetLastError() << std::endl;;
113             closesocket( ListeningSocket );
114             WSACleanup();
115             return 1;
116         }
117
118         p.ser = this;
119         p.soc = NewConnection;
120
121         hProcessThread = CreateThread(NULL, 0,&server::ThreadLauncher, &p,0,NULL);
122         if ( hProcessThread == NULL ){
123             std::cerr <<"CreateThread failed "<<GetLastError()<< std::endl;
124         }
125
126     }
127     return 0;
128 }
```

The aim of this function is to make the server listen to the port, here the 27 015. Then, as soon as it detects a new connection, it launches a new thread bound to this connection.

The mutex function:

```
137 DWORD server::ClientThread(SOCKET soc) {
138     m.lock();
139     int iSendResult, iResult;
140     iSendResult = send(soc, "1", iResult, 0);
141         if (iSendResult == SOCKET_ERROR) {
142             printf("send failed: %d\n", WSAGetLastError());
143             closesocket(soc);
144             WSACleanup();
```

```
145                 return 1;
146             }
147         m.unlock();
148         return 0;
149 }
```

That is the function each thread will execute. So, when a connection is detected, a thread is created and it executes it. The first thing is to lock the mutex. Then, it sends the message to php to proceed to the execution of the book. Then, it unlocks the mutex, so that another thread can lock it...

☞ **Observation 19**

To be honest, I don't think that this solution will work for a huge scale project. However, there are tools available in php, or even sql, to handle real time synchronisation.