REINVENT ENGINEERING

CENTRALE NANTES

Web Programming

Practical Work

JY Martin - JM Normand

August 2025

V 1.0

Table des matières

1 Overview

1.1 Objectives

The main purpose of these practical works is to introduce major frameworks in web programming. Here is the practical work sequence, and the time we recommend to spend :

- Introduction work
  - HTML-CSS basic introduction (2 hours)
  - Using responsive library bootstrap (2 hours)
  - Javascript and AJAX with JQuery (2 hours)
- All in One Web Application (12 hours)
  You will have to implement a SPRING and JPA web application
- Front And Back with Javascript (10 hours)
  You will have to implement one of
  - Angular and NodeJS
  - React and NodeJS

Of course, depending on your knowledge on a technology, you can sometimes go faster. When finished, you can try the technologies you did not tried yet.

Also, previously there was an PHP Synfony app as an All in One Web App. We removed it because using PHP on computers becomes more and more complicated. If you want to see what it lokks like, we put it in the Annexes.

1.2 Practical works

For each practical work, you may have install and check the softwares we will use. These softwares are mentionned at the beginning of each practical work chapter. The way to install and check them can be find in the annexes.

You can do the practical works at your own pace, but your are supposed finishing the set of practical works before the Final Examination.

Questions

For each practical work we add some questions in green boxes. Be sure you know the answer before you go on.
If you have a problem with a question, you should have a look to previous informations to find the answer.

There are no report to write for practical works.

For the practical work you have to implement, you may find most of possible solutions on https ://hippocampus.ec-nantes.fr. However, you are not supposed to copy all files of the solution. The solutions show you how you could implements the problems. You have to learn to implement this by yourself.

TAKE CARE IF YOU COPY TEXT FROM THE PDF : very often there are invisible chars or spaces included in PDF text to ensure presentation. When you copy and paste text, that includes these chars which may generate compilation problems

1.3 Final Examination

Final examination will take place the last 2 hours of this course.

You are supposed having ready your computer with the set of tools we use for all practical works. You should already have install and test them.

Test will start with a 15mn MCQ. There will be 3 subjects (the 3 frameworks) and you will have to fill 2 of them.

Next, you will have to write a web application, using SPRING REACT or ANGULAR. Your the practical work with the framework you choose must be fully operationnal.

2 Practical work problem

Our objective is to create an application that manages books borrowing. We manage users, books, and borrowers.

We have a database in PostgreSQL that stores informations. The schema is given in [Fig. 1]. You will find in the materials the database SQL commands to create the database and add some data.

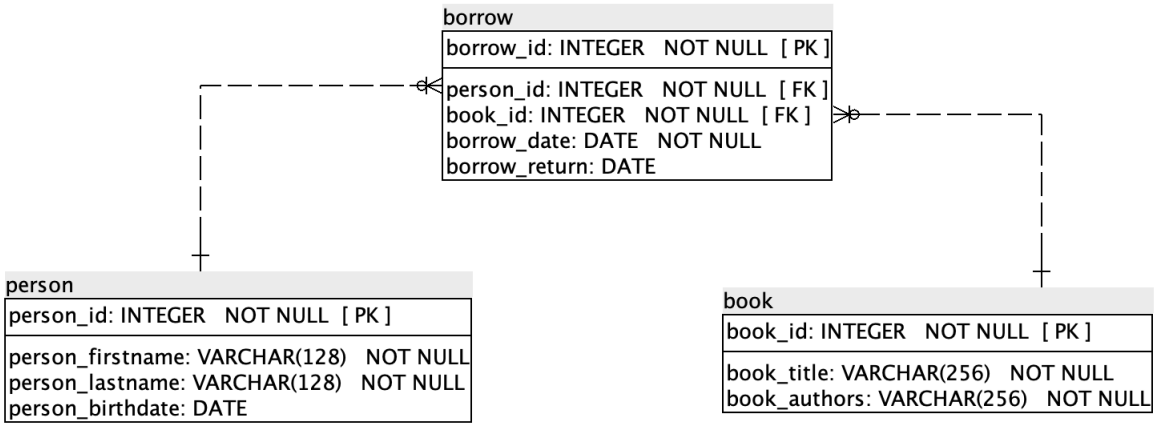


Fig. 1 : The database schema

Our objective is to create an application that interacts with this database. We will implement the same applications using 3 different ways of doing that.

We need :

- an authentication page
- a page to manage users
- a page to manage books
- a page to create / modify a single user
- a page to create / modify a single book.
- a page to link users and books when users borrow a book, or returns it.

During the HTML-CSS practical work, you will implement the HTML structure of these pages. Then you will learn to use Javascript and AJAX, that you will use during all the practical works. Then we will implement the application with Spring, Angular or React.

3 Introduction work

3.1 HTML-CSS

In this practical work we will build some HTML pages, using CSS files to present informations.

We will not deal with libraries. We will neither deal with responsive elements. However, if you want to use them, you can use libraries like bootstrap or materialize. Also, if you want to use a library to manipulate lists, you can use something like Datatable.

3.1.1 Required tools

For this part, you will need :

- a web server to serve your pages. You can use for example :
  - apache web server on linux and macos (already installed in these environments)
  - WampServer, xampp or any web environment of this kind (Windows users)
- a text editor to create pages
- a browser (web navigator) like chrome, firefox, safari, opera, ... to visualize the results.

3.1.2 Checking your files

In this chapter, we will suggest some files to implement. Feel free to change placement, colors, ...
For each page you implement, there are some points you have to check :

- Open the page in the browser. Check the result is ok.
- Copy page, images, ... to the root location of your web server.
  - Windows : depend on the application you chose. Maybe htdocs ?
  - Linux : usually /var/web/html
  - MacOS : /Library/WebServer/Documents
- Open the page through its URL (http ://localhost/mypage.html) and check you have the same result.

3.1.3 Validating pages

Most of the browsers can interpret pages even if they are not correctly written. Sometimes they can't because it is really unreadable.

However, HTML has a valid syntax, and miswriting HTML pages expose you to misinterpreting pages.

To check your page validity you can use the W3C validator : http ://validator.w3.org/

- Connect to the HTML validator [Fig. 2]
- Select your HTML file
- Check result [Fig. 3]

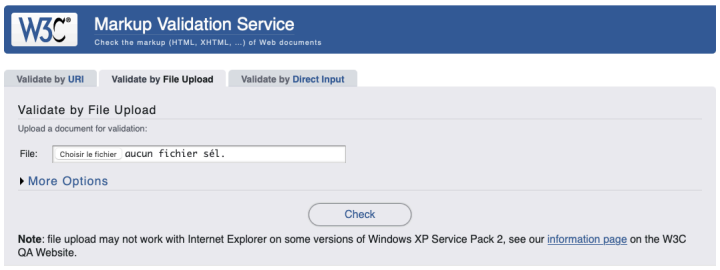


Fig. 2 : The HTML Validator

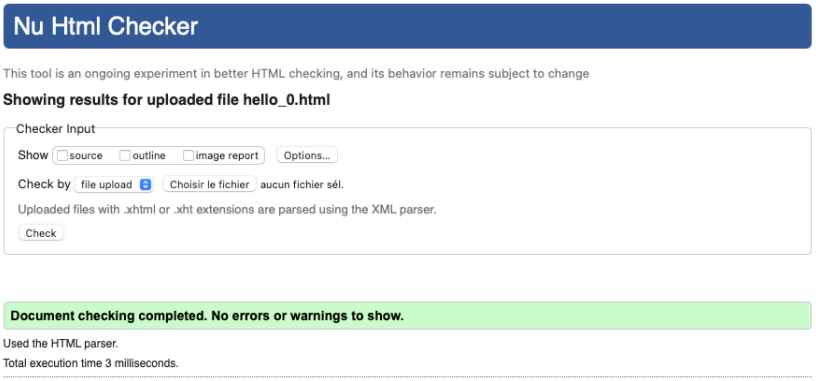


Fig. 3 : The HTML Validator result on an HTML file

3.1.4 Hello world Page : basic version

We will start with the traditional "Hello World".
First, the structure. Our hello.html page will contain :

- a html tag
- a head tag for the page elements (informations, included css, js, ...)
- a body tag for the page content (what is displayed)

The HTML source code is given in the left part of [Fig. 4]. Check it is really UTF-8 encoded. The result is in the right part of the figure.

File : hello.html

```
<!DOCTYPE html>
<html lang="fr-fr">
  <head>
    <title>Hello World</title>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <p>Hello World</p>
  </body>
</html>
```

Result :

Fig. 4 : HTML source code and result for hello.html

Open your page in your web browser and check you have the right result.

Now, place your file hello.html in the root directory of your web server. Ensure the web server is running. Once done, use your web browser with its URL to access to your page. The URL should be http ://localhost/hello.html

You should obtain the same result with the file protocol and the http protocol.
The difference between the 2 methods is the way you access the file.

- When you open the file, that means you use the "file" protocol, like in [Fig. 5]. You take the file on your disk and you directly open it with the browser. You do not use any web server.

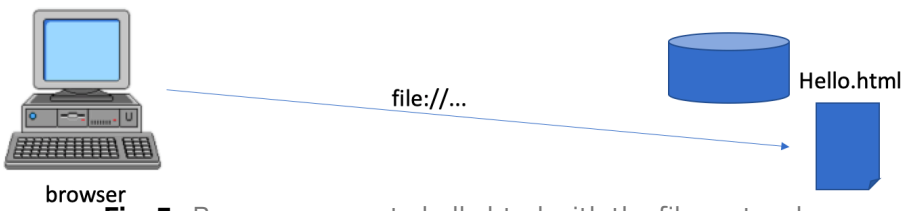


Fig. 5 : Browser access to hello.html with the file protocol

- When you use the URL in your browser, that means you use the “http” protocol, like in [Fig. 6]. The browser launches a request to the web server (even if currently this is the one on your computer), this server loads the requested file and sends it back to the browser.

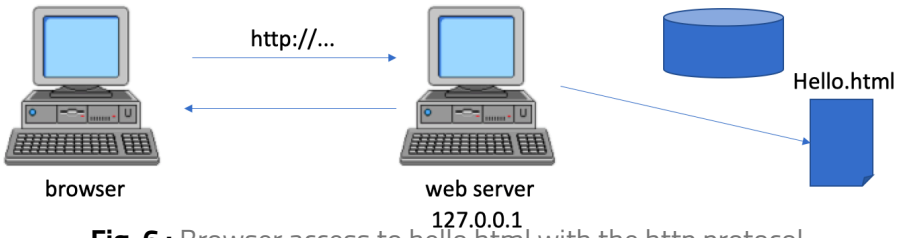


Fig. 6 : Browser access to hello.html with the http protocol

3.1.5 Hello world Page : with styled tags

There are several ways to add styles for tags. Here are the most used :

- in the tags of the HTML file
  (should be avoid, if possible)

```
<p style=”color :red”>Hello World</p>
```

- with default style tag definition in a CSS file

```
<p>Hello World</p>
```

CSS File :

```
p {
    color :red;
}
```

- with a class definition in a css file

```
<p class=”error”>Hello World</p>
```

CSS File :

```
.error {
    color :red;
}
```

- with an id definition in a css file

```
<p id=”error”>Hello World</p>
```

CSS File :

```
#error {
    color :red;
}
```

If you use a CSS file, you will have to include it in the HTML file, in the **HEAD** section.
This example includes file myCSSFile.css.

```
<head>
...
<link href=”myCSSFile.css” type=”text/css” rel=”stylesheet” />
...
</head>
```

The href attribute is given according to the HTML file location. If it is not beside the HTML file, you have to give its relative location. For example “css/myCSSFile.css” means there is a “css” directory beside the HTML file, and a file “myCSSFile.css” inside. If you use href “/css/myCSSFile.css”, that means there is a “css” directory at the root of your web server, and a file “myCSSFile.css” inside.

Change the “Hello world” using the 4 different methods. You have an example in [Fig. 7].

HTML file

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link href="hello.css" type="text/css" rel="stylesheet" />
    </head>
    <body>
        <p class="error">Hello World</p>
    </body>
</html>
```

CSS file “hello.css”

```
body {
    background-color :white;
}

p.error {
    margin : 5px 0;
    color : red;
}
```

Result in browser

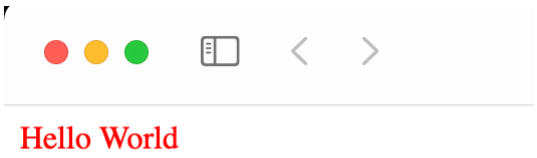


Fig. 7 : Colored Hello World example. HTML source code, CSS Source code and result

3.1.6 Tables

Now, let's use a **table** to present some kind of data.
Tables are created row by row using tag **tr**. In each row, you define the columns with **th** or **td**. th is for a header cell, td for a standart cell.

You may also merge cells on several columns (with **colspan**) and on several rows (using **rowspan**). In that case, take care of cells definition because some cells could already have been defined.

[Fig. 8] is an example of a table creation.

table.html

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link href="css/myCSSFile.css"
        type="text/css" rel="stylesheet" />
    </head>
    <body>
        <h1>This is my table</h1>
        <table>
            <tr>
                <th>Team #</th>
                <th>Players Name</th>
                <th>Score</th>
            </tr>
            <tr>
                <th>1</th>
                <td>Ayato-Julis</td>
                <td>97</td>
            </tr>
            <tr>
                <th>2</th>
                <td>Saya-Kirin</td>
                <td>85</td>
            </tr>
            <tr>
                <th colspan="2">Total</th>
                <td>182</td>
            </tr>
        </table>
    </body>
</html>
```

css / myCSSFile.css

```
body {
    background-color :white;
}

table {
    border : 1px solid green;
    color :forestgreen;
}

tr {
    background-color :white;
    color :forestgreen;
}

td {
    border :1px solid green;
    color :forestgreen;
}
th {
    background-color :yellowgreen;
    color :white;
}
h1 {
    margin : 0 0 10px;
    font-size : 20px;
    font-weight : bold;
    color :green;
}
```

Result :

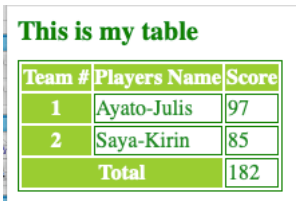


Fig. 8 : Table HTML code, CSS file and result

In the last line of the table definition, note that as we use colspan.
Colspan defines 2 cells and merge them, so we have to define the last third cell.

3.1.7 Linking pages

OK, we wrote separate pages but, usually, in a site, pages are linked together.
Let's try to add a link between 2 pages.

First, we add a link in "hello.html" so that it calls the table page. We use the tag **a** to link a page to another page. **href** sets the URL for the link.
Have a look to [Fig. 9] for the link definition. The right side of the figure shows what happens.

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link href="css/hello.css"
            type="text/css" rel="stylesheet" />
    </head>
    <body>
        <p><a href="table.html">Hello</a> World</p>
    </body>
</html>
```

```
body {
    background-color :white;
}
p {
    margin : 5px 0;
    color : red;
}
a, a :focus {
    color : blue !important;
    text-decoration : none;
}
a :hover {
    color : purple !important;
}
```

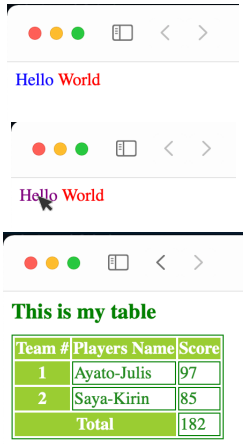


Fig. 9 : Define a link between 2 pages

The a tag links to table.html
The text displayed (on which you can click) is Hello (in black in the example).

3.1.8 Using forms

To interact with users we must use tools where we can select informations, give data and validate a set of informations.

For that, we will use different tags :

- input type
  - text (a line of text)
    <input type="text" name="..." value="..." placeholder="..." />
  - radio (a radio button to select an element among several ones). Radio buttons with the same name are linked together as a radio group.
    <input type="radio" name="..." value="..." />
  - checkbox (select an element or not)
    <input type="checkbox" name="..." value="..." />
  - password (a line with hidden chars)
    <input type="password" name="..." value="..." />
  - submit (a button to submit the form)
    <input type="submit" name="..." value="..." />
- select (a list of elements, lines are defined with tag option)
  <select name="..." ><option value="..." >... </option ><option value="..." >... </option >... </select >
- button (a button to submit the form, may context text, images, ...)
  <button name="..." >whatever you want </button >
- textarea (a text zone of several lines and cols)
  <textarea name="..." >the text by default </textarea >
- ...

A **form** is a set of interactive elements. Each element should have an attribute **name** because when sent, each name is associated with its **value**.

When a form is submitted, the browser sends a request to the server with the elements (not disabled) in the form. Each element is sent as **name=value**, even the button you clicked on to submit the form. Check bowes are sent nly if they are selected.

The form attributes define the action (URL) to execute on the server and the call method (GET / POST). You may also change this in the tag **button** with attributes formaction and formmethod.

NB : it is **stricly forbiden** to include a form in another form.

[Fig. 10] is an example of form tag. Try it.

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link href="css/myCSSFile.css" type="text/css" rel="stylesheet" />
    </head>
    <body>
        <h1>This is a form</h1>
        <form action="http ://dr-ser-info.ec-nantes.fr/prweb/test.php"
                    method="GET">
            <p>Give your login<br/>
            <input type="text" name="myLogin"
                    value="" placeholder="your name" /></p>
            <p>Give your password<br/>
            <input type="password" name="myPasswd"
                    value="" placeholder="your password" /></p>
            <p>Select your speciality<br/>
            <select name="mySpeciality">
                <option value="-1" selected="selected">-</option>
                <option value="1">INFO</option>
                <option value="2">RV</option>
            </select>
            </p>
            <p>Select your status<br/>
            <input type="radio" name="myStatus" id="status1" value="1" />
                <label for="status1">Teacher</label><br/>
            <input type="radio" name="myStatus" id="status2" value="2" />
                <label for="status2">Student</label>
            </p>
            <p>
                <input type="checkbox" name="myChoice" id="choicebtn" value="1" />
                <label for="choicebtn">Check the button</label>
            </p>
            <p><button>Valid form</button>
        </form>
    </body>
</html>
```

```
body {
    background-color :white;
}
h1 {
    margin : 0 0 10px;
    font-size : 20px;
    font-weight : bold;
    color :green;
}
input {
    display : inline;
    width : inherit;
    padding : 2px;
    background : rgba(0,0,0,.05);
    border : 1px solid rgba(0,0,0,.15);
    box-shadow : 0 1px 2px rgba(0,0,0,.1) inset;
    border-radius : 3px;
    font-size : 14px;
    & :invalid :not(:focus) {
        border-color : red;
    }
}
```

Fig. 10 : Form HTML code, and CSS file

In our example, the **form** tag has 2 attributes : the **action** attribute that tells which URL is called when the form is submitted (http ://dr-ser-info.ec-nantes.fr/prweb/test.php), and **method** that tells which method is used (GET, POST, ...).

We use a **label** tag connected to our **radio** tag and to our **checkbox** tag. If you click on the label, it as the same effect as when you click on the connected element. Connection between the input tag and the label tag is done through the **id**.

You can use any protocol (file or http) to call your file. It should work because our url is a full url to a server. If the URL was relative to your server, your file MUST have been on your server ans you MUST use protocol http to call it.

Open the html file in your browser, with its URL (http ://localhost/myFile.html).
Give informations and validate the form.

You should have a result as a JSON response (this is what does this script).

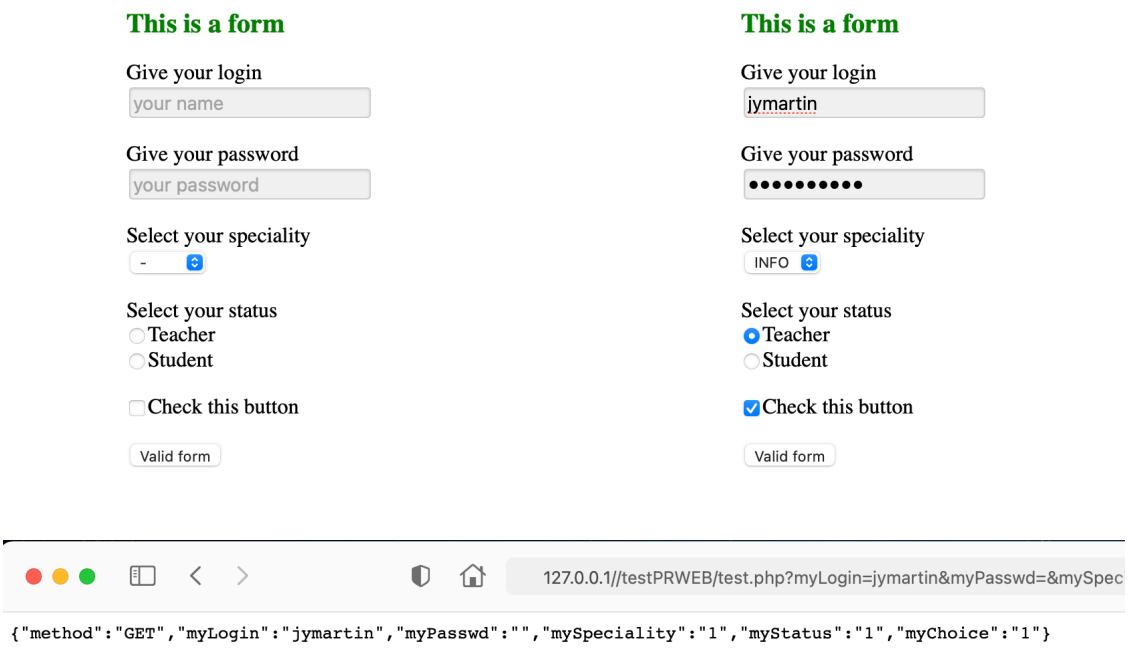


Fig. 11 : Form usage, and the result of the PHP script

Have a look to the URL used bottom of the [Fig. 11], just beside the small house. You should have the URL but also elements just after the "?" : the elements sent to the server. Check yours.

Now, replace the "GET" method in the form tag by "POST". Reload your page, fill the form and submit the form. What is the difference in the URL ? In the result ?

Have a look to the result. Compare the form input **name**s to the ones displayed.

Questions

What is the link tag used for ?
in the URL bar, why do we use http ://.... or file ://... or https ://...
What is the difference between file and http protocols ?
How can you send data to a server ?
Why should you prefer using CSS files instead of inserting CSS commands in HTML files ?
How do you give a name to an interactive element (input tag for example) in a form ?
What is the difference between a textarea and an input type text element ?
What is the difference between GET and POST methods ?
How do you connect a radio button or a checkbox button to its associated text ?

3.1.9 Our Software HTML pages

Now let's define some pages we will use in our software. Of course, you can customize pages as you want.

3.1.9.1 Login Page

Now, we will build pages for the problem we want to solve.

First, the idea is to create an authentication page. We call this file **index.html**, so that it will be called by default.

You have to create a form. In this form, you have to create 2 input elements, and something to validate the form :

- first element for login,
- second one for password.
- Add a button to validate the form.

Your form will call action "login" using method "POST". Of course, this action does not exit yet. If you want to check it works, you can replace action "login" by the "test.php" on ser-info-03.

As an example, you can use [Fig. 12]. We used a texfield tag to build the rounded rectangle.

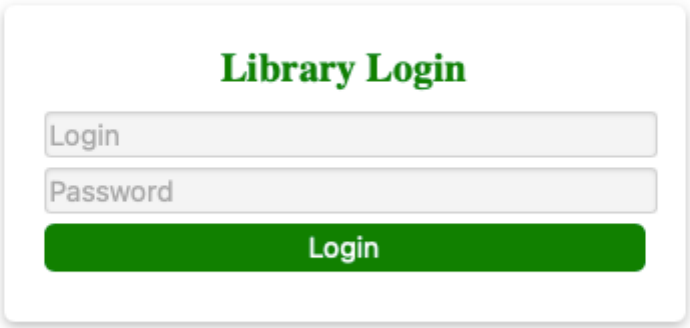


Fig. 12 : Example of Login page

3.1.9.2 List Users page

The idea is to create a page that lists users.

Currently, in this page the users will be fake ones because the page is not connected to the database. We call this file **users.html**

You have to create a table. Each line contains a user. We also need some buttons to launch actions like adding a user, modifying a user, delete him-her. For these buttons, we will use icons in buttons.

For each user (for each row), in the last column, create a form that calls action "editUser" in method "POST". In the form, add a button that could link to an "Edit user" page, and another one to a "Delete user" action. Do not connect the buttons yet. You only have to display them.

Tu use an icon in a button you can :

- create a directory "img" (or whatever you want) to store your images
- put icons (png images) in your image directory
- create buttons in your form. The button contains an a tag img that includes an image

```
<button>
    <img src="img/edit.png" alt="edit" class="icon" />
</button>
```

```
body {
    background-color :white;
}
img.icon {
    display : inline;
    height : 20px; // icons are 20px height
}
```

As an example, you can use [Fig. 13].

Fig. 13 : Example of List users page

3.1.9.3 List Books page

This is exactly the same king of page than the “List Users” page. Columns are not the same. We call this file **books.html**

As an example, you can use [Fig. 14].

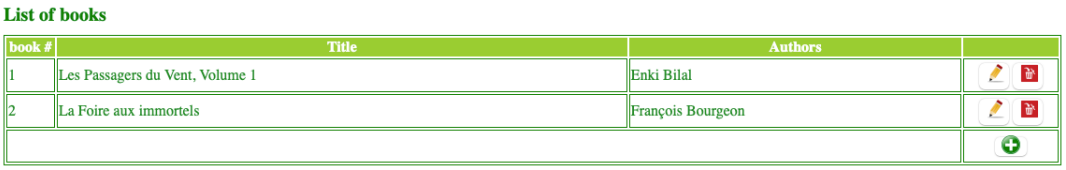


Fig. 14 : Example of List books page

3.1.9.4 Edit / Create User page

The idea is to create a page that edit user’s informations.
This page will also be used to create a new user. We call this file **user.html**

You have to create a table. Each line contains an user information.

- First colomn is the information title
- Second column is an input element

As an example, you can use [Fig. 15].

Fig. 15 : Example of Create user page

3.1.9.5 Edit / Create Book page

The idea is to create a page that edit book's informations.
This page will also be used to create a new book. We call this file **book.html**

You have to create a table. Each line contains an user information.

- First colomn is the information title
- Second column is an input element

As an example, you can use [Fig. 16].

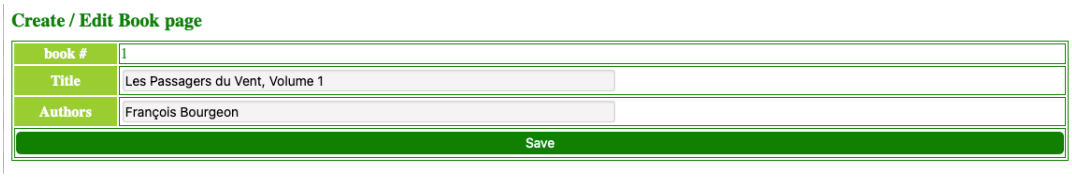


Fig. 16 : Example of Create book page

3.1.9.6 Edit / Create Borrowing page

An user usually borrows books.
This page will manage the books a user borrows. We call this file **borrow.html**

We have to display the user's firstname and lastname.

Then, we display the list of borrowed books. We display an icon if the book is not returned, the return date otherwise. And add last, we must add something so that the user can borrow a book.

As an example, you can use [Fig. 17].

Fig. 17 : Example of borrowing list page

3.2 Using responsive library

Ok, let's try to switch to a responsive website. We will use **bootstrap** for that. When we write these lines current version is 5.3. You can find bootstrap documentation here :
https ://getbootstrap.com/docs/5.3/getting-started/introduction/.

3.2.1 Required tools

First, we need **JQuery** : https ://jquery.com. When we write these lines, version is 3.7.1.
Version 4.0 is announced.

- You can use the http link to the file and put it in the head section (this is called a CDN link : Content Delivery Network). The file remains on its original server and you only link it.
  <script
  src="https ://code.jquery.com/jquery-3.7.1.js"
  integrity="sha256-eKhayi8LEQwp4NKxN+CfCh+3qOVUtJn3QNZ0TciWLP4="
  crossorigin="anonymous"/ ></script/>
- You can download the file (at https ://jquery.com/download/) and place it in a specific directory and refer to it. <script src="theLinkTo jquery-3.7.1.js" ></script/>

Then you might add bootstrap. You can link it or download it, the same way as JQuery.
There are 2 links : 1 for CSS and 1 for JS.

- You can use the http link to the file. Have a look to https ://getbootstrap.com and use CDN section.
- You can download the file and place it in a specific directory and refer to it in you hrefs. Current link is https ://getbootstrap.com/docs/5.3/getting-started/download/.

Both works and you can use the one you want. In our sources we use files located on our server to be able to develop without any internet connection.

Also, in the materials, you will find bootstrap and JQuery libraries we downloaded to build our solutions.

3.2.2 Some bootstrap principles

Bootstrap uses **JQuery**, **CSS** elements and **JS** scripts to manage visual aspect of pages. According to page size, the way elements are displayed can be different.

Bootstrap uses **div** bocks to manage visual aspect, positionning, ... With the div tags, Boostrap uses CSS classes that defines grids, rows, ...

How do it works ?
First we have to include the libraries in the head section. Have a look to [Fig. 18].

```
<!DOCTYPE html>
<html lang="fr-fr">
  <head>
    <title>Library Login</title>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <script type="text/javascript" src="js/jquery-3.7.1.min.js"></script>
    <link rel="stylesheet" href="bootstrap/css/bootstrap.css">
    <script src="bootstrap/js/bootstrap.min.js"></script>
  </head>
  <body>
```

Fig. 18 : Include Bootstrap files and viewport definition

Next we have to define blocks position, define a grid to place elements, ...

For block placement in the page, you should have a look to
https ://getbootstrap.com/docs/5.3/utilities/spacing/.

In the placed block, we define a grid. https ://getbootstrap.com/docs/5.3/layout/grid/ explains how it works.

First, we have to place a grid (a bloc). It is defined through a tag “div” with class “container”. A grid defines 12 columns, and the element placements (with div) uses these 12 columns.

To place elements we have to define rows (a div witch class row), and for each row we have to define blocks with their correct width (in terms of columns). Remember to count how large are your elements in the row (sum might be less or equal to 12, and if possible 12).

3.2.3 Add Bootstrap to our files

Let's try to add Bootstrap in our application file.
First remember you have to include Bootstrap elements in your files.

3.2.3.1 Bootstrap in index.html

Let's start with our login page. Feel free to change the elements according to what you want to create.

We need a main block to place the elements. We use a div with class py-5 to add padding to our elements, And a container to define the grid.

First row is the for the title.
Next row is for the elements to get from user (login and password).

For each row, we use a bloc with the full available with (12 columns).

We need a form to ask for the login (an input text) and password (an input password), and a button to validate. Documentation about forms can be found in https ://getbootstrap.com/docs/5.3/forms/overview/

For our 2 lines, we use 2 columns for the label and 10 columns for the input elements. Depending on the form you want to build, you can remove the label and use 12 columns (or less) for your input element. For each row, we tell it is a "form-group". Each input element has a class "form-control" to be managed.

We need a button to validate the form. Buttons have the class "btn" to be displayed as buttons. Their color is managed with another class that use predefined colors according to the role the have. Have a look to https ://getbootstrap.com/docs/5.3/components/buttons/ for more informations.

[Fig. 19]shows what we would like to build.

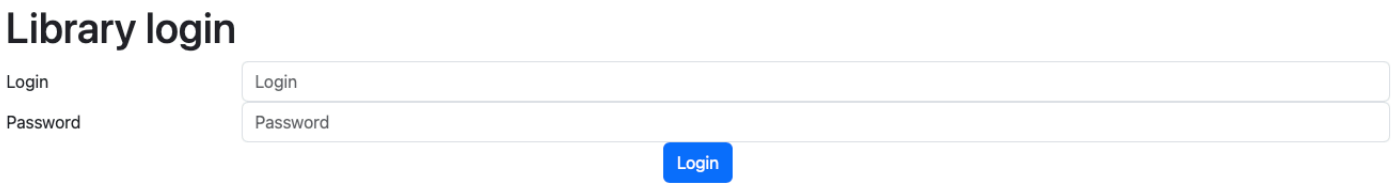


Fig. 19 : Login page we want

[Fig. 20] gives you a way to implement the Login form using Bootstrap.

```
<body>
  <div class="py-5">
    <div class="container">

      <div class="row">
        <div class="col-md-12">
          <h1 class="">Library login</h1>
        </div>
      </div>

      <div class="row">
        <div class="col-md-12">
          <form action="login" method="POST">
            <div class="form-group row">
              <label for="inputLogin" class="col-2 col-form-label">Login</label>
              <div class="col-10">
                <input type="text" class="form-control" id="inputLogin"  name="login" placeholder="Login" required="required" />
              </div>
            </div>
            <div class="form-group row">
              <label for="inputPassword" class="col-2 col-form-label">Password</label>
              <div class="col-10">
                <input type="password" class="form-control" id="inputPassword"  name="password" placeholder="Password" required="required" />
              </div>
            </div>
            <div class="form-group row">
              <div class="col-12 text-center">
                <button type="submit" class="btn btn-primary">Login</button>
              </div>
            </div>
          </form>
        </div>
      </div>
    </div>
  </div>
</body>
```

Fig. 20 : Use Bootstrap for the form

Try the page in the web browser. Also, try to resize your page to see if it is responsive.

3.2.3.2 Bootstrap in users.html and user.html

We would like :

- A title for the page
- A list of users
- Something to add a new user.

So, what do we need ?

- A block that holds the elements. and a grid to place them.
- First row of the grid is for our title. 12 columns width.
- Another row for the table. We want the table to be responsive.
- Table rows for data (user).
- Forms to manage actions on each row
- A form to add a new user.
- Buttons

As many elements, tables can be managed by bootstrap.
Have a look to https ://getbootstrap.com/docs/5.3/content/tables/ for details.

[Fig. 21] and [Fig. 22] gives a possible page. Let's call it users.html.

```
<body>
  <div class="py-5">
    <div class="container">
      <div class="row">
        <div class="col-md-12"><h1>List of users</h1></div>
      </div>
      <div class="row">
        <div class="col-md-12">
          <div class="table-responsive">
            <table class="table table-striped table-md sortable">
              <thead>
                <tr>
                  <th scope="col" class="col-md-2">user #</th>
                  <th scope="col" class="col-md-3">FirstName</th>
                  <th scope="col" class="col-md-3">LastName</th>
                  <th scope="col" class="col-md-2">Birthdate</th>
                  <th scope="col" class="col-md-2"></th>
                </tr>
              </thead>
              <tbody>
                <tr>
                  <td>1</td>
                  <td>Jean-Yves</td>
                  <td>Martin</td>
                  <td>1963-08-12</td>
                  <td class="text-center">
                    <form action="" method="POST">
                      <button class="btn" formaction="editUser" name="edit"><img src="img/edit.png" alt="edit" class="icon" /></button>
                      <button class="btn" formaction="deleteUser" name="delete"><img src="img/delete.png" alt="delete" class="icon" /></button>
                    </form>
                  </td>
                </tr>
              <tbody>
              <tfoot>
                <tr id="addNew">
                  <td colspan="4"></td>
                  <td class="text-center">
                    <form action="createUser" method="POST">
                      <button class="btn"><img src="img/plus.png" alt="add" class="icon" /></button>
                    </form>
                  </td>
                </tr>
              </tfoot>
            </table>
          </div>
        </div>
      </div>
    </div>
  </div>
</body>
```

Fig. 21 : Users page with bootstrap

Fig. 22 : Users page with bootstrap display

Next, we need a page to edit 1 user. Let's call it user.html. We build it as a table with a title, text lines to get data, buttons to save, ...

[Fig. 23] and [Fig. 24] gives a possible page.

```
<div class="py-3">
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <h2 class="">Create / Edit User page</h2>
            </div>
        </div>
        <div class="row">
            <div class="col-md-12">
                <div class="table-responsive">
                <table class="table table-striped">
                    <tbody>
                        <tr>
                            <th scope="col">user #</th>
                            <td>1</td>
                        </tr>
                        <tr>
                            <th scope="col">FirstName</th>
                            <td><input type="text" class="form-control" name="FirstName" value="Jean-Yves" /></td>
                        </tr>
                        <tr>
                            <th scope="col">LastName</th>
                            <td><input type="text" class="form-control" name="LastName" value="Martin" /></td>
                        </tr>
                        <tr>
                            <th scope="col">Birthdate</th>
                            <td><input type="date" class="form-control" name="Birthdate" value="1963-08-12" /></td>
                        </tr>
                    </tbody>
                    <tfoot>
                        <tr>
                            <td scope="col" colspan="2" class="text-center"><button type="submit" class="btn btn-block btn-primary">Save</button></td>
                        </tr>
                    </tfoot>
                </table>
                </div>
            </div>
        </div>
    </div>
</div>
```

Fig. 23 : User page with bootstrap

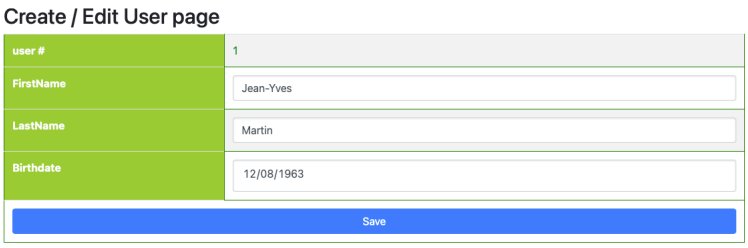


Fig. 24 : User page with bootstrap display

If you do not like button's color, you can change it easily.

3.2.3.3 Bootstrap in books.html and book.html

Books have been created the same way we created users. So, you only have to apply the same tools on books.html and book.html
Have a look to [Fig.25] for these pages.

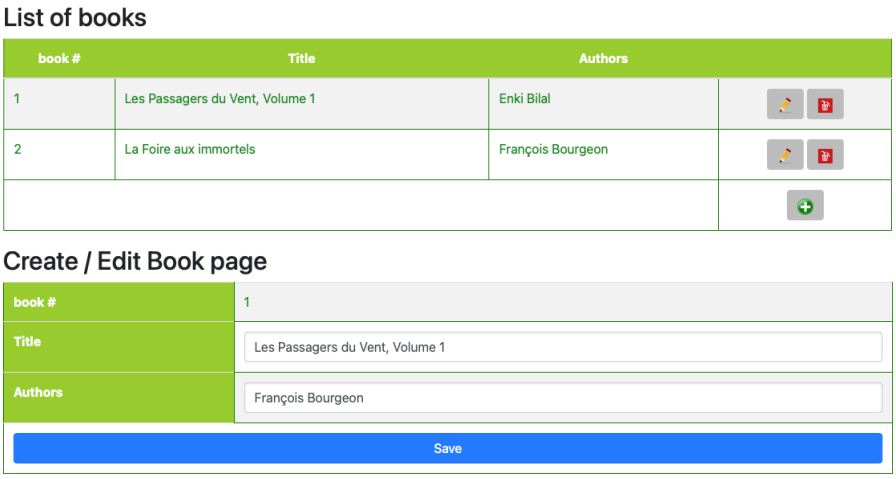


Fig. 25 : Books and book page with bootstrap display

3.2.3.4 Bootstrap in borrow.html

Same for the Borrow page.

For the tag select, you can find documentation here : https ://getbootstrap.com/docs/5.3/forms/select/.

We suggest you add class “form-control” too.

Have a look to [Fig.26] for the page.

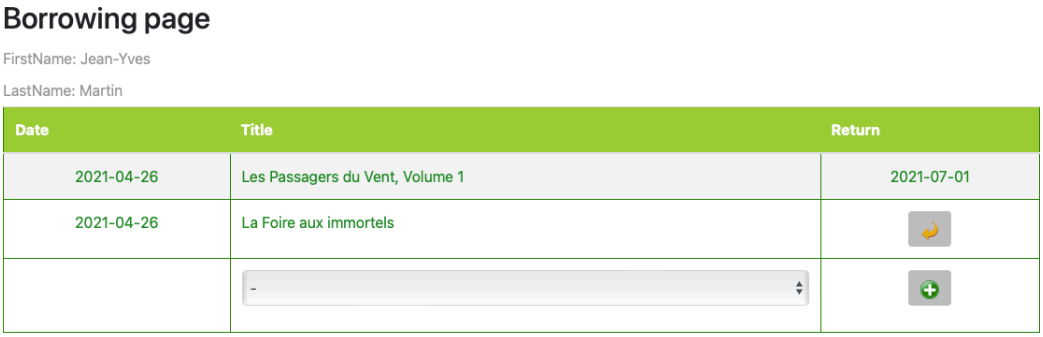


Fig. 26 : Borrow page with bootstrap display

3.2.3.5 Navigating

Books and Users are different page. Of course, we could use buttons to switch from a page to another one.

HTML 5 introduced the **nav** tag to navigate between pages.

Let's try to add a nav tag to users.html and books.html to navigate between pages. You will find documentation here : https ://getbootstrap.com/docs/5.3/components/navbar/.

First we have to define a nav tag with bootstrap classes. We define a "container" inside the nav tag. Then we use an unordered list to display menu elements. Each element is linked to a page.

Have a look to [Fig.27] for the script that is placed at the beginning of body in the 2 pages.

```
<nav class="navbar navbar-expand-md navbar-dark bg-dark">
  <div class="container">
    <div class="collapse navbar-collapse" id="navbar1">
      <ul class="navbar-nav ml-auto">
        <li class="nav-item"> <a class="nav-link text-white" href="users.html">Users</a></li>
        <li class="nav-item"> <a class="nav-link text-white" href="books.html">Books</a></li>
      </ul>
    </div>
  </div>
</nav>
```

Fig. 27 : nav script

And [Fig.28] is the result :

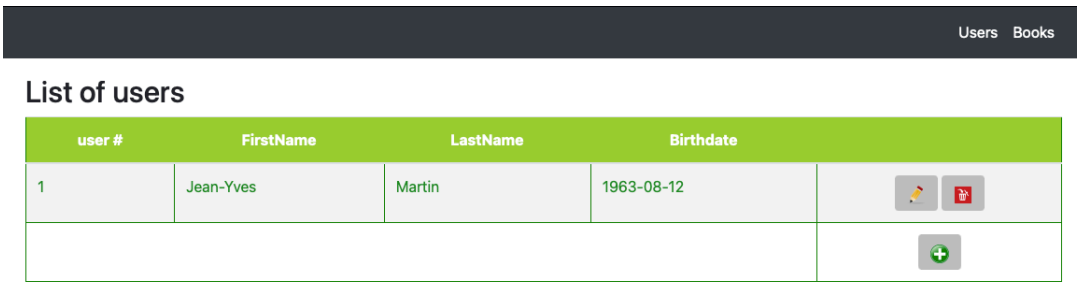


Fig. 28 : nav display

Click on the menu items to navigate.

Questions

In Bootstrap, what is a container used for ?
How large is a grid in terms of columns ?
How do you ensure a table is responsive ?
How do you place a background text in an input text element ?

3.3 Javascript

In this part, we will use some Javascript / Typescript Elements.

3.3.1 Required tools

For this part, you will need :

- A web server to serve your pages.
- a text editor to create pages
- JQuery file. You can of course use a link to the JQuery library but you will reduce downloads if it is on your computer.

3.3.2 A bit of Javascript

3.3.2.1 javascript overview

Javascript is a full programming language. Don't mix up with Java.

Originally, Javascript was created to increase interaction between web pages and users.

Javascript (or JS) is mainly used with web browsers. It includes variables, control structures, functions, objects, ...

TypeScript (or TS) is a superset of Javascript. Typescript compiles scripts before executing them. It includes also elements like typing variables, return types for functions, ... Currently, it can also be used without browsers as a full programming language.

Most of the instructions are synchronous instructions. That means most often, an instruction is completed before next instruction is launched.

Sometimes an instruction has to wait elements before it completes. For example, if an instruction waits for an external request to complete, it shouldn't block the instruction, so it generates a callback function that will manage the request when the request will complete. The instruction ends but it is not fully completed. The callback function will completes later. To manage this, that kind of function returns a promise of result, not that result. Therefore there are also methods that waits for the callback to complete and give a result. That means using asynchronous function may not be written the same way you write the other instructions.

3.3.2.2 Writing Javascript

There are 2 ways of writing JS scripts.

- Scripts inside an HTML file
- External script files included in HTML files

There are 2 ways of launching scripts.

- Events based attributes in tags
- Scripts launched by a script tag

3.3.2.2.1 Writing JS in a HTML file

This way of using Javascript is not the best ones, but, sometimes, for specific uses, ... why not.

Use the file Hello.html you wrote in the previous chapter for these manipulations.

Add a script just after <body>, like in [Fig. 29]

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link rel="stylesheet" type="text/css"
                media="screen" href="css/hello.css"/>
    </head>
    <body>
        <script type="text/javascript">
            document.write("This is my text");
        </script>
        <p>Hello World</p>
    </body>
</html>
```

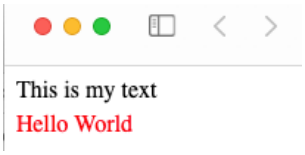


Fig. 29 : JS script execution in an HTML file

Move the script tag after the tag ... , just before the end of body. Save and reload the page. What happens?

3.3.2.2.2 Using JS files

Create a directory “js” in your directory.

Create a file myScript.js in the js directory and add it in your html file. The file myScript.js contains a js function that writes some text. We call the function in the html file. Have a look to [Fig. 30].

Take care when you include the js file : you cannot use the “script” tag as an empty tag (ending with /). Always use a start and end tag, even if it is empty.

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link rel="stylesheet"
                type="text/css"
                media="screen"
                href="css/hello.css"/>
        <script type="text/javascript"
                src="js/myScript.js" >
        </script>
    </head>
    <body>
        <script type="text/javascript">
            writeText();
        </script>
        <p>Hello World</p>
    </body>
</html>
```

```
function writeText() {
    document.write("This is my text");
}
```

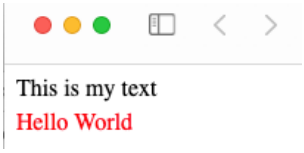


Fig. 30 : JS script execution using JS file

You should have the same result as when the instruction was in the html file.

3.3.2.3 Event capture

When you click on your mouse, when you move it, ... it generates an event. These events are sent by default to the objects that may use them. Very often, most of them are ignored because not used.

However, it is possible to ask an objet to capture the event and manage it. This is done through triggers linked to the objects.

You can add triggers to a large set of objects (understand tags). These triggers take into account events (mouse is over a tag, you click on a tag, you hit a key, you change an input value, ...).

To implement a trigger, you have to :

- add an event capture in the tag
- define what happens when you catch the event.

Let's try to implement a trigger. Have a look to [Fig. 31].

```
<!DOCTYPE html>
<html lang="fr-fr">
  <head>
    <title>Hello World</title>
    <meta charset="UTF-8"/>
    <link rel="stylesheet"
      type="text/css"
      media="screen"
      href="css/hello.css"/>
    <script type="text/javascript"
      src="js/myScript.js" >
    </script>
  </head>
  <body>
    <p>
      <span onmouseover="writeText2();"> Hello</span> World
    </p>
  </body>
</html>
```

```
function writeText2() {
    alert("This is my text");
}
```

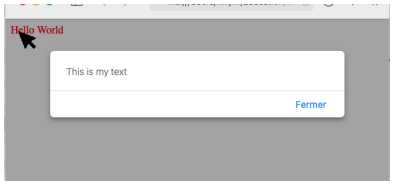


Fig. 31 : A basic trigger

When your mouse goes over the word "Hello", your browser opens an alert message.

What happens ?

- We define a span tag around "Hello". This tag is often used to define new properties on a set of text, an image, ...
- we add a "onmousever" trigger to the tag span
- when the trigger "onmousever" is activated (the mouse is over the span) it launches the selected action that displays a message.

3.3.2.4 Triggers return value

Each time you launch a trigger, the action you launch may have a result :

- true : consider that the event was not managed, so your browser has to manage the event.
- false : means the event has been managed, so your browser must not manage the event.
- if there is no result, acts as if the result was true.

Implement the scripts in [Fig. 32].

hello_31.html

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link rel="stylesheet"
              type="text/css"
              media="screen"
              href="css/hello.css"/>
    </head>
    <body>
        <p><a href="hello_32.html"
              onclick="return true;">Hello</a> World</p>
    </body>
</html>
```

hello_32.html

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link rel="stylesheet"
              type="text/css"
              media="screen"
              href="css/hello.css"/>
    </head>
    <body>
        <p>It works !!</p>
    </body>
</html>
```

Fig. 32 : Using trigger return value

Try to change the return value by false. What does it change when you click on word "Hello".

Here is a short list of triggers you can use :

- onclick : click on a tag
- onmouseover : mouse is placed over the tag
- onmouseenter : mouse arrives over a tag
- onmouseout : mouse was on a tag and leaves
- onchange : tag value changes (input text, select, ...)
- onload : page is fully loaded
- ...

You can have a look to https ://developer.mozilla.org/fr/docs/Web/Events for a more complete list.

3.3.2.5 Using elements properties

Usually, javascript is used to manipulate elements properties. How can we do this ?

3.3.2.5.1 Displaying elements properties in browsers debuggers tools

When your browser loads an HTML documents, it creates a DOM model. That means it keeps in memory the properties of every element it manipulates.

Using your browser debugger, have a look to the properties of a basic html page. [Fig. 33] shows an example of tag properties displayed with SAFARI browser. You can use the same kind of tools with most of the browsers.

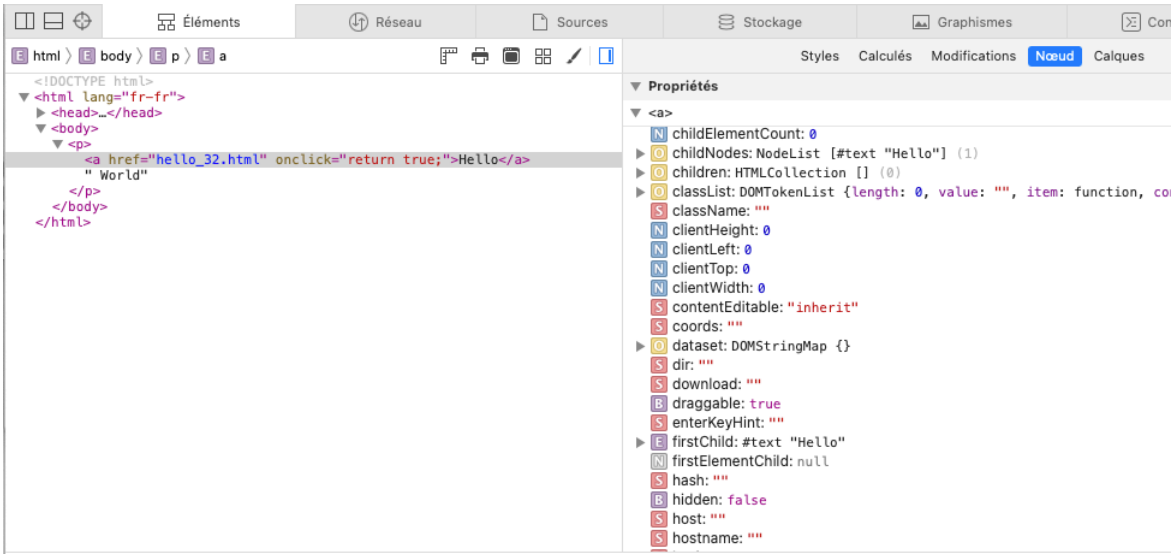


Fig. 33 : Using SAFARI debugger

Do not hesitate to use the debugger to know the properties you can change, their current values, ...

3.3.2.5.2 Manipulating elements properties.

As elements properties are in the browser's memory, it is possible to manipulate then, get their values, change values, ...

For example, in [Fig. 34] you can see on the left the HTML script. We add an id to tag p to get the element. In the javascript function, we can use getElementById to get the element and manipulate its properties.

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link rel="stylesheet" type="text/css" media="screen"
                href="css/hello.css"/>
        <script type="text/javascript" src="js/myScript.js" >
        </script>
    </head>
    <body>
        <p>span id="mytext"
                onmouseover="changeColor('blue')"
                onmouseout="changeColor('red')">
            Hello</span> world</p>
    </body>
</html>
```

```
function changeColor(aColor) {
    var aRef= document.getElementById("mytext");
    if (aRef !== null) {
        aRef.style.color = aColor;
    }
}
```

Fig. 34 : Use 'id' to change tag's properties

"id" is a useful property : an id is unique in an HTML page. Therefore, in Javascript, we can use methods like getElementById to get the element that has a specific id value.

You can also use the property "name" and the method getElementsByName which gives every tag which as the property "name" with a given value. You can get all elements with a specific tag with getElementsByTagName, ...

Methods you can use are in https ://developer.mozilla.org/fr/docs/Web/API/Document#méthodes.

We can also reference elements by using parameters. When you refer to object "this" in a HTML script, it refers to the tag it is in. [Fig. 35] uses that kind of reference : "this" refers to tag "span".

```
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8"/>
        <link rel="stylesheet" type="text/css" media="screen"
                href="css/hello.css"/>
        <script type="text/javascript" src="js/myScript.js" >
        </script>
    </head>
    <body>
        <p><span id="mytext"
                onmouseover="changeColor(this, 'blue')"
                onmouseout="changeColor(this, 'red')">
            Hello</span> world</p>
    </body>
</html>
```

```
function changeColor(aRef, aColor) {
    if (aRef !== null) {
        aRef.style.color = aColor;
    }
}
```

Fig. 35 : Use 'this' to reference an element

3.3.2.5.3 Navigating with element properties

HTML structure is a tree. That means, we could navigate from a node to its sons (if they exist), to its father (if it exists), its siblings. DOM representation includes such a navigation.

If you open an HTML page in your browser, use debugger tools, and check a node property, you should find these :

- parentNode (or parentElement) : link to the parent node
- firstChild (or firstElementChild) : link to the first child. Take care these properties are not equivalent. firstElement links to the first tag. It cannot be linked to a text node. firstChild will give you the first child, that can be a tag or a text node.
- nextSibling (or nextElementSibling) : link to the next sibling in the tree (same parent, next in the same level in tree). nextElementSibling will give the next tag, whereas nextSibling may also give you text nodes.
- lastChild (or lastElementChild) : link to the last node with the same level (same parent). lastChild and lastElementChild may also give you different nodes according to the node type.
- childNodes : list of the node children.
- nodeType : 3 for text, 1 for tags, ...
  see https ://developer.mozilla.org/fr/docs/Web/API/Node/nodeType for detailed values
- document : the root node.

Let's try to navigate through the tree.

Using the same HTML code than in [Fig. 35], now we want that the whole text changes color. That means that from the span element, we have to go to its parentElement, and change parent color.

Check that when cursor goes to "Hello", the text changes color, but it doesn't change when you put it over "world".

3.3.2.5.4 Debugging

In your browser, select debugging tools (see annexes for that).
It is often called web inspector, or web dev tools.

That should open a new window or part the current window to display informations.

Select “Console” tab.

3.3.2.5.5 Using console with a program

In your JS script file, in changeColor, add a line :

```
console.log(aRef.style);
```

This instruction displays the variable content in the console.

Reload the page.
Use your mouse and go to “Hello”. What is the result in the console ?
Move it way. What now ?

[Fig. 36] shows SAFARI console result.

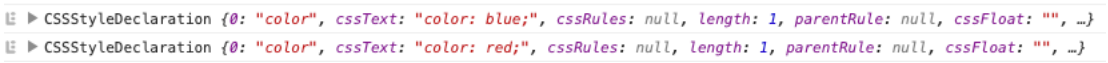


Fig. 36 : The SAFARI console result

3.3.2.5.6 Using console in the browser

In your console, use the following command :

```
console.log(document.getElementById(”mytext”).style);
```

[Fig. 37] shows SAFARI console result.

Fig. 37 : The SAFARI console result when using command

3.4 Using AJAX

AJAX (Asynchronous Javascript And XML) is a javascript tool to interact between your page and the web server. Instead of asking for a page and recharge it in the browser, we ask Javascript to exchange with the server, get data, and update our page according the the received data.

For that, even if it is not required, we will use the JQuery Library.

3.4.1 AJAX Overview

A browser can send requests to a web server. In that case, a web server's response is an HTML file. Such an exchange leads to a page reloading, that scrolls the page to the top.

Often, what we need is an exchange with the web server to update data. This doesn't work with a standart HTTP exchange.

Javascript is designed to interact with the page components. What we need is the javascript ability to directly exchange with the web server. This is the AJAX purpose.

In [Fig. 38], the schema shows an standart HTTP exchange between a browser and an HTTP web server. The whole page is reloaded. Page is scrolled to the top of the page.

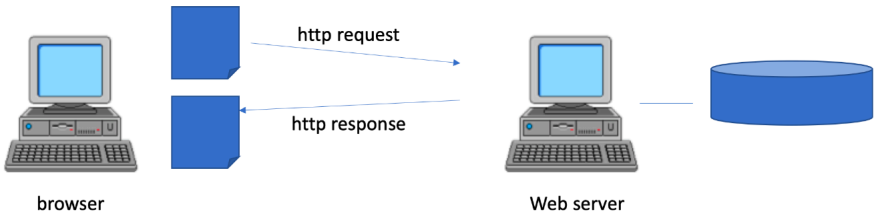


Fig. 38 : Standart HTTP request

In the [Fig. 39] schema, the page uses a Javascript script to exchange with the browser. The page is not reloaded, page cursor does not move, there is only on exchange between the script and the web server. When it receives data from the server, the script can update the current page. That means that, usually, **the request to the web server refers to script execution**, not to an HTML page.

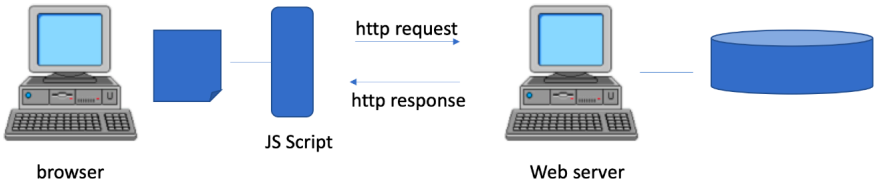


Fig. 39 : AJAX request

We have to take into account some constraints :

- Server is listening for an HTTP request.
- HTTP requests are either GET or POST requests.
- For security reasons, Javascript cannot send a request to any server. **It can only call a script on the same server as the page it is executed in**
- Javascript does not need an HTML response, only data to modify the current page.

That means Javascript/AJAX requests have to respect the HTTP protocol. To manage the exchange, AJAX uses an **xmlHttpRequest** object.

As AJAX does not need an HTML response, but data to modify the current page. That means the web server response might not be an HTML page but a data set. Response can be send using 3 formats : text, XML and JSON

Here is the sequence of an AJAX call :

- a Javascript function is called. It requires data from the web server to complete.
- the Javascript function collect data in the page and prepares an AJAX call.
- the Javascript function sends a request to the server through the AJAX call. The request refers to a script that might be executed on the web server.
- the server execute the requested script with data sent through the AJAX call. It is the same process as a standard HTTP call.
- The script on the server produces a response as a set of data to send back to the javascript function.
- web server send produced data back to the javascript function
- the Javascript function gets the response and extracts requested data
- the Javascript function uses data to alter the current page.

One last problem to solve is : what dose the script while it is waiting for a response ? Should it wait until the response arrives, or should it go on and process the response when it will arrive ? These 2 ways of acting are possible. That means 2 ways of using AJAX calls : synchronous and asynchronous mode.

- In **synchronous** mode, the script is stopped until the response arrives, even if it can take some times. That might have some consequence on what happens in the browser.
- In **asynchronous** mode, the script creates a **CallBack** function that waits of the response, whereas it continues to execute its instructions. That means it cannot exploit the data yet. It has to wait for the callback function to do it. When the response arrives, the callback function is notified and can process it. Technically, this is the callback function that will access to the response and change the page according to the collected data.

3.4.2 AJAX with JQuery

JQuery is a Javascript library that helps manipulating data in the page. JQuery includes a function that helps using AJAX calls.

[Fig. 40] shows what an AJAX call can look like. In this example, we use the most common fields, but you can use other properties.

You can find documentation on this site : https ://api.jquery.com/jquery.ajax/.

```
$.ajax( {
        url :"action to process on the server",
        data : { a JSON object with data to send },
        method :"use GET or POST, default is GET",
        async : use true or false, default is true,
        success : function(result) {
                // Callback function if successful
                // result is the response from the web server, process it
        }
        error : function(theResult, theStatus, theError) {
                // Callback function if an error occurs
                // you should log the error to the console and process an alternate response
        }
});
```

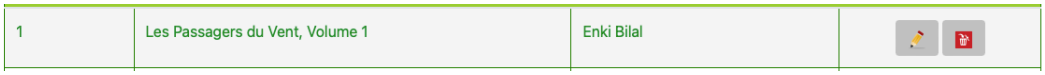
Fig. 40 : AJAX with JQuery

We will use this function for some calls.

3.4.3 AJAX call implementation

Use the file books.html. We will implement a fake function to delete books.

[Fig. 41] shows the row we will act on, and an example of the the HTML script of the row.




```
<tr>
    <td scope="col">1</td>
    <td>Les Passagers du Vent, Volume 1</td>
    <td>Enki Bilal</td>
    <td class="centered">
        <form action="editBook" method="POST">
            <button class="btn" name="edit"><img src="img/edit.png" alt="edit" class="icon" /></button>
            <button class="btn" name="delete"><img src="img/delete.png" alt="delete" class="icon" /></button>
        </form>
    </td>
</tr>
```

Fig. 41 : Book row

When we click on the delete button, we want to collect data (like the book ID) launch a javascript function that launches an AJAX call to delete de book. Then, the row is removed. Of course, this will be a fake call to the server. No database action will be done.

We need a trigger on the delete button. The captured event is onclick. We call a javascript function "deleteBook" you can also give the book id as a parameter. Also, you can give the current button as a parameter. That will allow you to get the row you are in to delete it.

Create a file main.js and write the function "deleteBook". Remember to add the script file in the head tag of the file books.html

Now, consider the javascript function "deleteBook". This function should call a script. Insead, we use a predefined response as a JSON file. In real life, your application server should get the request and give a response to the AJAX call.
We use method POST to call it. When it is done, we remove the line on the page.

[Fig. 42] is a possible deleteBook script and the way you can call it from the HTML script.

```
<button
    class="icon"
    name="delete"
    onClick="deleteBook(1, this); return false;">
    <img src="img/delete.png" alt="delete" class="icon" />
</button>
```

```
function deleteBook(bookID, buttonRef) {
    if (buttonRef !== null) {
        // Collect data - empty
        // Ajax call
        $.ajax({
            url :"prwebStep2.json",
            method :"POST",
            data :{
                "bookID" : bookID,
            },
            success : function (theResult) {
                // get current TR
                var ref = buttonRef;
                while ((ref !== null) && (ref.tagName !== "TR")) {
                    ref = ref.parentElement;
                }
                if (ref !== null) {
                    ref.parentElement.removeChild(ref);
                }
            },
            error : function(theResult, theStatus, theError) {
                console.log("Error : "+theStatus+" - "+theResult);
            }
        });
    }
}
```

Fig. 42 : deleteBook script

Some comments about the function and the trigger we wrote.

- trigger returns false to be sure the call to the form action is not applied.
- take care to the trigger's name "onClick" and write it like this, or it will fail.
- you can remove the form, the trigger will do the job.
- We launch an ajax call. That means you must have include the JQuery file in the head of your HTML file, and place the JQuery file to the right location.
- the file "prwebStep2.json" must be copied at the same level than your file
- the AJAX call can only be addressed to a server. That means you can only use it with an HTTP server. If you open the file from the browser (protocol is file :// instead of http ://) it will fail.
- your HTTP server should launch a script to build the response. Insead, we pre-built a response as a JSON response that is interpreted when received (never use this in real life)
- If ajax call is successful, we remove the current line. For that, we start from the button and we take the parents until we reach the TR tag. When TR tag is reached, we remove it from the children list of its parent.

Use your browser to load the books.html file from an URL like http ://localhost/.../books.html

Click on the delete icon of the book row. That should call the AJAX fonction, which is supposed to delete the book, and remove the line in JS. Ensure it works for every row.

Questions

How can you get any element in the HTML page to change one of its properties ?
What is the principle of an AJAX call ?
Can I call any script on any server with AJAX ? What are AJAX restrictions ?

4 Our database

For each of the implementation we use a database.

The database script is available from the materials on the pedagogic server (hippocampus).
Script is CreateDatabase.sql

Open PgAdmin and connect to the server.
Create an user “prweb” who can connect to your server. No need for a superuser.
Create a database “prweb” with owner “prweb”.

Right clic on the database and select “Query tool”.
Copy and paste content of the file “CreateDatabase.sql” to the Query tool window.
Run script.

Your database should be available with 3 tables dans some data in the tables.
Feel free to change data.

5 All in One Web Application

You have to choose between **PHP-Symfony** and **Spring** for this part of the practical work.
We strongly recommand Spring.

5.1 Frameworks : some basic principles

When you write web applications, you can of course do it by yourself, but are you sure you can manage collaborative development, security aspects, ...?

Using frameworks is a good solution. They are usually build on the MVC model. They include security management, database management, ...

5.1.1 MVC paradigm : Model, View, Controller

The MVC model is a way of developing applications, including web applications.

- Model represents data used in the application. Usually, there is a database, and tools to manipulate data. Often, it includes tools to manipulate informations in the database, like ORM (Object Relational Mapping).
- View represents screens displayed to user. They are usually templates that are filled according to the data required by user. They include mechanisms that includes data in the templates.
- Controller is the link between user and the application. When the application receive a request, it is send to a controller that gets the request, it defines which informations are required, it interacts with the model to get data, it defines the most adapted view to display informations, and it sends the view with the required informations to user.

[Fig. 43] shows how these elements interact.

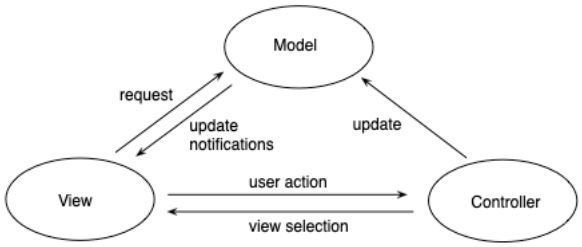


Fig. 43 : MVC paradigm

5.1.2 Using ORM

Object Relational Mapping (or Mapper, it depends on the definition) manages links between databases and objects in the framework.

First we have to represent database tables as objects for the program. That means you define objects to manage tables in the database. They are usually called "**items**" or **entities**.

To manage the items, we use "**repositories**". A repository manages a kind of item. it implements CRUD methods (to create and save items), find list of items, delete items (remove from database), update items (update in database). One of the main purpose of repositories is to hide SQL management.

5.1.2.1 Entities / Items

Entities are tables in the database. Most of the tables are translated as entities. Usually, tables that link informations between tables are not.

- Each object has attributes according to the columns in the database
- Links (foreign keys, table that links tables) are also attributes in the objects.

Usually annotations are used to define items properties.

- "**Item**" means the class define a class linked to a table
- "**Table**" may define the table name associated with the object
- "**id**" to define the table attribute primary key. It may be linked to autoincrement annotations.
- "**NotNull**" tells the attribute can't be null.
- "**Column**" define the database column associate with.
- When you use foreign keys or joined tables :
  - "**OneToOne**" : the attribute is linked to a unique attribute in the linked table, and this attribute is only linked to this object.
  - "**ManyToOne**" : the attribute is linked to a unique attribute in the linked table, but this attribute can be linked to many objects.
  - "**OneToMany**" : the reverse attribute of "ManyToOne". The attribute type is an object array.
  - "**ManyToMany**" : this attribute may be linked to a set of objects, and the linked objects may be reversely linked to a set of objects. The 2 attributes type are arrays. Usually, this represents a linked table.

Depending on the programming language, definitions can be a little bit different, but they look the same.

5.1.2.2 Repositories

For each entity, a repository may be defined. A repository manages one kind of entity.

Usually, the framework includes default methods to create, remove, update, find data, the CRUD functions. So you do not need to implement most of the methods because they are already defined in the framework.

However, if you require specific methods to manipulate entities, you may define them in the repository. When you want to manipulate the item, you may use the repository for that.

Sometimes repositories are not defined by default. So you will have to create them by yourself.

Remember : you shouldn't manipulate the database, except through repositories. Or sometimes through the start method.

5.1.3 Views

When server replies to the browser it may send HTML files, JSON files, XML files, ... and AJAX responses too.

Views are **templates** that define how data are send to the user tool (the browser). Therefore, a view is a generic structure that manages the way to display informations.

A view receives data from controller and is in charge of displaying it. That means the controller has to send all required data to the view.

Often, a view uses a specific language to describe the template (TWIG for Symfony, JSP for Spring, ...). It contains HTML elements, CSS files, JS scripts, images, ... and uses specific instructions for tests, loops, ...

5.1.4 Routes

When interacting with the browser page, the user's action may result in a request for a new page or an AJAX call. In both cases, the request sent to the server contains information about what is required. These informations includes :

- site reference (http ://....)
- resource requested (a page, a script to be launched, ...)
- parameters.

At the beginning, the resource requested was an existing file that was returned as a response.

With web application development, the resource was a script to be executed on the server. This script produced a response sent back to the browser.

The problem is that when you call a script you give the name of the files you are using on your server, so, to imprive security, and readability, developpers added **Routes** to the frameworks. Theses routes are path to classes and methods, and are managed by the server.

In the browser request (directly, from a form, an AJAX call, ...), you call a Route. Server translate the route to a method to call. The corresponding method is executed. The response is sent back.

Questions

When a browser send a request to a MVC based application, which module does what ?
Why can't you have 2 same routes in an application ?
What are entities and repository used for ? Why is there a repository for each managed entity ?
What is the role of an ORM ?
Why are there some tables that are not implemented as Items ?

5.2 SPRING and JPA

5.2.1 Required tools

For this part, you will need :

- Java JDK >= 17. You can use the one you want. We checked different versions up to 24. Also take care, depending on Java version, the libraries versions may be different.
- Apache Tomcat 10 (maybe it works with version 11 too).
  Be sure you configured file tomcat_users.xml.
- Netbeans >= version 24, and maven
- JQuery file, Boostrap files, ...
- PostgreSQL server >= version 15

Installation guides are in the annexes.

5.2.2 A bit of configuration

Start Netbeans. In the left panel, select “Services”.
Clic on the >character behind “Servers” to open it, and check if your server is already installed.

If not, right clic on “Servers” and select “Add server”.
Have a look to [Fig. 44] for the configuration screens.

Select “Apache Tomcat or TomEE” as server type.
Give a name to the server in the “Name” area.
Click on “Next”.

Give tomcat directory location on your disk, the root of the Tomcat directory, not the bin one.
Give the username and the password of an admin user. You should have created it in tomcat-users.xml
Validate to create server by clicking on “Finish”.

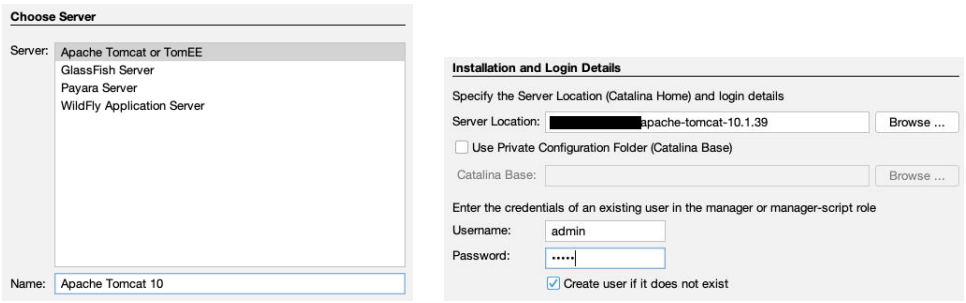


Fig. 44 : Netbeans adding tomcat server

5.2.3 Creating project

5.2.3.1 Create new Web application

Start Netbeans.
Ask for a new project. Select “Java with Maven” and “Web Application Project”.
[Fig. 45] shows the web application project selection.

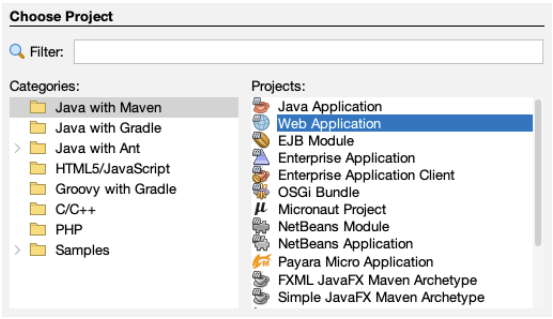


Fig. 45 : Netbeans create web application project

When you create a web application for the first time, you may have to install some components.
Click on “Download and Activate” to install the components. Then validate forms.

Then, we create the project. Fill the form like in [Fig. 46]. Of course, you can change project location, Group ID, ...
Keep in mind that the group ID defines the package you work on.

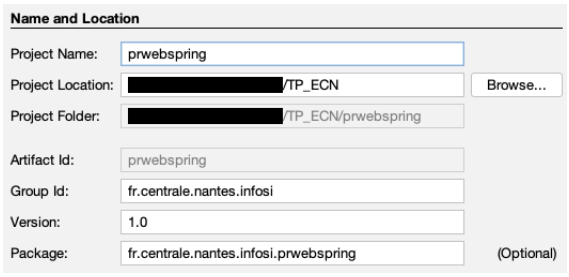


Fig. 46 : Web Application creation form

Now select your Web Application Server. [Fig. 47] shows the select server form.

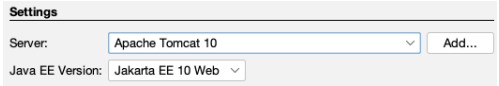


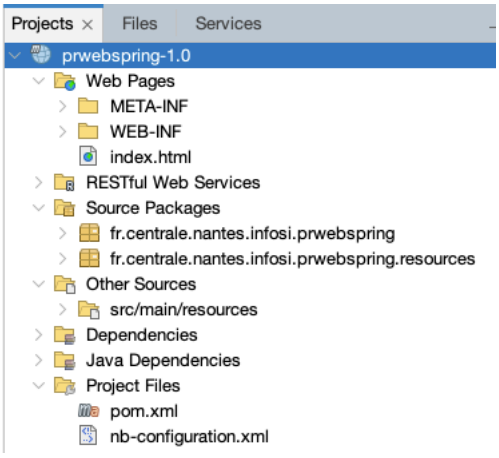
Fig. 47 : Select Application server

As a server version, we use **Java EE 10 Web**.
When you use a Tomcat 9 server, you can use Java EE 8 Web.
Jakarta is an evolution of Java EE and J2E, but there are some differences. Java EE still belongs to Oracle, but their rights were given to the eclipse foundation. As the Eclipse foundation cannot change what they need, they renamed the project as Jakarta.

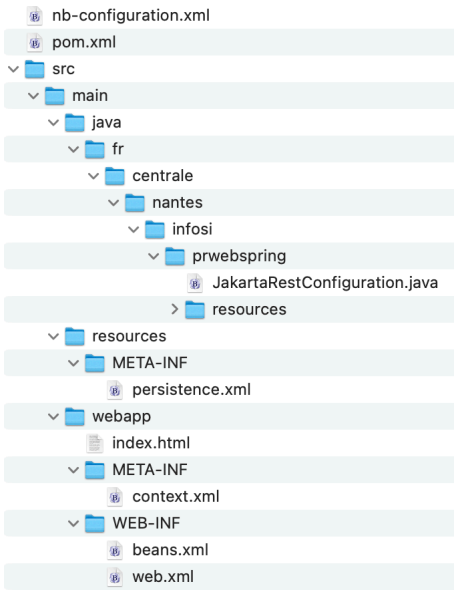
If you have no server defined, or none corresponding to the one you want (that means you didn't have a look to "a bit of configuration"), click on the "add" button, on the right of the server selection. Then follow instructions we gave in the "Configuration" section. Maybe you should have a look to [Fig. 44]. Of course, if your server has already been added, select it instead of adding it twice.

Click on "Finish" to create the project.

Ok, Project is created.
[Fig. 48] is what you should have in Netbeans and in your Project directory. We will change several things that will add/remove some elements.



Netbeans project



Project directory

Fig. 48 : Web project structure

5.2.3.2 Project configuration

We need to check everything is ok for our projet, and add the Spring MVC library.
Right click on your project and ask for "Properties".

5.2.3.2.1 Compiling version

In the properties screen, have a look to "Build" - "Compile" in the left area.
You will see on which plateform you will run the project.

Ensure your Java Plateform is >= 17. Also set the Target Release >=17.

The target release defines on which java version your application can run on. The Java Plateform defines, on your computer, which one you will use.

5.2.3.2.2 Add MVC Spring framework

In the properties screen, use "Frameworks" item in the left area.

Click "Add" to add a framework. Select "Spring Web MVC". [Fig. 49] shows the framework selection.

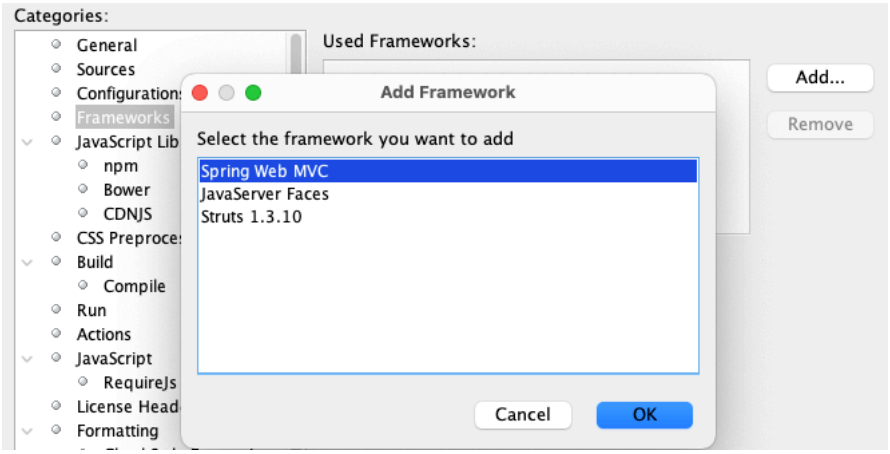


Fig. 49 : Add Spring Web MVC Framework

Then we select the version we want for SPRING. For "Spring Library", select version 5.xx (5.3.31 when we write these lines). We will change it a bit later in the file pom.xml
Ensure "Include JSTL" is selected.
Then switch to tab "Configuration" and change dispatcher mapping to "*.do". This mapping tells SPRING which URL it will catch. Every URL that ends with ".do" will be routed to SPRING listener and

managed by SPRING controllers. Of course ".do" is not mandatory and you can change routes ending by what you want, but remember to take this into account for your controllers.

[Fig. 50] shows the 2 configuration forms you may have.

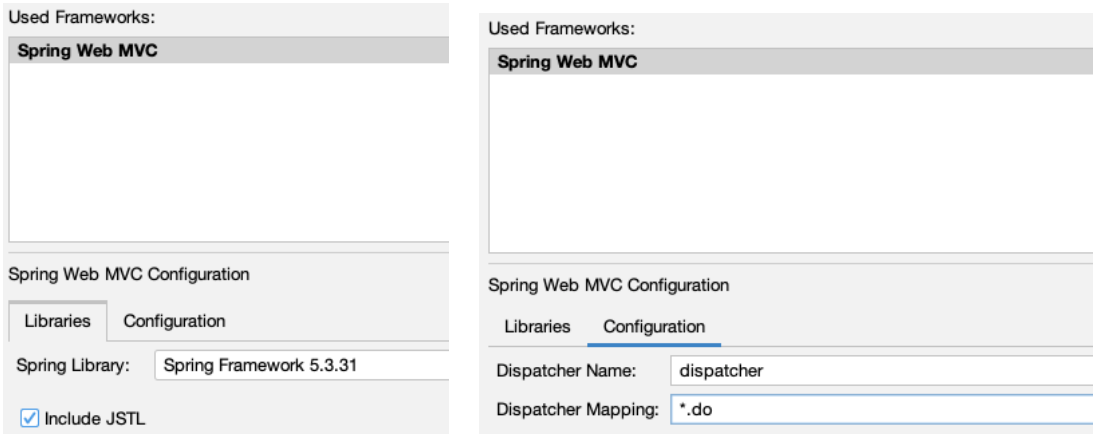


Fig. 50 : Libraries and configuration for Spring Web MVC Framework

Click "ok" to save the modifications, and come back to "Properties".

Now you should have a "Spring Framework" item in the left panel.
You can see that spring configuration is managed by 2 files :

- applicationContext.xml : manages the application, how we write controllers, repositories, ...
- dispatcher-servlet.xml : manages the link between the controllers and the views.

We will configure these files a bit later.

5.2.3.2.3 Application server to launch the application

Have a look to tag "Run".

The content should confirm on which server you run the application.
The "Context Path" gives the path use for the URL.
You can also select the browser you will use.

5.2.3.3 A bit of configuration in pom.xml

In your projet, open the "Project Files" section (bottom of the list) and open pom.xml
This file is the configuration files for the project.
And it is a real nightmare.If something is wrong, project will compile but running may fail.

If you already tried to run the project, you may have experiment some trouble because some elements are improperly set.

You can have a look to the file pom.xml in the materials to copy the elements in your file.
Or you can do it by yourself in your pom.xml project file. Good luck.

In the "build" section, ensure **org.apache.maven.plugins / maven-war-plugin** is set to a at least 3.3.2, or 3.4.0 if possible.

Remove content of "Dependencies" and add the following ones (first part is groupId and second one artifactId).

groupId	artifactId	version
jakarta.platform	jakarta.jakartaee-api	10.0.0
jakarta.platform	jakarta.jakartaee-web-api	10.0.0
jakarta.servlet	jakarta.servlet-api	6.1.0
jakarta.persistence	jakarta.persistence-api	3.2.0
org.springframework	spring-core	6.2.7
org.springframework	spring-beans	6.2.7
org.springframework	spring-context	6.2.7
org.springframework	spring-web	6.2.7
org.springframework	spring-webmvc	6.2.7
org.springframework	spring-tx	6.2.7
org.springframework	spring-aop	6.2.7
org.springframework	spring-messaging	6.2.7
org.springframework	spring-jdbc	6.2.7
org.springframework	spring-context-support	6.2.7
org.springframework.data	spring-data-jpa	3.4.4
org.eclipse.persistence	eclipselink	4.0.6
org.springframework.boot	spring-boot-starter-data-jpa	3.4.4
jakarta.servlet.jsp.jstl	jakarta.servlet.jsp.jstl-api	3.0.2
jakarta.servlet.jsp	jakarta.servlet.jsp-api	4.0.0
org.glassfish.web	jakarta.servlet.jsp.jstl	3.0.1
org.json	json	20230618
org.eclipse	yasson	2.0.4
com.fasterxml.jackson.core	jackson-databind	2.18.3
org.slf4j	slf4j-api	2.0.17
org.postgresql	postgresql	42.6.2

Maybe newer versions may work. You can check it in the maven site.

If you have any trouble building this, have a look to the pom file in the materials.

5.2.4 Check it works.

Try to build the project.
Take care, maven might download many files, so you must have a connection to the internet.
If you have a mojo error, it may be because your org.apache.maven.plugins version is too old. Check pom.xml

Try to run/debug the project.
That should eventually compile the project and run it (transfert files, decompress, execute) on the server you selected.
Be sure the script that launches/stops the tomcat server are executable, or the server will not be launched.
You will have to give tomcat admin login/password to launch it (each time you relaunch Netbeans).

[Fig. 51] shows the result you may have in your browser.

Fig. 51 : First Spring project run

Why does it works ?
Have a look to dispatcher-servlet.xml in "Web Pages/WEB-INF" (remember Spring configuration files ?).

Look at bean "urlMapping". It tells that we map route "index.do" with the bean "indexController".
Now, look at the bean "indexController" bottom of the file. It tells you this is a default managing way and the associated view is "index".

Then have a look to the bean "viewResolver" (see [Fig. 52]).
It tells that view mapping consists in prefixing views by "/WEB-INF/jsp/" and suffixing it by ".jsp".
That means view "index" is mapped to "/WEB-INF/jsp/index.jsp". Have a look to this file. Do you understand what is displayed in your browser ?

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/jsp/"
      p:suffix=".jsp" />
```

Fig. 52 : ViewResolver in dispatcher-servlet.xml

The directory name defined in the “viewresolver” can be set to the one you want.
You only have to create (or rename) the directory in “WEB-INF” and replace the directory name in the prefix line.

Usually we use another name like **“views”** to manage the jsp files, but you can keep “jsp”, or use another name if you want. Remember, you have to change the “viewresolver” according to the name you choose.

The viewresolver is a bean you MUST NOT REMOVE.

5.2.5 Stoping server

There are 2 ways to stop a tomcat server.

- In the left panel in Netbeans, use tab “Services”. Open “Servers”. You should see your server. Right click and ask to stop it.
  Do you notice you can do some other actions ?
- When it is running, in the bottom panel, you should have a tab “output”.
  Select tab “Apache Tomcat 10” (or the name you gave to your server), or the name you gave to your server.
  On the left, you should see a small red square. Click on it to stop server.

[Fig. 53] shows the 2 ways to stop server.

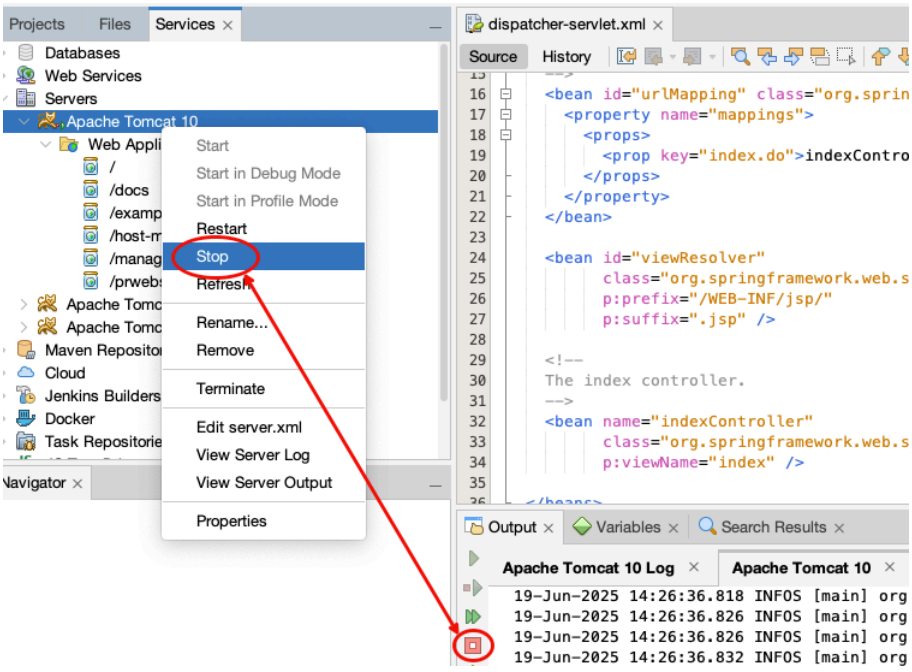


Fig. 53 : Stop server

5.2.6 Configurations

5.2.6.1 Creating packages

Controllers manage interaction with users, the requests.
Entities (or Items) represent the objects we manage and which almost reflects the database structure.
Repositories manage items and their instances in the database.

To manage these classes / interfaces, we use different java packages. One per role.

In the netbeans project, open “Source Packages”.
Create 3 packages (right click on fr.centrale.nantes.infosi.prwebspring) :

- fr.centrale.nantes.infosi.prwebspring.controllers : for the controllers
- fr.centrale.nantes.infosi.prwebspring.items : for the items
- fr.centrale.nantes.infosi.prwebspring.repositories : for the repositories

Of course you can change projets package names / paths according to your projet. You will have to give YOUR packages names in the configuration files.

You can also remove the files initially created in the project, we will not use them.

[Fig. 54] shows the result.

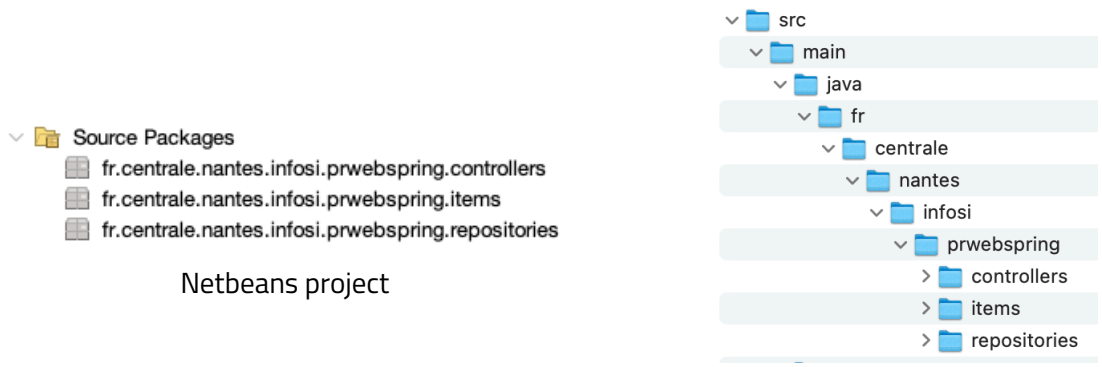


Fig. 54 : Adding 3 packages : controllers, items and repositories

5.2.6.2 Web Pages

Root directory for your web pages is “Web Pages” in your project. That corresponds to directory “webapp” in your project directory on your disk.

In that directory, you should find :

- META-INF : Configuration directory for execution.
  It contains a file “context.xml” that gives the route for your application.
  DO NOT CHANGE IT if you are not required to.
- WEB-INF : Configuration directory for the application.
  It contents :
  - some configuration files.
    - web.xml that defines servlet, listeners, the starting page, …
    - applicationContext.xml that defines how the application works, how to find controllers and repositories
    - dispatcher-servlet.xml that defines how to find controllers, repositories and views.
    - beans.xml that define the list of controllers in some spring versions
  - html, jsp files : used when you start the application

In “Web Pages”, you can also add your own directories to manage javascript files, css, images, …

5.2.6.3 Configuring web.xml… or not

web.xml tells many things about the way the application works.

- “contextConfigLocation” : where is the file applicationContext.
- the servlet “dispatcher” (the servlet-name sets the name) : tells which module manages the servlets.
- the tag “servlet-mapping” for “dispatcher” : explains that the routes that ends with “.do” are sent to the dispatcher to be managed by the servlet.
- the tag “welcome-file-list” : tells which file is the starting page.

We have no specific configuration to do, but you should understand what you have in this file to be able to change some informations when you have to.

Initialization files (we will not use them in this project) should also been defined here.
For that you would have to define a specific listener.

If you have any trouble with forms and character encoding you can enforce form encoding in this configuration file.

5.2.6.4 Configuring applicationContext.xml

We have to tell our application where are our controllers, repositories, how to use them, ...

When we will write our classes, it would be nice to avoid managing everything through configuration files. We will use **annotations** in the files.

The application, when it compiles will have to decode the annotations and produce the right instructions to manage things as they should be.
What do we have to do ?

- Tell the application where are the controllers
- Tell the application where are the repositories
- Tell the application we use persistence and which environment we use
- Tell the application that we let JPA manage informations
- And of course that we manage things with annotations

These elements are described, in terms of syntax by XML schema. They are located on internet and may be included in our XML configuration files with URL and URI.

Have a look to **applicationContext.xml** in **Web Pages/WEB-INF**, and especially the first tag, "beans". "beans" is the root tag, the one that contains the other tags. It also defines the namespaces we will use in the file.
Namespaces are associations between names we use in the file and schemas. That allows to manage a specific set of tags defined in the associated schema.

Have a look to the way the namespaces are defined :

- "xmlns :xxx" means we define a namespace xxx (xmlns means XML Name Space). It is linked to an URL to the namespace schemas location.
- "xsi :schemaLocation" defines the namespace locations. In each namespace location, it tells which file to use. So, for each namespace, there are 2 informations, the schema location and the file location.
- To use a namespace, we use the namespace and a tag defined in the schema.
  For example, to use tag annotation-driven in namespace mvc, we use mvc :annotation-driven

You can also have a look to the URL with your browser. You will see it contains several files, that is why we have to tell which one to use.

For example, namespace aop is located in http ://www.springframework.org/schema/aop
For aop, the file we use (the entry point) is spring-aop-4.3.xsd

Currently, some are already defined, like aop, tx We need some more like mvc, context and jpa.
Have a look to [Fig. 55] for the definitions to add.
Also, you can find these definitions in the appropriate file in the materials,and copy/paste them (we strongly recommand that if you want to avoid miswriting).

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
       http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd
       http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
       http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">
```

Fig. 55 : Add namespaces to applicationContext.xml

Now we can use the namespaces to define the elements we need.
Currently, applicationContext.xml is empty, there are only some comments.
We have to add some elements :

- Tell we use annotations
- Give controllers' package
- Give repositories' package and the way transactions are managed
- Link repositories to the Persistence Unite (the mechanisme that will manage the link with the database).

[Fig. 56] shows what you should add. Remember to give YOUR informations in base-packages so that they correspond to your project.

For the persistenceUnitName, we choose **"infosi_prwebspring_war_1.0PU"**.
As this name is not defined yet, you can use a different one if you want, but **remember that the link to the persistence unit is here**.

```
<!-- ADD PERSISTENCE SUPPORT HERE (jpa, hibernate, etc) -->
<mvc:annotation-driven/>
<context:component-scan base-package="fr.centrale.nantes.infosi.prwebspring.controllers" />

<jpa:repositories base-package="fr.centrale.nantes.infosi.prwebspring.repositories"
                  entity-manager-factory-ref="entityManagerFactoryBean" />

<context:load-time-weaver/>
<bean id="entityManagerFactoryBean"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="infosi_prwebspring_war_1.0PU"/>
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
  </property>
</bean>

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactoryBean" />
</bean>
</beans>
```

Fig. 56 : add configuration elements

5.2.6.5 Configuring dispatcher-servlet.xml

When a request arrives (with a specific URL to reach), we have to send the request to the right controller.
When a controller displays information through a view, it calls a JSP file (Java Server Page).
So we have to know where they are located. This is the purpose of "dispatcher-servlet.xml".

What do we have to change ?

- We will manage controllers, repositories, ... through annotations.
- We also have to tell where are the packages that contains controllers
- We have to define views locations

Remember what we did for "applicationContext.xml", **the lines we added in tag beans** ?
We need to add 2 namespaces : mvc and context.
As the namespace we have to add are nearly the same as for applicationContext.xml, you can copy/paste the beans definition in dispatcher-servlet.xml

Ok, now we can use the namespaces and change the file a little bit.

- We do not need the bean urlMapping definition any more, so remove it.
- We have to tell SPRING that we use annotations with mvc :annotation-driven.
- We have to explain where are located the controllers with context :component-scan.
- The viewResolver bean should be ok. It manages the link between the views' name and the files. You can rename the views (jsp) directory in WEB-INF, and change the information in this file.
  For example, if you want to use "views" for the views, you have to set prefix to "/WEB-INF/views/" and rename the directory "jsp" in WEB-INF to "views".
- Bean IndexController is not required any more because we use annotations. So remove it.

[Fig. 57] shows how you should have changed the file dispatcher-servlet.xml
Do not forget to set YOUR package in "base-package".

```
    <mvc:annotation-driven/>
    <context:component-scan base-package="fr.centrale.nantes.infosi.prwebspring.controllers" />

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          p:prefix="/WEB-INF/views/"
          p:suffix=".jsp" />
</beans>
```

Fig. 57 : dispatcher-servlet.xml configuration

Try to compile it. If you have a problem, maybe it comes from your source version selection.

Do not try to run it, it will fail because we haven't define persistence environment yet.

5.2.7 Database connection, Entities generation and Persistence

Ok, now let's create the entities.
We create them from the database tables.

Therefore, we need a connection to the database.

This step also creates persistence unit and entities.

In Netbeans, **right click on the items package** (should be "fr.centrale.nantes.infosi.prwebspring.items").
Select New.

If "Entity Classes from database..." is available, select it.
If not, select "Other..." and "Entity Classes from database...".
Select folder "Persistence" and item "Entity Classes from database...".

[Fig. 58] shows the selection by menu and by form.

Fig. 58 : Ask to create entities from database

Next, you have to define the database connection informations.
[Fig. 59] shows the database form selection.

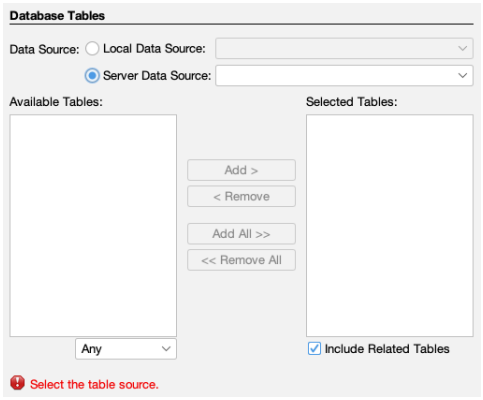
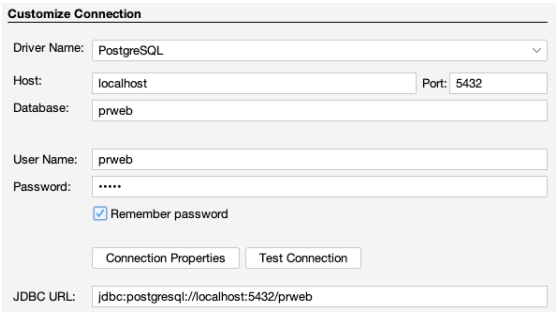


Fig. 59 : Select database and entities to be created

Switch to "Local Data Source", and in the menu on the right, create a "New Database Connection".

Select your Driver (the Database server type) : PostgreSQL. When selected, click "Next" to continue.

Give database connection informations (host, database, User name, password). You should check “Remember Password” and click on “Test Connection” to be sure it works.
[Fig. 60] shows what you should have.



- host is the database server location (maybe your computer so ”localhost”)
- database name is the name of your prweb database
- ”user name” and ”password” are the ones you defined for your database

Do not forget the ”Test connection” button to check it is ok.

Fig. 60 : Create database connection

Click “Next” when it is ok.

Select schema to use. You can keep “information_schema” or switch to “public” (prefer public). This indicates the schema (in term of database) to use.
Click “Next” to continue.

You have to choose a “connection name”.
This is the name Netbeans will keep for your connection if you want to re-use it (have a look to Services / Databases). You can let the default name, or change it if you want.
Click “Finish” to validate database connection informations.

If everything is ok, Netbeans connects to the database and retrieves your database schema.
It proposes entities you can create.
[Fig. 61] shows the table selection to create entities.

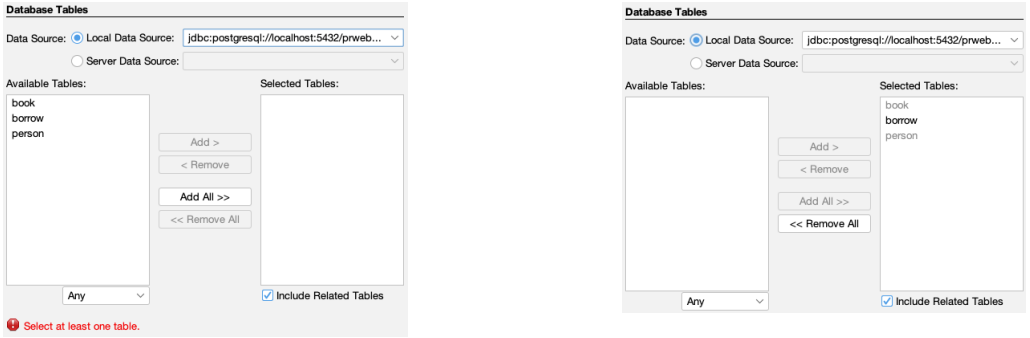


Fig. 61 : Select entities to create

Select the ones you want (Person, Book, Borrow). Note that if you select linked tables, the linked ones are automatically added.
Select All.

Click “Next” to continue.

Next screen defines some parameters about the entities generation.
You should not have to change something.
Click “Next” to continue.

This screen manages mapping informations.
If you prefer using Lists instead of Collections, you can change it, but Collection is ok for our project.
Keep other selections as they are.
Click “Finish” to continue.

At this step, Netbeans generates entities in the package you selected, “fr.centrale.nantes.infosi.prwebspring.items” for us. It takes informations from the database (columns, types, primary key, foreign keys, columns nullable/not, ...) and build classes according to the informations.

[Fig. 62] shows an example of a generated entity : “Book.java”.

```
@Entity
@Table(name = "book")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Book.findAll", query = "SELECT b FROM Book b"),
    @NamedQuery(name = "Book.findByBookId", query = "SELECT b FROM Book b WHERE b.bookId = :bookId"),
    @NamedQuery(name = "Book.findByBookTitle", query = "SELECT b FROM Book b WHERE b.bookTitle = :bookTitle"),
    @NamedQuery(name = "Book.findByBookAuthors", query = "SELECT b FROM Book b WHERE b.bookAuthors = :bookAuthors")})
public class Book implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "book_id")
    private Integer bookId;
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 256)
    @Column(name = "book_title")
    private String bookTitle;
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 256)
    @Column(name = "book_authors")
    private String bookAuthors;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "bookId")
    private Collection<Borrow> borrowCollection;
```

Fig. 62 : Generated entity (Book.java)

Netbeans should have generated 3 classes, one for each entity.

Note that you can re-generate entities when you want.
It will add the informations to the items, creating new ones if necessary.
Unfortunately it doesn’t remove old data, so if you remove something in tables, it will not be removed from entities.

Have a look to Book.java. You have the beginning of the class in [Fig. 62].

Spring uses annotations to manage entities.

- @Entity tells the file is an entity
- @Table gives the name of the corresponding table in the database
- @NamedQueries gives predefined requests you can use to get persisted objects. Note that each @NamedQuery is written in **JPQL** (close to SQL), and uses class attributes instead of table fields.
  Also note that each @NamedQuery has a name.
  NB : you can define yours if you want to.
- the class implements Serializable, that means the entities (the instantiated objects of this class) are stored as writable objects (arrays of bytes usually).
- @Id defines the primary key (Primary keys with several attributes are generated with another annotation)
- @GeneratedValue manages auto-increment
- @Basic tells if the attribute can be null or not. @NotNull too, but they do not manage the same elements in the scripts, so you may use the 2 ones.
- @Column gives the attribute associated column in the table
- @OneToOne, @OneToMany, @ManyToOne and @ManyToMany manages table links

Note, at the end of the attributes definition, the attribute borrowCollection manages the borrowed books. You should find the reverse link in the borrow entity.

Attributes are private, so we need getters and setters to access them. These accessors are already generated.
Note that you can add your own getters and use them as you want.

Now have a look to Borrow.java
You may find the same kind of annotations.
Have a look to the way it manages the link with Book.

- @ManyToOne tells 1 Borrow is linked to Many Books
- @JoinColumn defines join elements. “book_id” in table “borrow” refers to “book_id” in table “book”.

Most often you will not have to change them, but if you add/remove columns, you might know how it works.

5.2.8 Persistence file

In the Netbeans project, have a look to the "Other Sources", then "src/main/ressources", and "META-INF". You should find a file "persistence.xml" that manages persistence, that means table and entities.

Open the file "persistence.xml". Use "Source" tab. [Fig. 63] shows our persistence file.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.0" xmlns="https://jakarta.ee/xml/ns/persistence" xmlns:xsi="http://
  <!-- Define Persistence Unit -->
  <persistence-unit name="my_persistence_unit">
    <class>fr.centrale.nantes.infosi.prwebspring.items.Person</class>
    <class>fr.centrale.nantes.infosi.prwebspring.items.Book</class>
    <class>fr.centrale.nantes.infosi.prwebspring.items.Borrow</class>
  </persistence-unit>
</persistence>
```

Fig. 63 : Persistence file

Have a look to tag "persistence-unit".
The "name" refers to the persistence unit in our application (the object storage mechanism). It **MUST BE** the one defined in **applicationContext.xml**, in bean "entityManagerFactoryBean", property "persistenceUnitName". Name it in this file as it should be.

The file "persistence-unit" contents our instantiated entities. They are defined in the "class" tags.

We have to add some informations.

We will manage connection locally. In **persistence-unit** tag, add **transaction-type="RESOURCE_LOCAL"**. Connection informations will be locally defined.

```
<persistence-unit name="infosi_prwebspring_war_1.0PU" transaction-type="RESOURCE_LOCAL">
```

Next, we have to tell what kind of provider we use. Add this line inside the "persistence-unit" definition.

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```

At last, add connection properties bottom of the tag like in [Fig. 64].
Of course, you might change values according to you connection parameters.

- javax.persistence.jdbc.url = database jdbc connection.
  Should be : jdbc :**protocol** ://**server** :**port**/**database**
- javax.persistence.jdbc.driver : driver type
- javax.persistence.jdbc.user : login to connect to database
- javax.persistence.jdbc.password : password to connect to database

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.0" xmlns="https://jakarta.ee/xml/ns/persistence" xmlns:xsi="http://www.w3.org/20
  <!-- Define Persistence Unit -->
  <persistence-unit name="infosi_prwebspring_war_1.0PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>fr.centrale.nantes.infosi.prwebspring.items.Person</class>
    <class>fr.centrale.nantes.infosi.prwebspring.items.Book</class>
    <class>fr.centrale.nantes.infosi.prwebspring.items.Borrow</class>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/prweb"/>
      <property name="jakarta.persistence.jdbc.user" value="prweb"/>
      <property name="jakarta.persistence.jdbc.password" value="prweb"/>
      <property name="jakarta.persistence.jdbc.driver" value="org.postgresql.Driver"/>
    </properties>
  </persistence-unit>
</persistence>
```

Fig. 64 : Persistence file modified

5.2.9 Repositories

Repositories are used to manage entities. That means CRUD operations. The set of repositories manages data locally (with serialized objects) and in the database.

There is only 1 instance of each repository created (when needed). Spring is in charge of sharing repositories when there are multiple requests to the server.

No repository is defined by default. So we have to create repositories by ourselves.

For each repository, we will create 2 interfaces and 1 class. Consider xxx is an entity name, you have to create :

- xxxRepository.java
  This interface is our main repository, the one we will use.
  It pre-defines most of the CRUD methods by extending xxxRepositoryCustom.
  In this file we define only selection requests that produces Collections of items.
- xxxRepositoryCustom.java
  This interface defines the specific CRUD actions we need.
- xxxRepositoryCustomImpl.java
  This class is an implementation of xxxRepositoryCustom.java

This set of 3 files is not mandatory, but it is a good way of structuring repositories. The 2 interface defines accessible methods and the java class defines how we implement specific functions.

According to applicationContext.xml, the files must be located in “fr.centrale.nantes.infosi.repositories”. Or in the package you indicated in applicationContext.xml in tag jpa :repositories. For example, for the entity Book, we create in fr.centrale.nantes.infosi.repositories :

- interface BookRepository.java,
- interface BookRepositoryCustom.java,
- class BookRepositoryCustomImpl.java

Let's start with interface "BookRepository.java".
BookRepository.java is an interface. We have to give informations to the compiler and You can let it nearly empty for the moment. We will fill it on requirements.

We add annotation @Repository before the interface definition to tell it is a repository.
BookRepository extends JpaRepository. This interface (in SPRING library) defines most of the CRUD methods. Spring defines classes that implements JpaRepository.

When you extend JpaRepository, you have to give the Object you manage and its ID Type.
For Books, the JpaRepository is extended with Book and Integer (bookId is an Integer).

Interface BookRepositoryCustom extends BookRepository.

Class BookRepositoryCustomImpl is an implementation of BookRepositoryCustom.
It is a repository too, so we add @Repository before its definition.

[Fig. 65] shows the content of our 3 BookRepository files.

```java
package fr.centrale.nantes.infosi.prwebspring.repositories;

import fr.centrale.nantes.infosi.prwebspring.items.Book;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface BookRepository extends JpaRepository<Book, Integer>, BookRepositoryCustom {

}
```

```java
package fr.centrale.nantes.infosi.prwebspring.repositories;

public interface BookRepositoryCustom {

}
```

```java
package fr.centrale.nantes.infosi.prwebspring.repositories;

import org.springframework.stereotype.Repository;

@Repository
public class BookRepositoryCustomImpl implements BookRepositoryCustom {

}
```

Fig. 65 : Book repository files

if you have a red dot before jpaRepository (or any org.springframework.data class), that means the persistence libraries are not included. Have a look to the pom.xml configuration.

If you need a bit of documentation on JPA, you can find it here :

https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

Do the same for Person and Borrow.

5.2.10 Some rules for repositories

We will use programming rules for the methods in repositories :

- create(...) : creates an object in the persistence unit and in the database
- findXXX : returns a collection of objects
- getXXX : returns a single object
- update(...) or updateXXX(...) : updates informations
- remove(...) : removes an object from the persistence unit and in the database

5.2.11 Try it

Ok. Compile it. Run/debug it and....
it fails !!!! You might have a HTTP 404 error.

But why ?
Message says it can't find route (URL) index.do with method GET.

When application is launched, when you connect, your first page is redirect.jsp (have a look to web.xml). Open redirect.jsp : it redirects to **"index.do"**.

Have a look to the configuration files (applicationContext.xml). We said we use annotations, and that the controllers are in "fr.centrale.nantes.infosi.prwebspring.controllers". To solve route acces (via URL), SPRING tries to find a route "index.do" in a controller in that package.

Is there a controller in "fr.centrale.nantes.infosi.prwebspring.controllers" ? No. So tere is no route to "index.do".
So, it failed.

We need a controller that routes "index.do".

5.2.12 Creating index controller file

Ok, now let's create our first controller.

As the controller name is not important, let's call it IndexController.
Its location is important : controllers are located in package "fr.centrale.nantes.infosi.prwebspring.controllers".

Now, its content. We have to create a java source code.
We add annotation **@Controller** to tell this class is a controller.
IndexController must contain a method that manages route "index.do". That means, the method catches "index.do" and returns the view "index". So, according to the elements in dispatcher-servlet that refers to "WEB-INF/views/index.jsp".

Spring uses the class **ModelAndView** to manage views. So, our method has to return a ModelAndView for route "index.do".

[Fig. 66] shows how we write this controller.

```
package org.centrale.prweb.prwebspring.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class IndexController {

    @RequestMapping(value="index.do")
    public ModelAndView handleIndexGet() {
        return new ModelAndView("index");
    }
}
```

Fig. 66 : First controller

- the controllers are in the specific package we defined in dispatcher-servlet and applicationContext.
- Annotation @Controller tells the class is a controller. The annotation is located just before the class definition.
- Annotation @RequestMapping tells what route we manage.
  Here, we manage route "index.do". We do not give any method for the route (GET, POST), so it will work for both GET and POST. If you want to restrict a route to a method, use method=RequestMethod.xxx
- To manage the view, we use an instance of ModelAndView. When we create it, we tell which view is associated with the object.
- the java method returns the ModelAndView we created.
- the name of the method that manages the route is not important. Use the one you want.

Clean and build your project. Re-run it.
You should have informations displayed. The index file you already displayed.

Questions

What are the main configuration files ? What are they used for ?
What is a persistence file ? Where is it located ?
Where do you find the persistence unit name ?
How do you define a route in a controller ?
What are repositories used for ?
Where are located views ? How are they used by controllers ?

5.2.13 The Login page

We need a Login page, List books, users, edit users, edit books. And borrow them.

Let's start with the Login page. It is the first page we use, so it should replace the current file index.jsp in our views. We will use the HTML pages you wrote for the first practical work.

5.2.13.1 Adding JS, CSS and Images

Our web server root directory is "Web Pages" in Netbeans. You can also access to with the "webapp" directory in 'src/main".

Copy the 4 directories (bootstrap, css, img and js) located in the materials (03_SPRING) to "src/main/webapp" directory. They should appear in your "Web Pages" in Netbeans. [Fig. 67] shows the web page directory in Netbeans.

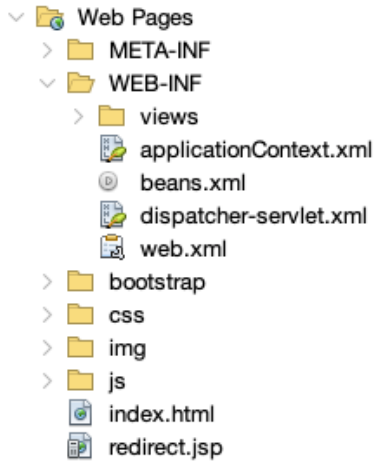


Fig. 67 : Web Pages section in Netbeans

Remember that "Web Pages" (or src/main/webapp in your project directory) is the root of your app. So, the URL to access a file for example in img, is "img/...".

5.2.13.2 Login JSP page

For the JSP Login Page, we will use the one we wrote in the first part of the practical work.
This page must be located in "WEB-INF/views/index.jsp".

Keep the first line of current index.jsp that tells encoding is UTF-8, and replace the file content by yours. Your JSP file should be something like the one in [Fig. 68].

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html lang="fr-fr">
    <head>
        <title>Library Login</title>
        <meta charset="UTF-8"/>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="bootstrap/css/bootstrap.css">
        <script type="text/javascript" src="js/jquery-3.3.1.min.js"></script>
        <script type="text/javascript" src="bootstrap/js/bootstrap.min.js"></script>
    </head>
    <body>
        <div class="py-5">
            <div class="container">
                <div class="row">
                    <div class="col-md-12">
                        <h2 class="">Library Login</h2>
                    </div>
                </div>
                <div class="row">
                    <div class="col-md-12">
                        <form id="c_form-h" action="login" method="POST">
                            <div class="form-group row">
                                <label for="inputlogin" class="col-2 col-form-label">Login</label>
                                <div class="col-10">
                                    <input type="text" class="form-control" id="inputlogin" placeholder="
                                </div>
                            </div>
                            <div class="form-group row">
                                <label for="inputpassword" class="col-2 col-form-label">Password</label>
                                <div class="col-10">
                                    <input type="password" class="form-control" id="inputpassword" placeh
                                </div>
                            </div>
                            <button type="submit" class="btn btn-success">Submit</button>
                        </form>
                    </div>
                </div>
            </div>
        </div>
    </body>
</html>
```

Fig. 68 : Login page script

Use a "Clean and build" on your project and rerun it. The result should be the one in [Fig. 69].

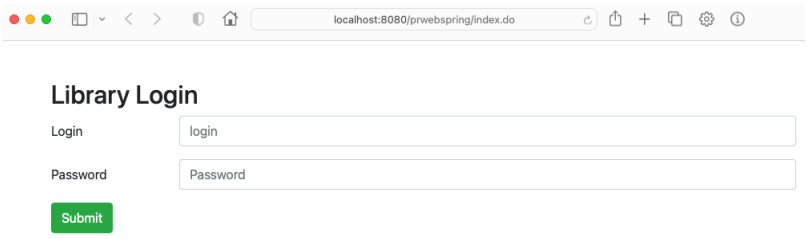


Fig. 69 : login page

5.2.13.3 Responding to Login

When user gives a login and a password, we have to identify user.

In our file index.jsp, the form'action is "login". However, we told in the configurations files that we manage URL ending with ".do". So, we have to change the form'action to "login.do" and manage route "index.do" in POST mode in our controller. So :

- ensure form'action in index.jsp is "login.do" (or any name ending by .do).
- catch route "login.do" (or the action name you used) with POST mode in the controller to check login/password validity.

Remember to use a **unique** method name, even if you can use the name you want.
[Fig. 70] shows what your method can looks like.

```
@RequestMapping(value="login.do", method=RequestMethod.POST)
public ModelAndView handleLoginPost() {
```

Fig. 70 : Method handle login.do route in POST mode

To implement the method, we have to get login and password and call the right view.

There are several ways to do this. You can get them from the request object or as a specific object in the parameters.

Use the one you prefer. Both are ok.

We prefer the first one because it does not require to define an objet to get data. For large apps, it is easier to adjust parameters to the form.

5.2.13.3.1 Get parameters from the request

This is the easiest way to get the informations.

In your request, add a parameter **HttpServletRequest request** and import library from **jakarta.servlet.http.HttpServletRequest**. Parameter request is used to get the request data defined in the form.

You can get request parameters using method **getParameter** that returns the parameter's value as a string and null if it doesn't exists. retrieve "login" and "password" from your form (check their "name" in the inputs in your form).
Then check login and password. If they are correct, use the ModelAndView "users" (means file users.jsp), if not, go back to "index".
Model "users" will be defined later.

[Fig. 71] shows what your method can looks like.

```
@RequestMapping(value = "login.do", method = RequestMethod.POST)
public ModelAndView handleLoginPost(HttpServletRequest request) {
    ModelAndView returned;

    String login = request.getParameter("login");
    String password = request.getParameter("password");

    if ((login != null) && (password != null)
            && (login.equals("admin")) && (password.equals("admin"))) {
        returned = new ModelAndView("users");
    } else {
        returned = new ModelAndView("index");
    }
    return returned;
}
```

Fig. 71 : Get login / password and call right view

5.2.13.3.2 Get parameters from an object

To get parameters we can use an adapted object as a parameter for our method. That can be used **if and only if** the object's setters can set all object's attributes. If there is 1 missing, it will fail.

How can we do that ?

- We create a class MyUser in the same package as the controllers. We have 2 attributes that are the ones we find in the form (login and password).
  Do not forget to create constructors, getters and setters for your parameters.
- We use a MyUser parameter for our method in the controller.

Then we check login and password. If they are correct, we use model "users", if not, we go back to "index". Model "users" will be defined later.

[Fig. 72] shows what you should have.

```
package org.centrale.prweb.prwebspringmaven.controllers;

public class MyUser {
    private String login;
    private String password;
```

```
@RequestMapping(value = "login.do", method = RequestMethod.POST)
public ModelAndView handleLoginPostWithUser(MyUser user) {
    ModelAndView returned;

    String login = user.getLogin();
    String password = user.getPassword();

    if ((login != null) && (password != null)
            && (login.equals("admin")) && (password.equals("admin"))) {
        returned = new ModelAndView("users");
    } else {
        returned = new ModelAndView("index");
    }
    return returned;
}
```

Fig. 72 : Get login / password with an object as a parameter

But we didn't created any MyUser instance. How is it supposed to work ?

When your request is sent to your server, Spring catches it to analyse what it can do with it. There is a route corresponding to the request, with the right method. So, this is what it might call. The associated method has parameters, could we match them (we should say it because we only have one, but it would be the same for several parameters) with the request parameters ? So spring creates a MyUser instance and calls the setters (that is why all that is required) with the corresponding parameters in the request. If there is no parameter with the same name, attribute filling failed.

5.2.13.4 View response

Both methods may lead to use model (view) "users" if it is ok, that means a file "WEB-INF/xxxx/users.jsp". xxx depends on the folder you defined in the dispatcher.

Copy file users.html from your first practical work and paste it in the jsp directory. Rename it as "users.jsp" and add a first line for JSP to tell you use UTF-8, the same as in "index.jsp". We will change the file later. We only want to call this page if login is succesfull.

5.2.13.5 Try it

Stop tomcat server if it is still launched.
Clean and rebuild your project (this is the best way to ensure you use the latest compiled sources).
Re-run it.

Try with "admin"/"admin" and check you have a dummy list of users. Try with something else and check you go back to the login page.

5.2.14 List users page

When we login, we have a fake list of users. We would like to retrieve users from database. We have 2 things to do :

- retrieve data from database
- send data to the view
- use data in the view

The scripts (java and JSP) are a bit further.

5.2.14.1 Get data from database

First, have a look to the JpaRepository javadoc at :

https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

Do you notice method findAll in the doc ?
Maybe we could use our repository to get the list of users from the database.
So, if we have a PersonRepository (have a look to the interface in your project if you don't remember what it is) we could do something like :

```
Collection<Person> myList = personRepository.findAll();
```

5.2.14.2 Send data to a view

The object that represents the view is a ModelAndView. Javadoc is here :

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/ModelAndView.html

Have a look to the method addObject.
It will allow us to add an object to the view.
First parameter is the object's name in the view (can be different from the variable name).
Second parameter is the object we send.

So maybe we could use something like :

```
Collection<Person> myList = ...
returned = new ModelAndView("users");
returned.addObject("usersList", myList);
```

The object myList is the result of the JpaRepository tools, more precisely from method findAll.
Next, we create a ModelAndView based on the view "users" (understand users.jsp).
At last, the object myList is added to the view. It can be used in the view as "usersList".

You can send any object to the view when using **addObject** : a basic object, a list, a hashmap, ... And of course you can send as many objects as you want. you only have to call addObject for each object you want to send to the view.

That means we need an instance of PersonRepository to use findAll and send data to the view.

Problem : **how do we instantiate it ?**

Do we have to create a new .. something ?
We have a PersonRepository, but it is an Interface.
We do not have any implementation for PersonRepository.
And without any implementation of PersonRepository we can't create a new PersonRepository.

Hum... in fact, there is already something implemented by the libraries.
We use Spring, and JPA. These libraries have default implementations to manage many things (why do you think we used JpaRepository when we created the repositories ?).

The annotations (especially "**Autowired**") can tell to Spring and JPA to create default implementations for our interfaces.
When you compile the project, this is one of the task the annotation take into account.

So, we only have to tell SPRING that it will have to use a default implementation for PersonRepository and create an object that will manage our informations.

This object is a bit complex because it has to manage the link with the database, store created object somewhere so that they are not created each time we want them, and it has to share the stored objects between our application users, manage transactions, ...
... This is JPA main role.

The PersonRepository object we create is unique for our application. Multiple users will use the same object. From a theoretical point of view, we use a "Design Pattern" : **Singleton**. There will be only 1 instance of the PersonRepository created and each user of the application uses the same instance. They use the same stored (serialized) objects, ... That will ensure that if you access a stored Person, other users will access the same data.

Do you understand why creating a class that implements PersonRepository is not that easy ?
So, let's SPRING manage everything for us.

Here is the way we create our PersonRepository in our controller :

```
@Autowired
private PersonRepository personRepository;
```

Everything is done by **Autowired**.
Each time we need a repository in a class, we will use that annotation that gives instructions to Spring and JPA.
We defined the object "personRepository" but of course you can use the name you want.

In fact, Autowired is often used in Controllers and Repositories.
If you want to use a repository in a method in a class that is not declared as a Controller or a Repository, you have to send it as a parameter. This is also one of the reasons we tell where are controllers and repositories in the configuration file applicationContext.xml.

Next, we have to send informations to the view.
Views are defined through the object ModelAndView. To send anything, we use method **addObject** that defines the name of the object in the view, and its value.

[Fig. 73] shows what you should have.

```
@Controller
public class IndexController {

    @Autowired
    private PersonRepository personRepository;

    @RequestMapping(value = "index.do")
    public ModelAndView handleIndexGet() {
        return new ModelAndView("index");
    }

    @RequestMapping(value = "login.do", method = RequestMethod.POST)
    public ModelAndView handleLoginPost(HttpServletRequest request) {
        ModelAndView returned;

        String login = request.getParameter("login");
        String password = request.getParameter("password");

        if ((login != null) && (password != null)
                && (login.equals("admin")) && (password.equals("admin"))) {
            returned = new ModelAndView("users");
            Collection<Person> myList = personRepository.findAll();
            returned.addObject("usersList", myList);
        } else {
            returned = new ModelAndView("index");
        }
        return returned;
    }
```

Fig. 73 : Using Autowired and a repository

5.2.14.3 Use data in the view

How could we use data sent to the model in our HTML (hum... JSP) script?

JSP files can include a language to manipulate data : **JSTL** (or **J**ava server page **S**tandard **T**ag **L**ibrary).
So first, ensure JSTL (not the jakarta.servlet.jsp.jstl one) is included in your POM file.
If not, you might have missed something when we configurated the file pom.xml.

Then we have to include jstl namespaces in the JSP file.

Open file “users.jsp” in the views directory (you should already have rename the html file in a jsp file).

There are several parts in the JSTL lib we add to this file.
The one we will use is the "**core**" one.
If you want to internationalise your application, or format data, "fmt" would be nice. We add it in the example but it is not required.

To use the library and associate it with a prefix, we have to add it at the beginning of the jsp file, juste after the UTF-8 declaration.
Usually that means line 2 of the file. [Fig. 74] shows JSTL modules inclusion in a JSP file.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c"   uri="jakarta.tags.core" %>
<%@taglib prefix="fmt" uri="jakarta.tags.fmt" %>
<!DOCTYPE html>
<html lang="fr-fr">
```

Fig. 74 : Add JSTL library modules to JSP file

To use JSTL tags, we prefix them with the prefix "c :" (according to the definition we gave), so if ypu change it (replacing "c" by something else) remember to change the prefix according to what you chose in the tags you will use. For example, if you changed prefix by "abcd", that means your JSTL tags must be prefixed by "abcd :".

Some elements about using JSTL and ModelAndView objects in pages :

- ${ data }
  means you use / evaluate object "data" that was sent (addObject) to the view.
- ${ data.myAttribute }
  means you use method "getMyAttribute()" in object "data" which was sent to the view.
- ${ data.myMethod(...) }
  means you use method "myMethod" in object "data" which was sent to the view.
  myMethod is supposed to return an object.
- <c :if test="..." >... </c :if >allows you to check some data to define what to display. Remember, these are tags, so there is no "else" instruction.
- <c :foreach var="iter" items="${data}" >will loop on iterable data. Iterator variable is "iter". That means, in the loop, you should use ${iter}
- <c :choose ><c :when test="" >...</c :when >... </c :choose >is a switch. Use <c :when test="..." >to define cases and <c :otherwise >for the other cases.
- ...

When we sent the list of users to the view, we called it "usersList". So, to display the users list, we have to use a loop on "usersList" and display fields. Something like [Fig. 75].

```
<tbody>
    <c:forEach var="item" items="${usersList}">
    <tr>
        <td scope="col">${item.personId}</td>
        <td>${item.personFirstname}</td>
        <td>${item.personLastname}</td>
        <td>${item.personBirthdate}</td>
        <td class="text-center">
            <form action="editUser" method="POST">
                <button name="edit" class="btn"><img src="img/edit.png" alt="edit" class="icon" /></b
                <button name="delete" class="btn"><img src="img/delete.png" alt="delete" class="icon"
            </form>
        </td>
    </tr>
    </c:forEach>
</tbody>
```

Fig. 75 : Loop on users in JSP file

5.2.14.4 First try, and a bit of customisation

Ok, try it. The result should be something like in [Fig. 76]. Of course the aspect depends on your css file.

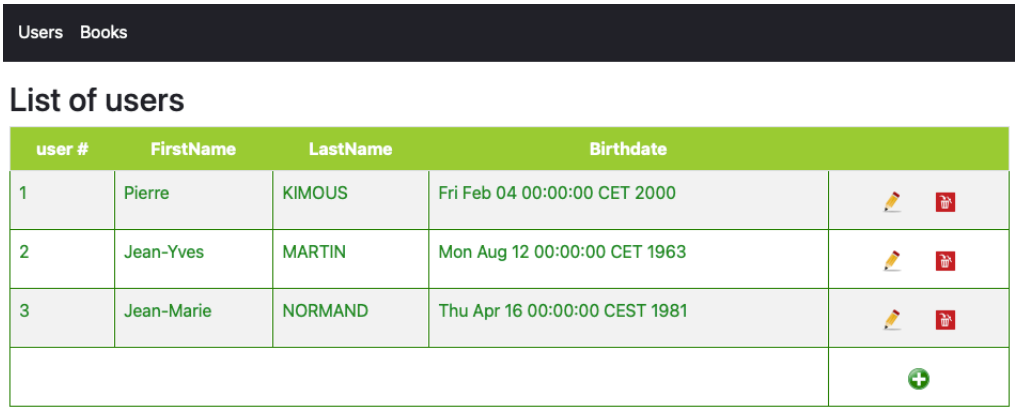


Fig. 76 : list users result

What is this date format? Oh, this is the default method "toString" for Date.
Could we change it for something more convenient?
But, if possible, we do not want to add a method in our java scripts (especially in Person.java).
The problem refers to the view so it should be managed by the view.

We can use JSTL "**fmt**" for that. The one we added but that we didn't use yet. Change your date display by :

```
<td><fmt:formatDate value="${item.personBirthdate}" pattern="yyyy-MM-dd" /></td>
```

Of course, you can change the format pattern according to what you want.
Let's try again. Result is in [Fig. 77]. Ok. Seems better.

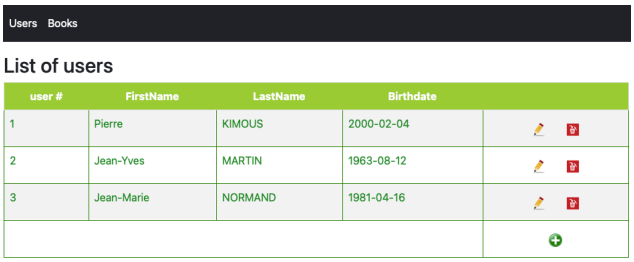


Fig. 77 : list users result with right date format

5.2.15 Manage users

Now let's create a route that display users...
But haven't we done that just before ?
Yes, but let us change this a little bit.

Create a **PersonController** class that will manage users.
Ensure it is defined as a Controller.

Manage route "users.do" in POST mode.
Copy the lines we used to manage users in login.do to define the view to manage.
Of course you have to define a personRepository, as you did in indexController.

[Fig. 78] shows the the management of route "users.do".

```
@RequestMapping(value="users.do", method=RequestMethod.POST)
public ModelAndView handlePostUsers(HttpServletRequest request) {
    Collection<Person> myList = personRepository.findAll();

    ModelAndView returned = new ModelAndView("users");
    returned.addObject("usersList", myList);

    return returned;
}
```

Fig. 78 : list users

5.2.16 Forwarding ?

When login / password is correct in login.do, we display users.
We also define a route to manage users.

Why do we have the same code twice ?
In fact, when login/password is correct in login.do, we should switch to users.do.
And when it is incorrect, we should switch to index.do.

To do this there are 2 methodes : Redirecting and Forwarding.

Redirection is used to tell the browser to do to another URL, on another site.
Forwarding is used to tell the browser to do to another URL on the same site.

In our case, it corresponds to forwarding.
Spring manages that with class **RequestDispatcher**.
Mode is kept according to the request. To forward a request, we also need the HttpServletResponse object sent to the initial request. So, we have to add it to the parameters.

Change the method that manages login.do by the one in [Fig. 79].

```
@RequestMapping(value = "login.do", method = RequestMethod.POST)
public void handlePostIndex(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String login = request.getParameter("login");
    String password = request.getParameter("password");

    if ((login != null) && (login.equals("admin"))
            && (password != null) && (password.equals("admin"))) {
        RequestDispatcher requestDispatcher = request.getRequestDispatcher("users.do");
        requestDispatcher.forward(request, response);
    } else {
        RequestDispatcher requestDispatcher = request.getRequestDispatcher("index.do");
        requestDispatcher.forward(request, response);
    }
}
```

Fig. 79 : Forward to users or index

Method getRequestDispatcher defines the new route to use.
Method forward applies the forwarding to the new route. This method requires the request and the response that were initially used.

Of course you can remove elements in indexController that are no longer required (like repositories).

Clean and build the project. Relaunch it and check forwarding works. You should still have the users page when login/password is ok, but it is displayed through another route.

5.2.17 Sorting lists

Our data comes from the database, but we do not manage display order. The informations is displayed according to the way findAll get informations. So… we don't know. It depends on the way the database collects informations.

Could we manage display order ?

Class **Collections** has a method sort to sort a List according to a comparing method.
And maybe we could define a method in each item class to compare 2 items and use it to sort our lists.

First we create the comparison method. Let's try with class Person. Add the methods in [Fig. 80] at the end class Person.

Method compareTo compares 2 "Person".
Method getComparator returns a comparator method that compares 2 objects.

```
public int compareTo(Object object) {
    if (object == null) {
        return 1;
    } else if (! (object instanceof Person)) {
        return 1;
    }
    Person itemnId = (Person) object;
    if (this.getPersonLastname().toUpperCase().equals(itemnId.getPersonLastname().toUpperCase())) {
        return this.getPersonFirstname().toUpperCase().compareTo(itemnId.getPersonFirstname().toUpperCase());
    } else {
        return this.getPersonLastname().toUpperCase().compareTo(itemnId.getPersonLastname().toUpperCase());
    }
}

public static Comparator<Person> getComparator() {
    return new Comparator<Person>() {
        @Override
        public int compare(Person object1, Person object2) {
            if (object1 == null) {
                return -1;
            } else {
                return object1.compareTo(object2);
            }
        }
    };
}
```

Fig. 80 : Sorting items

We can use the method sort in Collections to sort the list. When we retrieve the "Collection" of "User", we put it in a **List** (required) and use the appropriate method in **Collections** to sort users.

[Fig. 81] show how you can use it.

```
List<Person> myList = new ArrayList<Person>(personRepository.findAll());
Collections.sort(myList, Person.getComparator());
```

Fig. 81 : Sorting items with Collections

5.2.18 Edit user page

Now, let's edit an user.
When we click on the "edit" icon in the users list, we want to edit an user.

5.2.18.1 Modifying users list view

So, first in the file users.jsp, we have to add the user's ID in the form.
We should do it with an **hidden** field tag.

Then we have to select a route to edit user. Form set method to POST, so it is ok.
We only have to add an attribute **formaction** in the button to manage the route.

[Fig. 82] shows the JSP file modification.

```
<td class="text-center">
    <form action="editUser" method="POST">
        <input type="hidden" name="id" value="${item.personId}" />
        <button name="edit" class="btn" formaction="edituser.do"><img src="img/edit.png" alt="edit" cla
        <button name="delete" class="btn"><img src="img/delete.png" alt="delete" class="icon" /></buttor
    </form>
```

Fig. 82 : list users form modification

The hidden input will be sent to the server when form is validated.
The formaction in the button changes the method called by the one in the attribute formaction.

That means that when we click on the edit button, route "edituser.do" is called. The hidden input "id" and the clicked button's name are sent too.

5.2.18.2 Manage edit user route

Next, we have the manage route "edituser.do" in POST mode.

Create a method to handle the route "edituser.do" in POST mode. Of course, we define it in PersonController.

We get the parameter ID from the request.
We have to ask a PersonRepository to get the user (Person) informations.
And at last call a "user" view.

How could we get user informations from the repository ?
Have a look to the JpaRepository javadoc. Search for method getReferenceById (old GetById is deprecated). Using getReferenceById on our repository should give us the Person with the right id.

[Fig. 83] shows how we could do this.

```
private int getIntFromString(String value) {
    int intValue = -1;
    try {
        intValue = Integer.parseInt(value);
    } catch (NumberFormatException ex) {
        Logger.getLogger(PersonController.class.getName()).log(Level.WARNING, null, ex);
    }
    return intValue;
}

@RequestMapping(value = "edituser.do", method = RequestMethod.POST)
public ModelAndView handleEditUserPost(HttpServletRequest request) {
    ModelAndView returned;

    String idStr = request.getParameter("id");
    int id = getIntFromString(idStr);

    if (id > 0) {
        // ID may exist
        Person person = personRepository.getReferenceById(id);
        returned = new ModelAndView("user");
        returned.addObject("user", person);
    } else {
        returned = new ModelAndView("users");
        Collection<Person> myList = personRepository.findAll();
        returned.addObject("usersList", myList);
    }
    return returned;
}
```

Fig. 83 : get ID and call user view

Do you think it would be a nice idea to sort users ?
And what about using a unique method to create the view that display the list of users ?

5.2.18.3 Create user view

Our controller uses view “user” that we have to add.

Copy file user.html from the first practical work and change it to take into account it is a view, add jstl tags and manage the data we have sent to the view. In our controller, data is called “user”.

Maybe it would be a nice idea to add a form that includes the table. Route should be something like “saveuser.do”, in method POST of course. Then we could add an hidden field for the ID in the form, to be able to send it back to the server when we save modifications.

[Fig. 84] shows the form you could have in user.jsp.

```
<form action="saveuser.do" method="POST">
<table class="table table-striped">
    <tbody>
        <tr>
            <th scope="col">user #</th>
            <td>${user.personId}<input type="hidden" name="id" value="${user.personId}" /></td>
        </tr>
        <tr>
            <th scope="col">FirstName</th>
            <td><input type="text" class="form-control" name="FirstName" value="${user.personFirstname}" /></td>
        </tr>
        <tr>
            <th scope="col">LastName</th>
            <td><input type="text" class="form-control" name="LastName" value="${user.personLastname}" /></td>
        </tr>
        <tr>
            <th scope="col">Birthdate</th>
            <td><input type="date" class="form-control" name="Birthdate" value="<fmt:formatDate value="${user.personBirthdate}" pattern=
        </tr>
    </tbody>
    <tfoot>
        <tr>
            <td scope="col" colspan="2" class="text-center"><button type="submit" class="btn btn-block btn-primary">Save</button></td>
        </tr>
    </tfoot>
</table>
</form>
```

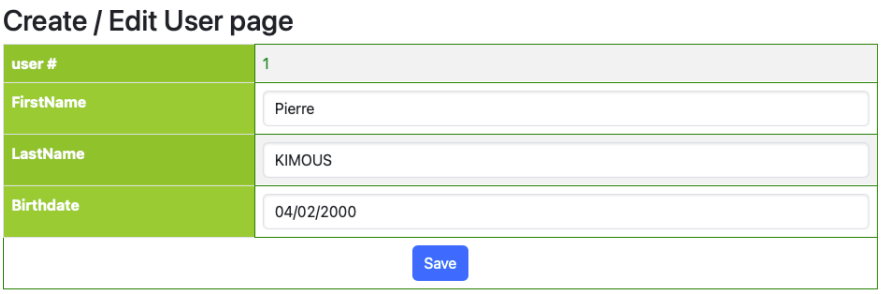


Fig. 84 : edit user form

5.2.18.4 Saving user

Next step, saving user.

We use route “saveuser.do”, method is POST. We can get parameters value from the HttpServletRequest, including the id. So we have to write a new controller (method) for this route. This controller should :

- get parameters (id, firstname, lastname, birthdate)
- get corresponding Person, update fields and save to the database
- al last, ask for the users list and display the “users” view to come back to the list of users.

We know how we can get parameters, but Dates should be a little bit different. We retrieve a string but we should convert it to a Date.

With the person’ ID, we have to get corresponding Person and update this Person.

We know how to get users list and call the “users” view.

First : converting a date as String to a Date format. [Fig. 85] shows a conversion method.

```
private Date getDateFromString(String aDate, String format) {
    Date returnedValue = null;
    try {
        // try to convert
        SimpleDateFormat aFormater = new SimpleDateFormat(format);
        returnedValue = aFormater.parse(aDate);
    } catch (ParseException ex) {
    }

    if (returnedValue != null) {
        Calendar aCalendar = Calendar.getInstance();
        aCalendar.setTime(returnedValue);
    }
    return returnedValue;
}
```

Fig. 85 : Convert STR to Date

You can use the method like this :

```
Date birthdate = getDateFromString(birthdateStr, "yyyy-MM-dd");
```

Now, about modifying user's fields and saving to the database? This kind of thing should not be managed in the controller. Repository should do it.

PersonRepository is the main container and should be used for standart get requests only.
PersonRepositoryCustom is the part of the repository where we can defined personalized methods.
PersonRepositoryCustomImpl implements PersonRepositoryCustom methods.

Let's define something in PersonRepositoryCustom and PersonRepositoryCustomImpl.
We need a method to update user.
In interface PersonRepositoryCustom, add a method "update" like in [Fig. 86]

```
public interface PersonRepositoryCustom {

    /**
     * Update person
     *
     * @param personId
     * @param firstName
     * @param lastName
     * @param birthday
     * @return
     */
    public Person update(Person personId, String firstName, String lastName, Date birthday);
}
```

Fig. 86 : saveuser method in PersonController

Now, we have to implement something to update user in PersonRepositoryCustomImpl.
First we will need a repository instance to work with. Nearly the same kind of declaration than in the controller. For our method, we have to get the Person with the right ID. We use the repository and its method **getByReferenceId** to get it.

So, we have to define a repository with Autowired annotation to create it.
We add annotation Lazy to create it only if required. When you include the library, take care to use the spring one.

If the Person really correspond to its id (it exists in the database), we can change the fields with the Person setters.

If you have a look to the JpaRepository javadoc, you may notice it extends CrudRepository. Have a look to the methods provided by CrudRepository to JpaRepository (still in the JpaRepository javadoc). Do you notice the "**save**" and "**saveAndFlush**" methods? These methods are the ones we need to save data to the database. We only have to save the person to transfer data to the database. Both methods are ok. "SaveAndFlush" do it right now, and "Save" a bit later.

At last, we ensure we have the right data by re-asking it (not required, but it mays have change it someone else is doing the same modification); Then we return the saved Person.

[Fig. 87] shows the "update" method in PersonRepositoryCustomImpl.

```
@Repository
public class PersonRepositoryCustomImpl implements PersonRepositoryCustom {

    @Autowired
    @Lazy
    private PersonRepository personRepository;

    @Override
    public Person update(Person personId, String firstName, String lastName, Date birthday) {
        if (personId != null) {
            // Ensure valididy from database
            personId = personRepository.getReferenceById(personId.getPersonId());
        }
        if ((personId != null)
                && (firstName != null) && (! firstName.isEmpty())
                && (lastName != null)  && (! lastName.isEmpty())
                && (birthday != null)) {
            // Update data
            personId.setPersonFirstname(firstName);
            personId.setPersonLastname(lastName);
            personId.setPersonBirthdate(birthday);
            // Save to database
            personRepository.saveAndFlush(personId);
            // Ensure we have the last version
            personId = personRepository.getReferenceById(personId.getPersonId());
        }
        return personId;
    }
}
```

Fig. 87 : update method in Person repository

Ok. Now we have tools to update user in database.

In our controller, when we manage route “saveuser.do”, we have to get parameters and to ask to the repository to update our database with these values.

[Fig. 88] shows the “saveuser” method in the controller.

```
@RequestMapping(value = "saveuser.do", method = RequestMethod.POST)
public ModelAndView handlePostSaveUser(HttpServletRequest request) {
    ModelAndView returned;

    // Create or update user
    try {
        request.setCharacterEncoding("UTF-8");
    } catch (UnsupportedEncodingException ex) {
        Logger.getLogger(PersonController.class.getName()).log(Level.SEVERE, null, ex);
    }

    String idStr = request.getParameter("id");
    String firstName = request.getParameter("FirstName");
    String lastName = request.getParameter("LastName");
    String birthdateStr = request.getParameter("Birthdate");
    Date birthday = getDateFromString(birthdateStr, "yyyy-MM-dd");

    int id = getIntFromString(idStr);
    Person personId = personRepository.getReferenceById(id);
    personRepository.update(personId, firstName, lastName, birthday);

    // return to list
    returned = new ModelAndView("users");
    Collection<Person> myList = personRepository.findAll();
    returned.addObject("usersList", myList);

    return returned;
}
```

Fig. 88 : saveuser method in PersonController

First part (try-catch) is to manage bad UTF-8 data from some browsers. We will see in a while how to generate it for every method without having to define it each time.

Then we get parameters in the form.
Then we update data.
At last, we go back to the list of users.

Ok, now let’s try.
Clean and rebuild the project.
Run it, or better Debug it (easier to control what happens). Give login/password. Check you have the users list. Click on the Edit button for one of the users. Change something. Save.
Check you go back to the users list and that the modifications are done.

5.2.18.5 Managing UTF-8 Data

We added some instructions at the beginning of the method to manage locally UTF-8 data.

In fact, we can manage this for every method managing a request. For that, we have to change one of the config files : web.xml.

Add the lines in [Fig. 89] at the end of the file web.xml.

```
<!-- Enforce tomcat to UTF-8 encoding forms -->
<filter>
  <filter-name>Set Character Encoding</filter-name>
  <filter-class>org.apache.catalina.filters.SetCharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>ignore</param-name>
    <param-value>false</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>Set Character Encoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>
```

Fig. 89 : UTF-8 management in web.xml

Next, remove the try-catch at the beginning of the function that manages route “saveuser.do”. You should only retrieve the strings and update the person.

Re-compile (suggest Clean and Build) the project and re-run it.

5.2.19 Delete User

To remove a user, we have to click on the trash icon on the user's line in the users list.

According to the JSP script, we have to add a formaction to the trash button and use a new Route. Let's call it "deleteuser.do". As the button is in the form, we already have a hidden tag that gives the user's ID. So nothing more to add.

Let's implement route "deleteuser.do" in PersonController. We use a HttpServletRequest parameter to get the ID (do not use a Person parameter, we do not have enough data to fill the attributes).

We also need a method in the repository to delete a user according to its ID.
Add a method "remove" or "delete" with a parameter "id" in the PersonRepositoryCustom interface. Create an implementation of the method in PersonRepositoryCustomImpl.

The remove/delete method in PersonRepositoryCustomImpl should get the Person from the ID and remove it.
Still have the JpaRepository javadoc ? Have a look to CrudRepository methods. Do you notice method "delete" ?

[Fig. 90] shows how you can delete user from database. Keep in mind you can only remove a Person stored in the database, and you may get the Person through the repository.

```
@Override
public void remove(Person personId) {
    if (personId != null) {
        // Ensure valididy from database
        personId = personRepository.getReferenceById(personId.getPersonId());
    }
    if (personId != null) {
        personRepository.delete(personId);
    }
}
```

Fig. 90 : Delete user code

Of course, this will not work when people already borrowed books. That should lead to a problem because we remove a line in a table that is linked (foreign key) to lines in another table. We will manage that later.

Back to the controller.
Define a method that handles the Route "deleteuser.do" in POST mode. Get id parameter from the request. Remove user using the repository.

Then you have to display the "list users" view.

5.2.20 Create User

Next, we have to create an user.

First add a form around the "+" button in users.jsp (around the button is suffisant, we only have to call the route). Form action is "createuser.do", and method is POST.

In PersonController, create a method to manage route "createuser.do" in POST mode.
Create an empty person and call the user view.
[Fig. 91] shows empty creation and the call to user view.

```
@RequestMapping(value = "createuser.do", method = RequestMethod.POST)
public ModelAndView handlePostCreateUser() {
    ModelAndView returned;

    Person newPerson = new Person();
    returned = new ModelAndView("user");
    returned.addObject("user", newPerson);

    return returned;
}
```

Fig. 91 : Call user view with empty user to create a new one

Ok, now let's manage an empty person in the JSP view. Null attributes are considered as empty attributes. That means that with an empty Person, every field in our JSP file lead to "".

Could we display "NEW" instead of an empty line for the ID ?
[Fig. 92] shows how you can change your JSP file to take empty user into account

```
<td>
    <c:choose>
        <c:when test="${(empty user) || (empty user.personId)}">NEW<input type="hidden" name="id" value="-1" /></c:when>
        <c:otherwise>${user.personId}<input type="hidden" name="id" value="${user.personId}" /></c:otherwise>
    </c:choose>
</td>
```

Fig. 92 : Display NEW when there is no user value

Next, we have to take into account value -1 when we click on "Save".
We call route "saveuser.do" when we click on "Save". That means :

- if ID is negative or null, we have to create new user.
- if ID is positive, we update fields (that is what we currently do).

So we have to add a test for ID, and if it is negative, ask the repository to create a new user.

Next, we need a "create" method in the repository. Add it to PersonRepositoryCustom and implement it in PersonRepositoryCustomImpl. Method should have 3 parameters (firstname, lastname and birthdate) and should return a Person.

How do we implement it ? We have to create a new Person, set fields, save it to the database and get informations from the repository for the created Person in database.

We have 2 ways of getting a Person from the database. getReferenceById is often the most convenient because it will take it from the persistence unit if it exists in. There is another one : findById that will force data coming from database. That can be a little be more heavy, but as we use it when we create the Person, it is ok. Keep getReferenceById when it is possible.

When we use findById we get an Optional{<... >} element. This means the function that should return it, didn't found it. [Fig. 93] shows a way to implement user creation in the repository.

We build person data and fill fields.
Person is saved in database. That generates the personId value.
Then we retrieve saved Person from database.

```
@Override
public Person create(String firstName, String lastName, Date birthday) {
    if ((firstName != null) && (! firstName.isEmpty())
            && (lastName != null)  && (! lastName.isEmpty())
            && (birthday != null)) {
        Person item = new Person();
        item.setPersonFirstname(firstName);
        item.setPersonLastname(lastName);
        item.setPersonBirthdate(birthday);
        personRepository.saveAndFlush(item);

        return personRepository.getReferenceById(item.getPersonId());
    }
    return null;
}
```

Fig. 93 : Create new Person in database

5.2.20.1 Check it

Do not forget to "Clean and Build" your project before you run/debug it. According to annotations, Netbeans may create files to implement interfaces, and sometimes, it is easier to force netbeans to recreate files instead of checking everything is ok.

Run your app. Connect. Create a new user with the icon "+".
Check new user appears in the users list. Also check user is in database.
At last, remove user, and check it is removed from database.

Questions

WHat is a repository used for ?
In a repository, how do you save informations to database ?
Why did we separate the repository files in 3 parts ? What is each part used for ?
Why don't we use JPA methods in controllers ?

5.2.21 Navigating

Now consider the black navigating bar, top of the page.

If you copied it from your previous practical works, you might have links to users.html and books.html. ... but these files do not exist in our project. Our routes look like "xxx.do", so we should use routes like "users.do" and "books.do".

Oh, nice, we already have a route to "users.do".

If you try to define the form around the "ul" tag, you will see it changed the way the classes are used. To avoid these effects, the form has to be around the tag "nav".

Next, calling "users.do" from the navbar. We have 2 ways doing this :

- replace tag "a" by a button and use a formaction to call the route.
- add an "onClick" attribute to tag "a" and call a javascript function that changes the "action" attribute in the form, submit the form and tell the browser the event was performed.

You can use the one you prefer, but the easiest one is the first one.

Replace the tag "a" by a tag "button". Replace attribute "href" by "formaction" and use the right route (users.do, ...). And, to keep the "a" style, add "btn btn-link" to the class list. It will remove the "button" aspect and use a "text" style.

Do the same for books and replace "books.html" by "books.jsp".

Try it. Click on "users". You should launch route to "users.do" and display the users list.

Ok. But the nav bar should be the same for "users" and "books". For that, we could use a **JSP Fragment**. It is a piece of JSP file with the extension "**jspf**". To include it, we use the JSP instruction "**include**".

[Fig. 94] shows how you can define and use a JSP fragment.
The jspf file is "navbar.jspf". It is located in the same directory as the jsp files.

navbar.jspf :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c"   uri="jakarta.tags.core" %>
<%@taglib prefix="fmt" uri="jakarta.tags.fmt" %>

        <form action="#" method="POST">
            <nav class="navbar navbar-expand-md navbar-dark bg-dark">
                <div class="container">
                    <div class="collapse navbar-collapse" id="navbar1">
                        <ul class="navbar-nav ml-auto">
                            <li class="nav-item"> <button class="btn btn-link nav-link text-white" formaction="users.do">Users</button></li>
                            <li class="nav-item"> <button class="btn btn-link nav-link text-white" formaction="books.do">Books</button></li>
                        </ul>
                    </div>
                </div>
            </nav>
        </form>
```

users.jsp :

```
    <script src="js/main.js"></script>
</head>
<body>
    <%@ include file="navbar.jspf" %>
```

Fig. 94 : Using JSP Fragment

5.2.22 Book pages

Ok. Now, let's manage the books.

5.2.22.1 Creating books elements

You have to do exactly the same things than for Person.

- Define methods to create, update and delete Books in BookRepositoryCustom and implement them in BookRepositoryCustomImpl
- Create a controller class BookController. Do not forget the @Controller.
- Implement routes and methods to manage routes (editbook.do, savebook.do, deletebook.do, createbook.do) in BookController.
- Add views to manage books list (books.jsp) and book edition (book.jsp). Ensure the attribute name of your tags correspond to the element you get in your methods.
- Manage book edition, creation and removing n the jsp files.
- Do not forget to use the navbar with the jsp fragment.

5.2.22.2 Try it

Now try it.
Use Clean And Build to rebuild everything.
Run/Debug it.

Log in with correct login/password.
Switch from users to books. Result should be like [Fig. 95].
Edit book. Save.
Create new book.Delete new created book.
Come back to users.

Fig. 95 : Book list

5.2.23 Tools class

One more thing, did you plan to have a method getIntFromString in PersonController and BookController ?

That kind of method does not depend on the controllers. These are tools that do not depend on any object. Maybe it would be a nice idea to collect that kind of methods (this is not the only one), change them to static methods and put them in a specific class that would manage them.

Let's call this class "**Tools.java**" or something like this. Do not forget to make methods public (static of course) so that they can be used.

You can put your class Tools in the package controller or in a dedicated package. If you put it in a dedicated package, you will have to import it in your controllers.

Check it works.

5.2.24 Borrowing a book

Ok, now last step : user borrows a book or returns a book.
We manage books borrowing from the user page.

First, add borrowing informations as a table bellow user informations. You can use what you wrote in the first practical work to add HTML informations to the user JSP file.

What do we have to do ?

- In the file user.jsp create a table for borrowed books to manage user's borrowed books.
- Loop on user's borrowed books to display books.
  You should find them in ${user.borrowCollection}
  If you didn't understand, create a jstl loop on ${user.borrowCollection}
- for each borrowed book, we display borrowed date, book title, and when the book was returned if it was returned, or a button to return it, if not.
  Maybe you should have a look to Borrow.java to be sure of what you do.
  - book borrowed date is borrow.borrowDate
  - book return date is borrow.borrowReturn
  - book title is borrow.book.bookTitle
    Remember that, in JSTL, each time you access a field, you use its getter.
- at last something to borrow a new book.

5.2.24.1 Returning a borrowed book

To return a book we will use a javascript function. This function will do an AJAX call.
[Fig. 96] shows how you can display borrow lines and link with AJAX function.

```
<c:forEach var="borrow" items="${user.borrowCollection}">
    <tr>
        <td><fmt:formatDate value="${borrow.borrowDate}" pattern="yyyy-MM-dd" /></td>
        <td>${ borrow.book.bookTitle }</td>
        <td class="centered">
            <c:choose>
                <c:when test="${not empty borrow.borrowReturn}">
                    ><fmt:formatDate value="${borrow.borrowReturn}" pattern="yyyy-MM-dd" />
                </c:when>
                <c:otherwise>
                    <button class="icon" name="return"
                            onclick="returnBorrow(this, ${ borrow.borrowId }); return false;">
                        <img src="img/return.png" alt="return" class="icon" />
                    </button>
                </c:otherwise>
            </c:choose>
        </td>
    </tr>
</c:forEach>
```

Fig. 96 : Borrow line display

The return button calls a JS AJAX function to return the borrowed book.
As we return false, that means we have to manage data sent back by the AJAX function.

The "returnBorrow" javascript function. This is an AJAX call. We will use a route, like we usually do. Let's put our returnBorrow function in a file "main.js". This file might be put in the js directory. Check the jquery library is in the js directory too.

The "returnBorrow" function has 2 parameters : the button (the "this" parameter), and the borrow ID. We have to put the borrow ID in a javascript object and send it to a route that will manage the book return. When done, we should receive an information that tells it is ok. In that case, we replace the button by the returned date.

[Fig. 97] shows JS script you could write.

```
function returnBorrowSuccess(theResult, buttonRef) {
  if (buttonRef !== null) {
    var refTD = buttonRef.parentElement;
    if (refTD !== null) {
      refTD.removeChild(buttonRef);
      var currentDate = new Date(((Date)(theResult.returnedValue)));
      var currentDateStr = currentDate.toLocaleDateString();
      var text = document.createTextNode(currentDateStr);
      refTD.appendChild(text);
    }
  }
}
function returnBorrow(buttonRef, borrowId) {
  if (borrowId > 0) {
    $.ajax({
      url: "returnborrow.do",
      method: "POST",
      data: {
        "id": borrowId,
      },
      success: function (theResult) {
        returnBorrowSuccess(theResult, buttonRef);
      },
      error: function (theResult, theStatus, theError) {
        console.log("Error : " + theStatus + " - " + theResult);
      }
    });
  }
}
```

Fig. 97 : Return Borrow JS script

Do not forget to include the 2 javascript files (your script and JQuery) in the head part of the file user.jsp or the returnBorrow script will not be called. Note that we use an AJAX call. Of course you can change the route called in the url part depending on the route you define in the controller. And do not forget to use a form over the table (syntaxically, you cannot but it over tfoot, or tr).

We need a route to manage returnborrow.do
We need a Controller (BorrowController) and the Repository (the 3 files).

Manage route "returnborrow.do" with method POST.
We have to get the ID, tell we return the book (borrow elements knows who and which book) and

return to the AJAX call a JSON object, something that tells it is ok. For that, we need the JSON library, so you should ensure it is in the Dependencies. If you haven't, look for "org.json" and add the latest one.

How could we use JSON library? This library let us use 2 kind of objects : JSONObject and JSONArray. We only need the first one. We only have to create a JSON object and send it to an ajax view that will manage it.

[Fig. 98] shows how we could use the JSON object. If there is a problem, we return an error status so that it could be used by the AJAX caller.

```
@RequestMapping(value = "returnBorrow.do", method = RequestMethod.POST)
public ModelAndView handleReturn(HttpServletRequest request) {
    ModelAndView returned = new ModelAndView("ajax");
    JSONObject returnedObject = new JSONObject();

    String borrowStr = request.getParameter("id");
    int borrowId = getIntFromString(borrowStr);

    Borrow borrow = borrowRepository.returnBook(borrowId);
    if (borrow != null) {
        returnedObject.append("id", borrow.getBorrowId());
    } else {
        returned.setStatus(HttpStatus.BAD_REQUEST);
    }
    returned.addObject("theResponse", returnedObject.toString());

    return returned;
}
```

Fig. 98 : Controller manages borrow return call

What about the ajax view? Quite simple : we added an object "theResponse" to the view. We only have to display it. [Fig. 99] shows what an ajax view could look like.

```
<%@page contentType="application/json" pageEncoding="UTF-8"%>
${theResponse}
```

Fig. 99 : AJAX view

Using a BorrowRepository, we get the borrow object. We only have to set the borrowReturn information to the current date.

That means we have to define a method returnBook in BorrowRepositoryCustom and implement it in BorrowRepositoryCustomImpl. That method should have a Borrow and a Date as parameters and update the borrow information. Do not forget to save the modified Borrow in the database. It returns the modified Borrow.

You should also create a returnBook method in the repository with only a "Borrow" Parameter. This method should update the borrow information with the current date. You should also do it with the

borrow ID as a parameter. That means retrieving the Borrow element and update it.
[Fig. 100] shows how we could implement the returnBorrow methods in the repository.

```
@Override
public Borrow returnBook(Borrow item, Date date) {
    if (item != null) {
        item = borrowRepository.getReferenceById(item.getBorrowId());
    }
    if ((item != null) && (date != null)) {
        item.setBorrowReturn(date);
        borrowRepository.saveAndFlush(item);
        return item;
    }
    return null;
}

@Override
public Borrow returnBook(Borrow item) {
    Calendar aCalendar = Calendar.getInstance();
    Date date = aCalendar.getTime();
    return returnBook(item, date);
}

@Override
public Borrow returnBook(int borrowId) {
    if (borrowId > 0) {
        Borrow item = borrowRepository.getReferenceById(borrowId);
        return returnBook(item);
    }
    return null;
}
```

Fig. 100 : Repository script to return a book

5.2.24.2 Borrowing a book

Now, we need something to borrow (add) a book.
We manage this with the book list and the button "+" located bottom of the table. [Fig. 101] shows how we could manage that.

```
<tfoot>
  <tr>
    <td colspan="2">
      <input type="hidden" name="userID" value="${ user.personId }" />
      <select name="bookID" class="form-control form-select form-select-lg mb-3">
        <option value="-1" selected="selected">-</option>
        <c:forEach var="book" items="${booksList}">
          <option value="${ book.bookId }">${ book.bookTitle }</option>
        </c:forEach>
      </select>
    </td>
    <td  class="text-center">
      <button class="btn" formaction="addborrow.do"><img src="img/plus.png" alt="add" class="icon" /></button>
    </td>
  </tr>
</tfoot>
```

Fig. 101 : Add borrow line display

Next, we have to borrow a new book in Borrow controller.
Have a look to the form in [Fig. 101]. We have the user's ID and the SELECT gives us the book's ID.
So, in the controller, we have to get the informations from the HttpServletRequest.
[Fig. 102] shows how you could write it

```
@RequestMapping(value = "addborrow.do", method = RequestMethod.POST)
public ModelAndView handleAddBorrow(HttpServletRequest request) {
    String userStr = request.getParameter("userID");
    int userID = getIntFromString(userStr);
    Person user = personRepository.getReferenceById(userID);

    String bookStr = request.getParameter("bookID");
    int bookID = getIntFromString(bookStr);
    Book book = bookRepository.getReferenceById(bookID);

    borrowRepository.create(user, book);
    // Refresh user data (bookCollection)
    user = personRepository.getReferenceById(userID);

    ModelAndView returned = new ModelAndView("user");
    returned.addObject("user", user);
    returned.addObject("booksList", bookRepository.findAll());

    return returned;
}
```

Fig. 102 : Add Borrow book route

Why do we reload user?

In fact, "user" may have change in the repository (because we added a book to the bookCollection).
So, the user we have before we call the function and the user after the call are a little bit different.
More, what happens if someone changed something while you added a borrowed book?
So, the best way to be sure of the user's value is to reload it from database.

Next, the repository.

Define a new method "create" in BorrowRepositoryCustom and implement it.
When you create a new "Borrow" item, you link an user and a book.
So do not forget ta add the "Borrow" item to the appropriate collection in "user" and in "book".

Oh, one more : never trust what is sent as a parameter.
You don't know when the book and the user where retrieved from database. The ony thing that is sure is their ID.
So, the best way to avoid problems is to reload data from their ID, and manage data with the latest value.

[Fig. 103] shows how you could write it

```
@Override
public Borrow create(Person person, Book book, Date borrowDate) {
    // Ensure we have full data
    if (person != null) {
        person = personRepository.getReferenceById(person.getPersonId());
    }
    if (book != null) {
        book = bookRepository.getReferenceById(book.getBookId());
    }

    // Build new borrow
    if ((person != null) && (book != null) && (borrowDate != null)) {
        Borrow item = new Borrow();
        item.setBorrowDate(borrowDate);
        item.setPersonId(person);
        item.setBookId(book);
        item.setBorrowDate(borrowDate);
        borrowRepository.saveAndFlush(item);

        Optional<Borrow> result = borrowRepository.findById(item.getBorrowId());
        if (result.isPresent()) {
            item = result.get();

            // Set reverse fields
            person.getBorrowCollection().add(item);
            personRepository.saveAndFlush(person);
            book.getBorrowCollection().add(item);
            bookRepository.saveAndFlush(book);

            // return item
            return item;
        }
    }
    return null;
}
```

Fig. 103 : Create a Borrow information in repository

Then we display user once more in the “user” view.

One more thing : to display the user view, we need the list of books. So each time you use “user” view, you should get the book list and add it to the view.

5.2.25 Final try

Ok, let’s “clean and build” the project.
Run/Debug it.

You should connect, switch from users’list to books’list and return.
You should edit a book. Create a new book.

You should edit an user. The you may borrow a book. then return one.

Questions

What is JSTL ?
How do you send data to a view ?
In a controller, how do you get informations from a form ?
How do you associate a route to a controller ?
How do you define a class as a Controller ? a Repository ?
How does SPRING knows in which packages are controllers and repositories ?
What is the difference between SQL and JPQL ?
What is @Autowired used for ?

5.2.26 Summary

Here is a summary of the main actions we used for the project

Create a Web Application (with Maven) in Netbeans
Check project properties, add Spring Web library
Ensure valid POM informations in pom.xml. Suggestion : keep a valid pom.xml file somewhere.
Create project infrastructure (controller, items, repository)
Update applicationContext.xml and servlet-dispatcher.xml
Ensure web.xml is valid
Add database connection and create entities
Ensure persistence.xml validity. Check name according to applicationContext.xml
Create repositories main infrastructure (Repository, CurstomRepository, CustomRepositoryImpl). Usually, you will need a set of 3 files per item.
Copy bootstrap, js, img, css directories to Webapp
Create controllers and views each time you need one. Add methods to repositories when you needed.
Keep an AJAX view somewhere to be able to reply to AJAX calls

Main things to take care of :

- JDK version and Tomcat version (Tomcat 9 and newer versions do not use the same libraries)
- Compatibility between JDK version, Spring version, and libraries versions you use (in pom.xml). We **strongly** suggest to keep a valid version somewhere and reuse it in other applications.
- Check ApplicationContext.xml, DispatcherServlet.xml and web.xml
- never trust parameters in repository, reload them if possible before using them.
- never trust items value after repository call, because it may have change.

6 Front And Back office with Javascript

For this part, we need to work a different way. We use 2 tools :

- Back Office : the server part, linked to the database server or other tools. It receive requests and send responses back.
- Front Office : the part of the application that manages user's interactions and sends requests to the Back Office.

A Back Office developper develops the Back Office part (managing informations, database, ...).
A Front Office developper develops the Front Office part (managing presentation HTML, CSS, JS, calling Back Office, ...).
A Full Stack developper develops both parts.

For this part, you will act as Full Stack Developpers.

The Back Office part is based on **NodeJS** (and npm) because it is a Javascript server.
When we write these lines current version is 24.4, but a more recent version may be available.

Database server is a PostgreSQL server.

For the Front Office part, you have to choose between Angular and React.

6.1 Why a JS framework ?

- We use the same language in the FrontEnd and in the BackEnd
- It can be deployed on most environments with a single development

React migrated to a framework in 2025, whereas Angular is still a TS library. However, as most react app are developed with old version, we keep the framework use.

If you want to use last version for REACT, use https ://react.dev/blog/2025/02/14/sunsetting-create-react-app.

TS means TypeScript. Typescript is very close to Javascript, so it will be available on most of the browsers.
Typescript is developped by Google.

Take care, **there will be 2 servers running for each application** :

- 1 server for BackEnd (NodeJS for us)
- 1 server for FrontEnd (Angular or React)

These servers are running on different ports and can be placed on different computers.
You have to ensure that the connected ports on the computers are open for the 2 servers, not only the FrontEnd part.

Yes, that means the BackEnd could be called by any browser, anywhere.

6.2 Requested tools

You will need :

- NodeJS and npm, with module pg. NodeJS. If you use React you might use version 19+. For angular version 18-.
- a text editor to create pages. We suggest "Visual Studio Code" that is more convenient for that kind of project. Drag and drop the project in VSC window when it will be created.
- PostgreSQL server
- React / Angular tools depending on your choice.

Have a look to the annexes to install nodeJS (and npm) and the pg module.

If you already installed NodeJS and NPM, check versions (see annexes). [Fig. 104] shows the result.

Fig. 104 : Check nodeJS and npm

If you want to manage several NodeJS versions, you can use NVM.

Install nvm :

- windows : https ://github.com/coreybutler/nvm-windows/releases
- macOS : brew install nvm

Here are some commands :

```
nvm install vernionNB
nvm list
nvm use versionNB
nvm alias default vernionNB
```

We use **version 24.4** but you can use any version >= 20.

6.3 React and NodeJS

React is a **typescript library**, not a framework.

We use React for the FrontEnd part and NodeJS for the BackEnd part. That means NodeJS ensures the link with the database, whereas React manages the browser part.

To understand what happens in React, you may consider using "console.log". When you use it, it will tell you which line is used, and you will get variables values and types.

Next, consider the console in your browser, very often, it displays error messages and informations to solve them.

6.3.1 Create React app

6.3.1.1 Create project

First we have to create the react project.

Use your Terminal / Command tool. Go to the parent directory of what you want to be your project location, the directory you want to put your application in.

We use npm to create the project, let's call it "prwebreact". You can change your project name, but avoid space characters, and use only lowercase characters. The npx instruction creates a directory for the project and downloads react infrastructure. "create-react-app" is a template to create React applications.

```
npx create-react-app prwebreact
```

[Fig. 105] and [Fig. 106] show the result of the command.

Fig. 105 : Create REACT project with npm / npx

At the end of the react installation, you may have main instructions. [Fig. 106] shows the end of the react project creation.

```
Success! Created prwebreact at                                          /prwebreact
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd prwebreact
  npm start
```

Fig. 106 : Create REACT project with npm - end

[Fig. 107] shows the prwebreact directory content.

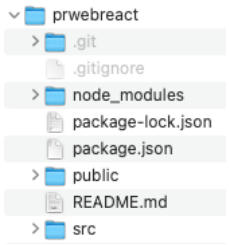


Fig. 107 : REACT project directory

What do we find ?

- you may note GIT files / directory.
- README.md, your project "README". Should be downloaded on GitHub if you use it.
- node_modules (created after the first launch) contains nodeJS modules that will be used for the project. This directory can be removed and rebuilt using the "npm update" command. Modules are uploaded according to json files.
- package-lock.json contains project and node_modules informations
- package.json contains project informations (name, modules used, ...)
- **public** is the root of our web site
- **src** is our React application

6.3.1.2 How does it work ?

Our application is made of 2 parts :

- BackEnd (coming soon).
  - It is managed by nodeJS. It is our "server" part.
  - **It runs in its own terminal / command tool**.
  - It deals with the databases, manage server data. It receives request and send responses back.
  - If you are not pleased with nodeJS, you can change it by any other kind of backend server. It can be, for example, a php set of scripts, a symfony application or a spring boot application.
- FrontEnd.
  - It is managed by React and is the "client" part.
  - This is the visible part of the project, what is displayed in the browser.
  - This part runs on any computer that connects to the application.
  - When your browser connects to the web application, it downloads the JS files of your react application and launches it **locally** (on your computer).

Never forget that, to run the application, technically you should use a computer for each element (BackEnd server, FrontEnd server, Database server). And 1 more for the browser that connects and runs a part of the react application. In our case, we run the 4 elements on the same computer.

6.3.1.3 React application structure

Have a look to the **"src" directory** in your project.

"index.js" is the entry point for the application. React is a "Single Page" Application. So, there is only 1 entry point, even if we will manage several routes in the application.

Now, the content of the file "index.js". It imports React and launches the first component : "App". Look at the way a component (a tag) is used. We will often do something similar.

Now, the components. They are located in the "App" directory. Have a look to the component App, "App.js". The App function returns... HTML code.

Also do you notice the line at the end of the file "export default App ;" ? **Never remove this line**. This exports the information so that "index.js" may know it.

Currently it is written as a function, we will change this soon.

Now, have a look to the "public" directory in your project.

Open file “index.html” with a text editor. It is the structure of **every page** of our application. Do you notice the id of the tag “div” in the body section ? .

Now, open file “index.js” in the directory “src”. This file is imported in the file “index.html”. Do you notice line 7 the id of the element ? Now, you can link it with “index.html”.

Still in “index.js”, line 8, some words about render. You will find this function very often. It is the function that explains what is displayed by the component. So, this is one of the more important function in a component. Line 10, here is the component called : App, in “App.js”.

6.3.1.4 First try

Ok, let’s check everything is ok. Use your terminal / command tool. Go to the root of your react project (might be prwebreact). Run the server with the npm command :

```
npm start
```

Project compiles. Then it launches your browser with this URL : http ://localhost :3000
By default, React apps runs on port 3000 but you can change it in file package.json. [Fig. 108] shows how to change it by port 8080. **DO NOT CHANGE IT, it is only an information**.

```
"scripts": {
  "start": "PORT=8080 react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

Fig. 108 : How to change REACT port in package.json

In your terminal, you should also have informations about the compilation results and how to connect. You can stop the server using CTRL-C. [Fig. 109] shows browser page you may have.

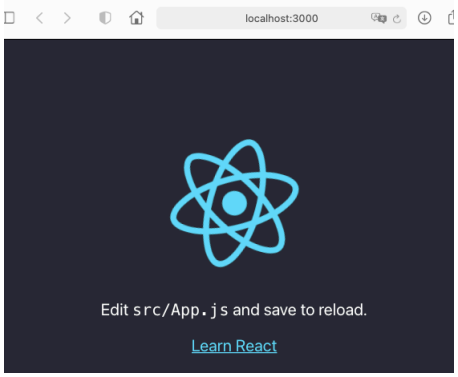


Fig. 109 : REACT project in browser

6.3.2 A bit of explanation

6.3.2.1 React

React is a **"Single Page Application"** library.
That means everything will be managed by a single html file. That page contains the structure of every page in our application.

One suggestion : when using your browser, always keep somewhere the browser console. You will probably have some messages that tells you to change something in your scripts. React is a bit more strict than HTML about tags and attributes writing.

6.3.2.2 Components

React uses Components.
These components are functions, or JS classes that implement React Component class.
Consider components as displayable elements in the page. A component can lead to a basic HTML structure or a set of components. Each time you want to structure elements in the page, that should lead to a component.

Some elements you may understand/know about Components :

- **State**, understand this as "local data", the informations you use in the component. They are usually implemented as a set of JS variables.
  "state" is a default object, usually populated in the class constructor. In the methods, you can access state variables as read-only variables with 'this.state.**state's name**'. To change a state variable value, you will have to use **"setState(state's name, new value)"**.
- **Behavior** : what the component is supposed to do. What can we ask it to do. It is a set of functions, methods, that we will use in the component.
  A component may contain behavior methods, and must include a "render" method that returns how to display the component.
- **render**, a mandatory method, that tells how to display the component in the page.
  "render" may return :
  - another component, on which will be called the render method.
  - an HTML script that displays the component. In that case, the returned HTML script **must only contain 1 tag**.
    For example it can be a div, a table, a form, or anything like that.
    As you cannot display more than 1 element, you cannot render a h1 element followed by a table, but you can render a div that contains the 2 elements.

- **Props**, (for properties) are parameters sent to the component by another component.
  If parent send a value to a component (let's call the value x), the component can acces the value with "this.props.**property's name**" " (this.props.x in our example).

These aspects are supposed to work together so that the component is ok. If your component has only the render aspect, you can write it as a JS function, but very often, you will have several aspects, therefore it is often more convenient to write it as a class. In that case, a component is a JS class that extends "React.component".

6.3.2.3 JSX

JSX is a language developed by Facebook (the main author of React) that comes over Javascript to manipulate informations as you would do with standard HTML Tags.

As JSX is not directly usable in javascript, we have to compile the js files that contains JSX elements to produce HTML and JS informations. React uses **"Babel"** as a compiler for the JSX scripts to convert them in standart JS elements, tags, ...

When you run the project, it first compiles it. This compilation is the traduction phase. That is why, if you have a look to the HTML content in your browser, you will not recognize the index.js file nor the App.js files. Maybe you can recognize part of the "index.html" file located in "public".

The compilation phase may also have an impact on the JS class files. For example the use of "this" is something you might take care of. Sometimes, "this" is not the class instance but... something else. So, we will have to add some instructions to manage that.

6.3.2.4 React and the DOM

React has its own way of representing data, that it maps to the DOM. When you run the application, JSX code is translated to JS and React manages its data and the DOM to display the application.

So, never manipulate directly the DOM. Each time you have to do something, use React functions to do it, or React will not be aware of what you did.

Questions

What is the render function used for?
Why is it impossible to display a single text in a render function?
What are props used for?
What are states used for?
How can you initialize states?

6.3.2.5 Manipulating components

Have a look to “App.js”. [Fig. 110] shows the current content.

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Fig. 110 : REACT default App.js

Currently, App is implemented as a function that returns a div tag.

Let’s change it and define it as a JS class and be a React component.

- Add a first line in the file, import React module Component.
  `import React, { Component } from 'react' ;`
- switch function App definition to a class definition that extends Component.
  `class App extends Component {`
- define a “render()” method that returns the elements to display (required for a component) That consists in defining a method render() that returns... what was returning the initial App function : HTML code. This method name is mandatory for every component. It explains how to display it.

Do you notice in the tags that the HTML “class” attribute is replaced by “className” ? The reason is that “class” is already used to define a class in JS.

DO NOT REMOVE THE LAST LINE that exports App. If you remove it, it will not be usable in “index.js”

[Fig. 111] shows the modifications.

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <p>
            Edit <code>src/App.js</code> and save to reload.
          </p>
          <a
            className="App-link"
            href="https://reactjs.org"
            target="_blank"
            rel="noopener noreferrer"
          >
            Learn React
          </a>
        </header>
      </div>
    );
  }
}

export default App;
```

Fig. 111 : REACT using components for App.js

Start your server with “npm start”. Or juste save the file if it is already started.

Do you notice any difference ? No ? So it’s ok.
We only wanted to use a class component instead of a function.

6.3.2.6 File index.html

The root directory of the web site is “**public**”. In this directory, there is a file “**index.html**” that is the main structure of our application. The entry point.

Of course you can change elements in this file.

But there are also several element you must not change :

- do not remove/change the div with id “root”. This is the root of the application itself, and it is used by React to build the elements
- do not remove/change the link to the manifest (manifest.json).

You can remove unused elements (description, comments, ...) if you want to.

6.3.2.7 Using CSS, Images, ...

There are 2 ways of using static elements in React :

- using React instructions,
- modifying the main HTML file, "index.html" to include them.

As index.html is the entry point, the reference path for css, js, ... is directory public.

6.3.2.7.1 Using React instructions

Let's have a look to the App.js script.

React components are located in "src" and imported as "./" and the file that contains the component. If needed you can use directories in src and import files according to their location.

Have a look to the css import. In that case, "./" means the "src" directory too, because scripts are compiled, and for the compiler, files are in "src". As css imports are translated to a style tag in the file, it will take data in "src".

This is different for images. They cannot be included in the head tag. So, the compiler just keep in mind a relation between a name and a path to a file. Therefore, using "./" for images means "public" directory, that is the root of the web site. If you have a look to App.js, you will notice the way we can use an image : we import a file as "logo", then we can use it in the script as {logo}.

6.3.2.7.2 ADD HTML tags

As the file index.html in the public directory is the main structure of the page, you can include common css files in this page. They will be loaded each time you need it.

So, in the head tag, you can add css imports. In that case, the files are supposed to be located root of the web server, so in "public".

And of course, you can use tags that refers to files in the render functions. As they are standart tags, they are loaded by a request to the web server. As the root directory is "public", you might put the files in "public".

6.3.2.7.3 Our index.html

Copy bootstrap, css, img and js directories to the "public" directory of your project.

Add lines import, just after the "title" tag. [Fig. 112] shows the modifications.

```
<title>React App</title>
<link rel="stylesheet" href="bootstrap/css/bootstrap.css">
<script type="text/javascript" src="js/jquery-3.7.1.min.js"></script>
<script type="text/javascript" src="bootstrap/js/bootstrap.min.js"></script>
<link href="css/main.css" type="text/css" rel="stylesheet" />
```

Fig. 112 : REACT import elements in index.html

6.3.2.8 Routing

React is a single page application. But we have to manage several pages. How could we do ?

React includes a routing system, but we have to install it.

```
npm install react-router-dom
```

Maybe you will have some identified vulnerabilities that you could fix. **DO NOT APPLY THE FIX INSTRUCTION**. Currently the instruction changes the "react-scripts" module to "0.0.0" and you will not be able to use commands like "npm start".

The "react-router-dom" adds some libraries to manage routes.
In the scripts, we will use specific tags like "BrowserRouter", "Switch", "Route", "Link", "Redirect" ... to manage them.

6.3.3 Our first page

React works with components.
A component is a JS file, with a specific structure, that must contain a "render" function that returns what to display.

Currently, our application (App.js) displays the react logo and a link to learn react. We would like it to render our login page. We will use the page developed in the first part of the practical works.

6.3.3.1 Login component

First, let's add some elements for the routing.

We have to change component "App" to explain it has to route to our components according to given routes.
We have to import tags from "react-router-dom". We will use tag "Routes" to tell we define routes.
Then, we use a Route tag for each route to define, and we link them to the component to use.

[Fig. 113] shows the modification for "App.js".
Only 1 route right now : Login, that we will define soon.

```
import React, { Component } from 'react';
import {BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import './App.css';
import Login from "./Login"

class App extends Component {
  render() {
  return (
    <div className="App">
      <Router>
        <Routes>
          <Route exact path='/' element={<Login />} />
        </Routes>
      </Router>
    </div>
  );
  }
}

export default App;
```

Fig. 113 : Effective routing in App.js

In [Fig. 113] we tell we import Login from "./Login".
We also tell route "/" is manage by this component. That means we have to define a file "Login.js" that manages the Login component.

Create a file "Login.js" (maybe you can duplicate App.js) that manages our Login Page.
For the HTML code, we will use the file "login.html" from the first practical work.

We have to replace the render method of the Login component by the login page. Remember that you must change "class" to "className" in the tags. You **MUST NOT USE CLASS AS A TAG ATTRIBUTE**.
Oh, one more thing : forget the "style" attribute, it would not compile.

We also have to import the css file, the js files, images, ... You have 2 solutions :

- "import" the file reference in the file Login.js (like App.css did). Files are compiled by Babel. So imports are translated as a style tag. The css file location is therefore "src".
- include files in "index.html", as a usual inclusions of css or js files. In that case, the file must be located in "public".

We used the 2nd solution previously and will go on with it.

[Fig. 114] shows a possible login page in "Login.js".
You may note that we replaces "class" by "className", and "for" by "htmlFor". When you launch your file, have a look to the console in your browser. It often gives you a clue to solve the problem.

```
import React, { Component } from 'react';

class Login extends Component {
  render() {
    return (
        <div className="py-5">
            <div className="container">
                <div className="row">
                    <div className="col-md-12">
                        <h2 className="">Library Login</h2>
                    </div>
                </div>
                <div className="row">
                    <div className="col-md-12">
                        <form id="c_form-h" action="login" method="POST">
                            <div className="form-group row">
                                <label htmlFor="inputlogin" className="col-2 col-form-label">Login</label>
                                <div className="col-10">
                                    <input type="text" className="form-control" id="inputlogin" placeholder="login" name="login" required="required" />
                                </div>
                            </div>
                            <div className="form-group row">
                                <label htmlFor="inputpassword" className="col-2 col-form-label">Password</label>
                                <div className="col-10">
                                    <input type="password" className="form-control" id="inputpassword" placeholder="Password" name="password" required="required" />
                                </div>
                            </div>
                            <button type="submit" className="btn btn-success">Submit</button>
                        </form>
                    </div>
                </div>
            </div>
        </div>
    );
  }
}

export default Login;
```

Fig. 114 : REACT component Login

If you stopped you server, you might relaunch it.

It is possible you have a error, in node_modules.
In that case, remove directory node_modules and launch "npm install". You have to be located in your project directory or it will fail. It uses package.json to rebuild the directory.

If your node version is >= 16, and you have an error "digital envelope routines :unsupported", open file "package.json" and change the line

```
"scripts": {
  "start": "react-scripts start",
```

by

```
"scripts": {
  "start": "react-scripts --openssl-legacy-provider start",
```

Relaunch server.

In your web browser you should have the login page, like in [Fig. 115].

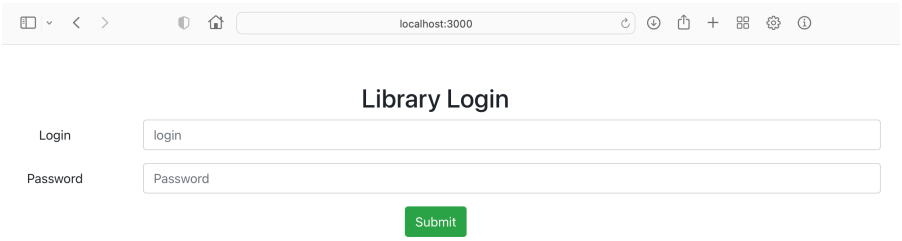


Fig. 115 : Login page displayed

But... what ? Why is the text centered ?
The problem comes from **App.css**.
Remove the text-centrering property of '.App". That should solve the problem.

6.3.3.2 Managing fields

Ok. Our form has fields. We would like our component to manage the fields.

For that, REACT offers tools to synchronise fields content and REACT state variables.

So, for that we will :

- create states for our fields. login for the login, pass for the password
- initialise states variable when the component is used, let's say in the constructor ?
  NB : the states initialisation will create the fields in "state".
- create 1 state to tell we logged in or not. Let's call it canLogin.
  NB : we initialise it with the other states.
- define behavior methods to manage the link between tags (in render) and states
  - we define a handle fonction for each field
  - when field changes, we call its handle function (using onChange)
- manage submit event and check login / password (using onSubmit)
  When form is submitted, we call a checking function, that changes the canLogin state.

That means, in Login.js :

- In the constructor
  - we define states values
  - we define handle functions binders
- for each state, we define its handle function, including one for canLogin
- in the render function
  - when text fields (login, password) change, we launch the handle function
    We also set values to the corresponding state value.
  - when form is submitted, we launch the canLogin handler.

We use a call to “event.preventDefault()” when we check login to avoid default management of the submit event.

[Fig. 116] shows a way to modify Login.js.

```
constructor(props) {
    super(props);

    this.state = {canLogin: false, login: "", pass: ""};

    this.handleChangeLogin = this.handleChangeLogin.bind(this);
    this.handleChangePass = this.handleChangePass.bind(this);
    this.checkLogin = this.checkLogin.bind(this);
}

// Tools to synchronize fields and variables
handleChangeLogin(event) {
    this.setState({login: event.target.value});
}

handleChangePass(event) {
    this.setState({pass: event.target.value});
}

checkLogin(event) {
    event.preventDefault();
    if ((this.state.login === "admin") && (this.state.pass === "admin")) {
        this.setState({canLogin: true});
    }
}
```

...

```
<form id="c_form-h" onSubmit={this.checkLogin}>
    <div className="form-group row">
        <label htmlFor="inputlogin" className="col-2 col-form-label">Login</label>
        <div className="col-10">
            <input type="text" onChange={this.handleChangeLogin} value={this.state.login} className="form-control" id="inputlogin" placehol
        </div>
    </div>
    <div className="form-group row">
        <label htmlFor="inputpassword" className="col-2 col-form-label">Password</label>
        <div className="col-10">
            <input type="password" onChange={this.handleChangePass} value={this.state.pass} className="form-control" id="inputpassword" pla
        </div>
    </div>
    <button type="submit" className="btn btn-success">Submit</button>
</form>
```

Fig. 116 : Login page displayed

In the render method, note the way we modify the fields (we showed it for login, the one is up to you). We link the field to the corresponding behavior method (onChange=...) and we define the associated value in the field (value=...).
Also, we added a "onSubmit" attribute to the form to tell that, when user submits the form, we check login and password.

Take care to uppercase and lowercase characters : compiler will not manage tag's attributes if you miswrite them.

6.3.4 Users component

6.3.4.1 Users component creation

Create file Users.js that lists users. For the moment our users' list will be a fake one. You can use the one you wrote in the first practical work.

One thing you have to take care of is that the function render **can only render 1 tag**. That means the render result cannot be composed of several tags. In that case, what you should do is include the result in a div tag.

[Fig. 117] shows the component Users.

```
import React, { Component } from 'react';

class Users extends Component {
    render() {
        return (
<div>
    <nav className="navbar navbar-expand-md navbar-dark bg-dark">
        <div className="container">
            <div className="collapse navbar-collapse" id="navbar1">
                <ul className="navbar-nav ml-auto">
                    <li className="nav-item"> <a className="nav-link text-white" href="users.html">Users</a></li>
                    <li className="nav-item"> <a className="nav-link text-white" href="books.html">Books</a></li>
                </ul>
            </div>
        </div>
    </nav>

    <div className="py-3">
        <div className="container">
```

...

```
    <tfoot>
        <tr id="addNew">
            <td colSpan="4"></td>
            <td className="text-center">
                <button className="btn"><img src="img/plus.png" alt="add"
            </td>
        </tr>
    </tfoot>
</table>
```

Fig. 117 : Users component with fake value

6.3.4.2 Routing from Login to Users

OK, now let's add another route and check if, when we log in, we can reach this route.

First defining the route.
Routes are defined in App.js.
We created a "Users" component. We have to add this as a Route in App.js as we did for Login. Do not forget to include Users top of the file.

[Fig. 118] shows the routing management in App.js.

```
<Routes>
  <Route exact path='/' element={<Login />} />
  <Route exact path='/users' element={<Users />} />
</Routes>
```

Fig. 118 : Routing in App.js

If npm is not started, start it. As it compiles the files, you should have informations about possible errors or warnings.

Now, let's check we can reach our components.
With your browser, call the 2 pages at

- http ://localhost :3000 should be called by default
- http ://localhost :3000/users

[Fig. 119] shows the display component for each of them.

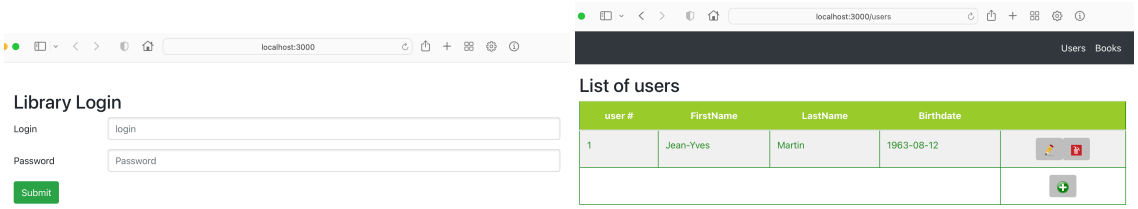


Fig. 119 : Login and Users screens

Now let's connect authentication and users list.

When user authenticates (canLogin is true) we redirect to component User. So, we only have to add these lines at the beginning of the render method in the Login component :

```
if (this.state.canLogin) {
    return <Navigate push to="/users" />;
}
```

Navigate is a react-router-dom component that will restructure the DOM to display our new page.

"to" tells the route is "/users" (we know we can reach the component, because we defined the route in App.js).

"push" means we push the page in the browser history to be able to go back with the browser arrows.

Do not forget to include "Navigate" import top of the file (from 'react-router-dom').

[Fig. 120] shows the added lines to switch to users in function render of Login component.

```
render() {
  if (this.state.canLogin) {
      return <Navigate push to="/users" />;
  }
  return (
      <div className="py-5">
```

Fig. 120 : Switch from Login to Users

Now, let's try.
Connect to the Login page.
Use bad login/password (means not admin/admin), you should come back to the Login page.
Use right Login/Password (admin/admin), you should switch to the Users page.

6.3.4.3 Access restriction

But what about POST ?
We might not be allowed to call users in GET mode.

Do not forget it is a single page application. That means you do not exchange with the server to load a new page. Pages are already loaded. Data is not, but pages are. Switching between pages means react empties most of the DOM structures to reloads a new structure. That means requesting a new page in POST mode is non-sense.

So, how could we do ?
In fact the solution is "session tokens". We will use the "sessionStorage" that can store variables in the browser session storage. But we will have to change several things for that.

First, we define 3 methods in App.js :

- setToken that sets a "token", a cookie in the session storage
- getToken that checks that there is a "token" in the session storage
- removeToken... that removes the "token"

[Fig. 121] shows the token functions management.

```
function setToken(userToken) {
    sessionStorage.setItem('token', JSON.stringify(userToken));
}

function getToken() {
    const tokenString = sessionStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return (userToken != null)
}

function removeToken() {
    sessionStorage.removeItem('token');
}

class App extends Component {
```

Fig. 121 : Token management functions in App.js

Next, we have to ask component “Login” to set the token element when we log in.
Maybe we should also remove the token each time we launch “Login”.

But how could we do this ? Methods are located in App.js and we can’t send fonctions to Login.
Have a look to Login constructor. Do you see the “props” ? props are properties that are sent to the component by a caller component. Sent properties can be variables …or functions.

So, we have 2 things to do :

- in App.js, when we call Login, we must send it our functions.
- in Login.js, we have to use these functions.

Let’s start with sending functions. Change a little bit Route calls. We add 2 props : setToken (equal to the function setToken) and removeToken (equal to the function removeToken) to route “Login”.

You can add as many properties as you want. The first element will be the name of the property in the props of the connected Component, the value is the one in the braces.

[Fig. 122] shows the way we send props to Login.

```
class App extends Component {
  render() {
    return (
      <div className="App">
        <Router>
          <Routes>
            <Route exact path='/' element={<Login setToken={setToken} removeToken={removeToken}  />} />
```

Fig. 122 : Route to Login with props

Next, we have to use the sent functions in Login.
You receive the functions sent by App.js in props, the constructor parameter. We can also use it as “this.props”. If you want to see how it works, you can use a “console.log(props);” at the end of the

constructor. That should display variable “props” in the console of your web browser. You should have the 2 functions.

In the render function, we first remove the token.
Then, if user correctly authenticated (canLogin is true), we Navigate to Login.
Maybe we could set the token just before we navigate.

We could also set the token in checkLogin, when user authenticates. But in this case, the remove token call MUST BE after we check canLogin in render. You can place it between the closing brace of the test about canLogin and the return instruction.

To set the token, we add “this.props.setToken(”whatever you want”);”.
And to remove it,”this.props.removeToken();”.

[Fig. 123] shows a way we can remove and set the token.

```
render() {
  this.props.removeToken();
  if (this.state.canLogin) {
      this.props.setToken("Kirito was here!");
      return <Navigate push to="/users" />;
  }
  return (
      <div className="py-5">
```

Fig. 123 : render removes token in Login.js

Last step : when we call Users, it might redirect to “/” if there is no defined token.

- in App.js, modify route to Users so that we send a property “getToken”, which value is the function getToken.
- in Users.js, each time we render, we get the token. If it is not defined we navigate to “/”.

[Fig. 124] shows the way we check token.

```
    render() {
        const token = this.props.getToken();
        if (!token) {
            return <Navigate to="/" />;
        }
        return (
<div>
    <nav className="navbar navbar-expand-md navbar-dark bg-dark">
```

Fig. 124 : Users checks token

Ok. Now let’s check everything works.
Did you add the import of Navigate? Top of Users.js?

Connect to http ://localhost :3000
That should display the login page.

Now, switch to http ://localhost :3000/users
You should be redirected to the Login page.

Log in.
You should access to the fake users list, whereas you were not able before.

6.3.5 Creating a BackEnd server

6.3.5.1 Creating nodeJS server

How can we display our database Users list ?
How could we manage authentication ?

Currently we manage authentication in our application and it is not a good way of doing things. Our users list is a fake one.

We spoke about a BackEnd. Shouldn't it be the right place for implementing all that ?

As a BackEnd server, we will use nodeJS. NodeJS is a Javascript server. So we have to use Javascript to build the scripts that will manage requests.

Create a directory "server" in the same directory as "prwebreact". Go inside this directory with you Terminal / Command tool. We will initialise a node server.

```
npm init
```

Then, validate every question for which you are asked for an answer (default answer is ok). Of course, you can change default answer if you want to, but be sure of what you do. That will create a file "package.json" that contains server configuration elements.

Now, we need 2 modules :

- pg to connect to the postgresql server
- express to manage some server elements for us.

```
npm install express
npm install pg
```

This will create a directory "node_modules" and install our requirements. You may find the informations in "package.json".

Now, we need the server script.
Create a file “server.js”. That file will contain our BackEnd scripts and will be used by nodeJS. We have left an initial version of server.js in the materials.

[Fig. 125] shows the content of the file “server.js”.

```
const express = require('express');
const pg = require("pg");
const app = express();

var conString = 'postgres://prweb:prweb@localhost:5432/prweb_react';

app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Default to accept all requests
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  next();
});


// Must be LAST instruction of the file
// Listen to port 8000
app.listen(8000, () => {
  console.log('Server started!')
});
```

Fig. 125 : NodeJS server script in server.js

What do we find in “server.js” ?

- 2 first lines are library inclusions : express to manage some elements, pg for postgresql.
- line 3 tells we have an app that uses express.
  We manage nodeJS routes through “app”.
- line 5 : conString tells what is our connection URL (we will use it later) :
  - protocol is postgres (understand we will use postgresql),
  - user connects with login prweb and password prweb,
  - postgresql server is at localhost on port 5432,
  - database is prweb_react
- 2 next lines (app.use ...) explain that the parser should be able to encode responses, and should be able to encode data as JSON data.
- next lines explain we don’t care where the request comes from (nodeJS do not manage React application, but might answer requests coming from it). These instructions will ALLWAYS be called when we send a request. We use “next()” to tell we go to the next valid “use”.
- last lines ask the server to listen to port 8000 (default NodeJS port)
  this instruction **MUST ALWAYS BE THE LAST ONE**.

OK, let's try it.
Open a new terminal / Command tools to launch the NodeJS server.

```
npm start
```

Check it compiles and there is no error when launched. You can stop it with CTRL-C.

Of course, when server is launched, you can check it works with http ://localhost :8000, but as there is no route to reply to, you will have a message that tells the server can't handle / URL.

6.3.5.2 Writing authentication function

Right now, our server does nothing.
We wil start by something easy to understand how it works.

Let's write a first replier for our server. We have a login page and we would like the server to identify our user. So we start by defining a function in our server that manages authentication.

To reply to a request, we use express, and the variable app we defined.

- **app.get** is used for a GET request
- **app.post** is used for a POST request

Both of these functions have 2 parameters :

- the route we manage
- a function that manages the request and the response for this route.

To get parameters, if req is the parameter that represents the request, then

- **req.query** contains GET parameters
- **req.body** contains POST parameters

So, what do we have to do in our function ?

- get parameters (login and password)
- check (login / password) is ok, and give the answer as a reply. We will use "JSON.stringify" that translates a JSON object to a JSON string, so that we send it back.
- send the response as a JSON string.

[Fig. 126] shows how we can implement such a function in "server.js". Our route is "/authenticate". Currently we implement it for a GET request to be able to test it.

```
app.get("/authenticate", function(req, res) {
  var login = req.query.login;
  var passwd = req.query.passwd;

  var jsonString;
  if ((login === "admin") && (passwd === "admin")) {
    jsonString = JSON.stringify({ok:1});
  } else {
    jsonString = JSON.stringify({ok:0});
  }
  res.setHeader('Content-Type', 'application/json');
  res.send(jsonString);
});
```

Fig. 126 : NodeJS server response to an authentication

Remember, this function has to be placed **before** the "listen" part.

Stop the node server and relaunch it. Once launched, the server does not check if you modify the file "server.js". So, **each time you modify "server.js", you will have to stop server and relaunch it**.

With your browser, check it works with that URL :
http ://localhost :8000/authenticate ?login=admin&passwd=admin

Do you have a response that tells ok is 1 ?
Ok, now change login and/or password. What is the response ?

Now, let's change the script to POST :

- change app.get to app.post
- change req.query to req.body

You may not be able to use the script with your browser because, now, it is in POST mode.

Do not forget to stop server and relaunch it.

6.3.5.3 Checking Login Password from the nodeJS server

As our React application will often interact with our nodeJS server, maybe it should be a good idea to write a function that manages this interaction. We will ensure our function exchange with a POST method so that we can protect our nodeJS server from unwanted requests.

This function will have 2 parameters : the route we call in the nodeJS server, and data to be send as a JSON object.

We will use the "fetch" function that can send a request to a server.
"fetch" is an asynchronous function, so we will have to manage its response with the right method.

The response from the nodeJS server is a string that contains a JSON string.
So first, we will have to change the string response to an object.

Next, our response is a **Promise**, a specific JS object that tells "I will give you the result when it will be filled". A promise is often linked to asynchronous functions. So, never consider a JS instruction is ended when you go to the next one. If it is an asynchronous function, call is launched and a background function is launched and in charge of managing the result. That means the instruction is ended but the result is not arrived yet. So, if you try to use the Promised result and is not filled yet, you process with undefined data.

That means you will often have to take care of the way you manage data when coming from another source. And for all asynchronous functions.

Let's create a file "util.js" to write our function, and place it in the "src" directory of the React project. You will find a version in the materials.

[Fig. 127] shows how we can implement such a function.

```
let server = "http://localhost:8000";

function postServiceData(method, params) {
    return fetch(server + "/" + method, {
        method: 'POST',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(params)
    })
    .then((response) =>
        response.json()
        .then((data) => {
            return Promise.resolve(data);
        })
        .catch (error => {
            console.log(error);
            return error;
        })
    );
}

export  { postServiceData };
```

Fig. 127 : Util manages a POST request to the nodeJS server

6.3.6 Login component connects to BackEnd to authenticate

Our Backend is able to reply to route "/authenticate" with a login and a password.
So, we have to tell the component Login that when it checks login/password, it should ask to our BackEnd.

As we exchange with the nodeJS server, we will use our function postServiceData. Let's import it from util.js at the beginning of the file.

```
import { postServiceData } from './util';
```

Then, we have to change our "checkLogin" function in Login.js so that it calls the server. We have to :

- Prepare parameters (let's call it params) as a JSON object
- Call method postServiceData and manage response when it arrives (remember the Promise ?) Method to be called for our NodeJS server is "authenticate". Its params should define the login and password to check.

[Fig. 128] shows how we can implement such a call.

```
checkLogin(event) {
    event.preventDefault();
    const params = {login: this.state.login, passwd: this.state.pass};
    postServiceData("authenticate", params).then((data) => {
        if (data.ok === 1) {
            this.setState({canLogin: true});
        }
    });
}
```

Fig. 128 : checkLogin calls postServiceData

When we receive data, we manage it with **"then"** to be sure result "data" is filled when it is really received (otherwise it will be undefined). "data" contains our response as a JSON object. It consists in a field "ok" that can be 0 or 1. Remember the tests on the nodeJS server ?

Ok, now try it. Close your browser and re-open it to be sure your session variables are deleted.
Use URL http ://localhost :3000
Use right login/password to connect. You should have access to the users page.
But this time, authentication comes from the server.

6.3.7 Manage database list of users

Now, the users list. We want the real one, not a fake one. So :

- nodeJS server must define a route that collects data from the database and send result back.
- Component Users must collect list of users from nodeJS and display it.

6.3.7.1 Collecting users data from database

Let's start with the nodeJS server.

Create a new POST route "/users" in "server.js". We create to get all users, send a SQL request to the server, collect data and send result as a JSON object.

As, for this project, we will have to manage several requests to the database, we will write a generic function "getSQLResult" that manages a SQL request and produces a response. That means :

- To create a postgresql client object (new pg)
- To connect to the postgresql URL
- To send the query, with a set of values (avoiding SQL injection)
- if it is ok, to get the result as a JSON object that we send back as a JSON string.

When you implement such a function, never forget it is asynchronous. That means you must wait for the result to be created before you use it. So, we have to take care to the way we write it. Never write it as a list of instructions or you will have a bad surprise when you run it.
[Fig. 129] shows how we can implement such a function.

```
function getSQLResult(req, res, sqlRequest, values) {
    var client = new pg.Client(conString);
    client.connect(function(err) {
        if (err) {
            // Cannot connect
            console.error('cannot connect to postgres', err);
            res.status(500).end('Database connection error!');
        } else {
            // Connection is OK
            client.query(sqlRequest, values, function(err, result) {
                if (err) {
                    // Request fails
                    console.error('bad request', err);
                    res.status(500).end('Bad request error!');
                } else {
                    // Build result array from SQL result rows
                    var results = [];
                    for (var ind in result.rows) {
                        results.push(result.rows[ind]);
                    }
                    // Convert object to a JSON string and send it back
                    res.setHeader('Content-Type', 'application/json');
                    res.send(JSON.stringify(results));
                }
                client.end();
            });
        }
    });
}
```

Fig. 129 : server.js getSQLResult function

Ok, now, let's call getSQLResult and return the rights users list. We need a route and a function that manages the route. We have to :

- build the SQL request that will give us the result
- call getSQLResult with the SQL request, the SQL request parameters (none here).

getSQLResult will connect to the database server, launch the SQL request (with no parameter here), and send the result back to the caller.

[Fig. 130] shows how we can implement such a function. We write it with as GET method to be able to test it with our browser. Then we will write the true version, with POST method.

```
app.get("/users", function(req, res) {
  var sqlRequest = "SELECT * FROM Person ORDER BY Person_LastName, Person_FirstName";
  var values = [];
  getSQLResult(req, res, sqlRequest, values);
});
```

Fig. 130 : server.js send a request to the database server and send back response

Stop the nodeJS server and relaunch it. Use URL : http ://localhost :8000/users
You should have a list of users in your database.

Change script so that you reply to a POST request.

6.3.7.2 Manage users list in component Users

Ok, now we have to get the real users list in Users. That means :

- call nodeJS server to get the users list
- display users.

Unfortunately, it is not so easy. the NodeJS call is an async function. If you try to get data in the constructor, the async call will not end before the constructor ends. And you will have a problem when you will try to use data : it will be undefined.

Therefore, we use a specific React function : **componentDidMount**
This function is called automatically (if it is defined) when the component is mounted. This is the place where async functions that initialise data should be called. We will create a "fetch" function that will get users from NodeJS server.

To manage data, we will have to set users list in a state object. So, in the function fetch, we call the postServiceData with url "users" to get the users list from the server. When it arrives (means when the promise gives us a result), we set our state to the returned value.

[Fig. 131] shows the way we can implement that.

- We initialize array users to an empty array in the constructor
- we define the componentDidMount function that calls the fetch function
- we define th efetch function that ask data to the NodeJS server ans set them into the users array.

For the params in postServiceData, we use a dummy param. That will avoid compiling problems. Also, we could use a null parameter and take that into account in postServiceData.

```
class Users extends Component {
    constructor(props) {
        super(props);
        this.state = {users: []};
    }

    componentDidMount() {
        this.fetch();
    }

    fetch() {
        const params = {ok:1};
        postServiceData("users", params).then((data) => {
            this.setState({users: data});
        });
    }
```

Fig. 131 : Get data for the Users component

Next, we have to display the users list. We have to loop on users to display each of them.
So we will create 2 components :

- 1 component to display the list of users (the tbody part of the table)
- 1 component to display 1 user.

Of course you can define 1 file per component but, as these components are linked to component Users, we will define then in the same file, just before the definition of the component Users.

Let's start with the users array. This array is in the component's props.
We only have to loop on this array (with function **map**) and display each user with the component that displays 1 user. Usually, map requires a key, a unique value to loop on the array. It is defined by attribute key. The associated value is the user's ID.

Oh, one more thing : a component returns a tag. So we can't return only the map call. It has to be included in a tag. That is why the "tbody" tag is used in this function.

Do you remember how we call a component when we select a Route in App? We use the same method to send a user to the component UserInList.

[Fig. 132] shows the script we can use for this component.
You can place it in Users.js, just before the class Users.

```
class UsersInList extends Component {
    render() {
        let users = this.props.users;
        return (<tbody>
            {users.map((user) => <UserInList user={user} key={user.person_id} /> )}
         </tbody>);
    }
}
```

Fig. 132 : loop on users

Next, the component that displays 1 user. It receives an user in the props. We only have to display the “tr” line. [Fig. 133] shows how we can display 1 user.

```
class UserInList extends Component {
    render() {
        let user = this.props.user;
        return (
            <tr>
                <td>{user.person_id}</td>
                <td>{user.person_firstname}</td>
                <td>{user.person_lastname}</td>
                <td>{(new Date(user.person_birthdate)).toLocaleDateString()}</td>
                <td className="text-center">
                    <button name="edit" className="btn"><img src="img/edit.png" alt="edit" className="icon" /></button>
                    <button name="delete" className="btn"><img src="img/delete.png" alt="delete" className="icon" /></button>
                </td>
            </tr>
        );
    }
}
```

Fig. 133 : display 1 user

Finally, we have to use component UsersInList in component Users when we want to display users. Remember that UsersInList displays the tbody tag.
[Fig. 134] shows how we can use UsersInList when we display the users list in component Users.

```
<thead>
    <tr>
        <th scope="col" className="text-center">user #</th>
        <th scope="col" className="text-center">FirstName</th>
        <th scope="col" className="text-center">LastName</th>
        <th scope="col" className="text-center">Birthdate</th>
        <th scope="col"></th>
    </tr>
</thead>
<UsersInList users={this.state.users} />
<tfoot>
    <tr id="addNew">
        <td colSpan="4"></td>
        <td className="text-center"><button className="btn"><img
    </tr>
</tfoot>
```

Fig. 134 : Using UsersInList in Users

Ok, let's try what we did.
If React is already started, it should have recompile source files when you saved them.
Stop and restart the NodeJS server.

If it is not open yet, with your browser use URL : http ://localhost :3000/users. You should have the users list, with the users in the database. [Fig. 135] shows our result.

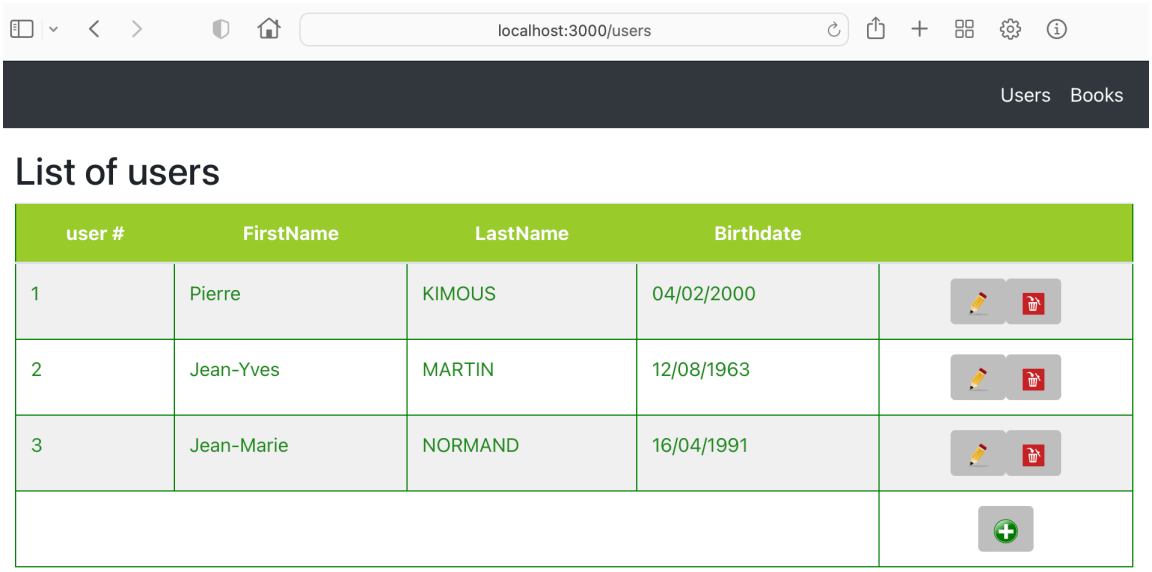


Fig. 135 : Users list

6.3.8 Edit user

Ok, now let's edit an user.
For each user, we have a button to edit user. What do we have to do ?

- When button is clicked in Users.js, get user's information and switch to a component User
- Create the component "User" that displays user informations

First, the button. We use a "**onClick**" attribute to launch a local method. Local method have to tell the render function that we want to navigate to "/user" with an attribute, the user's id. "render" method has to check if we want to navigate to "/user".

We need a state, let's say "wantToEdit", that tells we want to edit user. Do not forget to define it in the states in the constructor.

We also need a method that manages the onClick attribute of the button. Do not forget to set this method in the constructor so that it is associated to the current object.

[Fig. 136] shows the way we can switch to edit user in class UserInList. Did you add the editUser assignation in the constructor ?

Have a look to Login.js if you don't remember how you might assign a fonction in the constructor.

```
editUser(event) {
    event.preventDefault();
    this.setState({wantToEdit: true});
}

render() {
    let user = this.props.user;
    if (this.state.wantToEdit) {
        return <Navigate to="/user" state={{id: user.person_id}}/>;
    }
    return (
        <tr>
            <td>{user.person_id}</td>
            <td>{user.person_firstname}</td>
            <td>{user.person_lastname}</td>
            <td>{(new Date(user.person_birthdate)).toLocaleDateString()}</td>
            <td className="text-center">
                <button name="edit" className="btn" onClick={this.editUser}><img src="img/edit.png" alt=
                <button name="delete" className="btn"><img src="img/delete.png" alt="delete" className="
            </td>
        </tr>
    );
}
```

Fig. 136 : Edit User mechanism

Next, User.js, the script that edit 1 user.
Unfortunately, it is not so easy to transfer parameters from a class to another class. The easiest way is to use a **Hook**. Hooks are specific functions that will allow us to use react native functions. In our case, we have to use function "useLocation" to retrieve parameters.

Here are the comparison of a Hook and a Class way of implementing a component :

Hook

```
const MyFunction = (parameter) => {
    some instructions;
    return (
        something
    );
}
```

Class

```
class MyFunction extends Component {
    render() {
        some instructions;
        return (
            something
        );
    }
}
```

And, of course, you can use a hook in a tag, like you do for classes. You also have to export them, if necessary, the same way you do it for a class. The main problem with a class is that it can't use some react native functions (like useLocation, useParam, ...). The only way to do this is hooks.

So, first, we define a hook. As it is the one that is called, its name must be "User". This hook has to retrieve the sent parameter and render a component, what we wrote with the file User html in the

first practical work. Let's call this component UserClass. Do not forget to export User at the end of the file. UserClass do not have to be exported, it will be used locally.

[Fig. 137] shows how you can use the hook and the component.

```
import React, { Component } from 'react';
import { useLocation } from 'react-router-dom';

// Hook definition
const User = () => {
    const location = useLocation();
    return <UserClass id={location.state.id} />;
};

// Class definition
class UserClass extends Component {
    render() {
        return (
        <div>
        </div>
    );
  }
}


export default User;
```

Fig. 137 : User Hook and Component

For the UserClass class, your render can be build with your old user.html file. Do you remember what you did for Login ? You should replace class by className, define a handle function for each text input, link them with input texts, and bind them in the constructor.

Next, create a new component "User" in User.js As you did for Users, you should get data from the NodeJS server. Do not forget to import postServiceData at the top of the file. [Fig. 138] shows how you can get data in UserClass.

```
componentDidMount() {
    this.fetch();
}

fetch() {
    const params = { id: this.props.id };
    postServiceData("user", params).then((data) => {
        let user = data[0];
        let theDate = new Date(user.person_birthdate).toLocaleDateString();
        this.setState({person_id: user.person_id});
        this.setState({person_lastname: user.person_lastname});
        this.setState({person_firstname: user.person_firstname});
        this.setState({person_birthdate: theDate});
    });
}
```

Fig. 138 : Get data for a user

Next, NodeJS must be able to reply to the user request. It should get “id” from the request, and use a request to get data. [Fig. 139] shows the method you should add to server.js.

```
app.post("/user", function(req, res) {
    var id = req.body.id;
    var sqlRequest = "SELECT * FROM Person WHERE person_id=$1";
    var values = [id];
    getSQLResult(req, res, sqlRequest, values);
});
```

Fig. 139 : Upload Person data from the id

Do not forget to stop your NodeJS server and relaunch it, or your modification will not been applied.

So, your script for User.js should like [Fig. 140].

```
render() {
    return (
    <div className="py-3">
        <div className="container">
            <div className="row">
                <div className="col-md-12">
                    <h2 className="">Create / Edit User page</h2>
                </div>
            </div>
            <div className="row">
                <div className="col-md-12">
                    <div className="table-responsive">
                    <table className="table table-striped">
                        <tbody>
                            <tr>
                                <th scope="col">user #</th>
                                <td>{this.state.person_id}</td>
                            </tr>
                            <tr>
                                <th scope="col">FirstName</th>
                                <td><input type="text" className="form-control" value={this.state.person_firstname} /></td>
                            </tr>
                            <tr>
                                <th scope="col">LastName</th>
                                <td><input type="text" className="form-control" value={this.state.person_lastname} /></td>
                            </tr>
                            <tr>
                                <th scope="col">Birthdate</th>
                                <td><input type="text" className="form-control" value={this.state.person_birthdate} /></td>
                            </tr>
                        </tbody>
                        <tfoot>
                            <tr>
                                <td colSpan="2" className="text-center">
                                    <button type="submit" className="btn btn-block btn-primary">Save</button>
                                </td>
                            </tr>
                        </tfoot>
                    </table>
                    </div>
                </div>
            </div>
        </div>

    </div>
    );
}
```

Fig. 140 : User.js script

At last, we have to define the route in App.js

- import User
- add a route to User.

Do it like in [Fig. 141].

```
<Router>
  <Routes>
    <Route exact path='/' element={<Login setToken={setToken} removeToken={removeToken} />} />
    <Route exact path='/users' element={<Users getToken={getToken} />} />
    <Route exact path='/user' />
  </Routes>
</Router>
```

Fig. 141 : Add /user route in App.js

Ok, Let's try. Use http ://localhost :3000.
Connect. You should have the users list.
Click on an edit button for any user. That should lead to the edit page of that user.

[Fig. 142] shows the result.

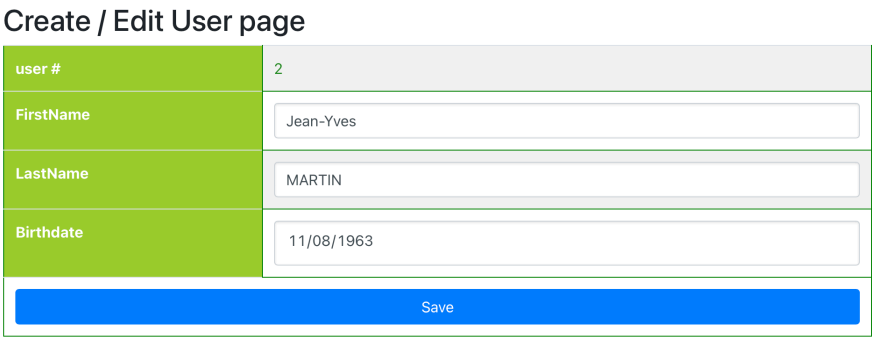


Fig. 142 : Edit page for a user

Questions

What is a Hook used for ?
How do you manage a route ? What is the link between Navigate and Route ?
How do you send informations to a Hook ? to a Class ?
Why can't you do a SQL request from React ?
What is a Promise ? Why should you use "then" with a Promise ?

Ok. Now the “Save” button and the saving elements.

Do you remember what we did for login.js to take into account the fields and associate them to states ?

Define 1 state for each field, 1 method to associate the input value with the state and add the onChange attribute to the fields. Do not forget to bind the methods in the constructor.

We also need a method to save user.
That method should collect data from the states, build a JSON param to call postServiceData. Do not forget to bind this method in the constructor.

When data are saved to the database, we go back to “/users”. So we need 1 more state that tells render to navigate back to “/users”.

[Fig. 143] shows a part of the render function.

```
render() {
    if (this.state.canGoBack) {
        return <Navigate to="/users" />;
    }
    return (
    <div className="py-3">
        <div className="container">
            <div className="row">
                <div className="col-md-12">
                    <h2 className="">Create / Edit User page</h2>
                </div>
            </div>
            <div className="row">
```

...

```
                        <tr>
                            <th scope="col">Birthdate</th>
                            <td><input type="text" className="form-control" onChange={this.handleChangePersonBirthDate}  value={this.state.person_birthdate} /></td>
                        </tr>
                    </tbody>
                    <tfoot>
                        <tr>
                            <td colSpan="2" className="text-center">
                                <button type="submit" className="btn btn-block btn-primary" onClick={this.saveUser}>Save</button>
                            </td>
                        </tr>
                    </tfoot>
                </table>
```

Fig. 143 : Convert String to Date

Oh, one more thing, … Dates.
Unfortunately, dates like ‘dd/mm/yyyy’ will not be recognized. We have to use the ISO format.
For that, we need a function that converts strings to an ISO date. You can write it at the same level as the hooks and classes or place it in util.js and import it when you need it.

[Fig. 144] shows a way to write such a function.

```
function stringToDate(_date) {
    var delimiter = "-";
    var formatLowerCase = "yyyy-mm-dd";
    if (_date.indexOf("/" > 0)) {
        delimiter = "/";
        formatLowerCase = "dd/mm/yyyy";
    }
    var formatItems=formatLowerCase.split(delimiter);
    var dateItems=_date.split(delimiter);

    var monthIndex=formatItems.indexOf("mm");
    var dayIndex=formatItems.indexOf("dd");
    var yearIndex=formatItems.indexOf("yyyy");
    var month=parseInt(dateItems[monthIndex]);
    month-=1;

    var formatedDate = new Date(dateItems[yearIndex],month,dateItems[dayIndex]);
    return formatedDate.toDateString();
}
```

Fig. 144 : Convert String to Date

Ok, now we can write the saveUser function. [Fig. 145] shows how we can write it.

```
saveUser(event) {
    event.preventDefault();
    var theDate = stringToDate(this.state.person_birthdate);
    var params = {person_id: this.state.person_id,
                  person_lastname: this.state.person_lastname,
                  person_firstname: this.state.person_firstname,
                  person_birthdate: theDate};
    postServiceData("saveUser", params).then((data) => {
        this.setState({canGoBack: true});
    });
}

render() {
    if (this.state.canGoBack) {
        return <Navigate to="/users" />;
    }
    return (
    <div className="py-3">
```

Fig. 145 : save user

Did you add “canGoBack” to the state variables in the constructor?
And did you import Navigate? And stringToDate?

And of course we need nodeJS to save user according to the informations we give to it.

We write our function so that if person_id value is negative, we create a new Person. If it is positive, we update it. We have to get parameters from the request. According to ID value, we build the SQL request. We use our function getSQLResult that will process the request and return a JSON object (the ID value in database) to the react app.

And of course, we have to define it as a POST route “/user”.

[Fig. 146] shows the script in server.js.

```
app.post("/saveUser", function(req, res) {
    var person_id = req.body.person_id;
    var person_firstname = req.body.person_firstname;
    var person_lastname = req.body.person_lastname;
    var person_birthdate = req.body.person_birthdate;

    var sqlRequest = "";
    var values = [];
    // We build a request that returns ID value to be able to return it
    if (person_id < 0) {
        sqlRequest = "INSERT INTO Person(Person_FirstName, Person_LastName, Person_BirthDate)"
        + " VALUES ($1, $2, $3)"
        + " RETURNING Person_ID";
        values = [person_firstname, person_lastname, person_birthdate];
    } else {
        sqlRequest = "UPDATE Person SET"
        + " Person_FirstName=$1, Person_LastName=$2, Person_BirthDate=$3"
        + " WHERE Person_ID=$4"
        + " RETURNING Person_ID";
        values = [person_firstname, person_lastname, person_birthdate, person_id];
    }
    getSQLResult(req, res, sqlRequest, values);
});
```

Fig. 146 : save user in nodeJS

Restart your nodeJS server to take into account your modification in “server.js”.
Close your browser. Re-open it.
Use URL http ://localhost :3000
Use right login/password to connect.
Click on a button to edit an user. Change something. Click on button save.
Check, in the list, that user is modified.

6.3.9 Add user

Back to Users.
When we click on the button “+”, bottom of the table, we want to create a new user.

Modify the button “+” so that it calls a behavior method “createUser”. This method set state “createNew” (yes, one more state) to true. Remember to define this state variable in the constructor and set it to false.

In the render method, check if createNew is true, and navigate to “/user” with a dummy person_id you may recognise in UserClass (-1 should be a nice idea).

Now, back to UserClass.
Maybe it is not useful to try to fetch the user only if person_id is >0.
When we save user (in server.js), if we have a negative value for the person_id, we create a new user.

6.3.10 Delete user

Quite easy.

Back to Users. UserInList should be better.
Add a call to a deleteUser method in the delete button of the user.
Create “deleteUser” behavior method. Bind it in constructor. “deleteUser” send a request to nodeJS with current ID. It also tells render it has to navigate to “/users” when the appropriate state is set.

Implement a POST method in server.js that delete user and returns :

- maybe an empty list if you use getSQLResult and it is ok,
- an error otherwise.

Restart nodeJS server.
Close your browser. Re-open it.
Connect with the right (login/password). Create new user.
Delete it.

6.3.11 Security management for User

When we built Users, we added restriction access with a token.
We have to do the same for User, we only have to check if token is ok.

First, add “getToken” in App.js for User as you did for the other Routes.
Next, we have to get “getToken” in the Hook User.

[Fig. 147] shows how you can get it and sent it to UserClass.

```
const User = ({getToken}) => {
    const location = useLocation();
    if (location.state === null) {
        return <Navigate to="/" />;
    } else {
        return <UserClass id={location.state.id} getToken={getToken} />;
    }
};
```

Fig. 147 : getToken in hook

Check it works.

- Close your browser. Re-open it. Open URL http ://localhost :3000/User
  You should have the Login page.
- Login, select user.
  You should have you user.

6.3.12 Books : list, edit, add, delete

Ok, now let's manage books.

This is quite similar to users. You should consider duplicating files and scripts from Users and User.

Create a Books component. Display list of books. You can use the work you did in the first practical work.

Maybe using a components BooksInList and BookInList would be a good idea. Add Edit and Delete buttons.

Create component Book. Implement the Save button for existing and new books.

Do not forget to declare routes to "/books" and "/book" in component App.
Do not forget to import them.

You will also have to write some scripts in nodeJS server. : "/books", "/book", "/saveBook" and "/deleteBook". Do not forget to restart the NodeJS server.

To ensure it works, you should temporarily remove security. Comment (use //) the lines that navigates to "/" when token is not found.

Use your browser to connect to URL to http ://localhost :3000/books.
Check your books list. Edit a book. Change something. Save.
Add new book.
Remove it.

[Fig. 148] shows our book list

Fig. 148 : List books

Add security with getToken in Books and Book.

6.3.13 Switching from Users to Books in the nav bar.

Ok. We have a nav bar. And we want to switch to "/users" or "/books".

The NavBar should be the same in every page.
What about writing it as a component?

[Fig. 149] shows our NavBar component in NavBar.js

```
import React, { Component } from 'react';

export class NavBar extends Component {
    render() {
        return (
    <nav className="navbar navbar-expand-md navbar-dark bg-dark">
        <div className="container">
            <div className="collapse navbar-collapse" id="navbar1">
                <ul className="navbar-nav ml-auto">
                    <li className="nav-item"> <a className="nav-link text-white" href="/users" >Users</a></li>
                    <li className="nav-item"> <a className="nav-link text-white" href="/books" >Books</a></li>
                </ul>
            </div>
        </div>
    </nav>
        );
    }
}
```

Fig. 149 : NavBar script

Now, we have to add this component in Books and Users.

- import NavBar at the beginning of each file
- replace the tag nav by a call to component NavBar (use <NavBar / >)

Close your browser.
Re-Open it, connect to you application and check you can switch from a page to the other one.

Questions

When is the security token created / removed?
Why do we have to send the function getToken as a parameter to the components?
Why did we define a NavBar component?

6.3.14 Manage borrowings

Ok, last part : users (people) may be able to borrow books, return books.

So, first we will define a Borrows component. And the same way we did for Users or Books, we define 2 components : Maybe BorrowsInList and BorrowInList.

Oh, when you will write your Borrow routes, do net forget that when you retrieve a line, it would be a good idea to include the book's title.

So, what do we have to do ?

- Create routes in server.js
- Move stringToDate from User.js to util.js if you created it in Users
  Do not forget to export it in util.js and import it in the files where you need it.
- Create Component Borrows and manage the borrows for a user
- And of course the Borrows call at the end of the User render

6.3.14.1 Defining routes in NodeJS

We have to :

- add a "/borrows" route to retrieve a person_id borrows in App.js
- add a "/borrow" route to retrieve a borrow_id informations in App.js
- add a "/saveBorrow" route to create a new line in Borrow according to the person_id, a book_id and a date of borrow in server.js
- add a "/returnBook" route to return a book for a given borrow_id in server.js

6.3.14.2 Component Borrows

You can build this component using component Users or Books.

But there are some things to consider :

- create a Component Borrows
  - Build the render return value frim first practical work
  - When component is mounted, retrieve borrows ans books from NodeJS server.
    NB : you should use routes "/borrows" and "/books"
- create a Component BorrowsInList (like for Users and Books)
- create a Component BorrowInList (like for Users and Books)

- we suggest you create a component BorrowReturn for the “return” cell of the borrow line.
  It will be easier to manage the render function.
  You can also manage the return button in this component (and the call to the NodeJS server to return a book)
- For the HTML line that adds a new Borrowed book
  - You need a loop (map) on the books to display the list in the SELECT tag.
  - define a state to manage the selected book
  - Do not forget a handle to manage the select change (and set the selected book value)
  - click on “+” launched a behavior method that creates a new borrow
- When you create a new Borrow :
  - you send (person_id, book_id (the selected book), current date) to create a new borrow.
  - you should retrive the created borrow_id
  - THEN you can ask to retrieve the borrow line corresponding to new borrow_id
  - THEN if you retrieve the current borrows list, you can push the line returned by NodeJS.
    You only have to reset the new borrow list with the increased array.

[Fig. 150] shows the result you may obtain.

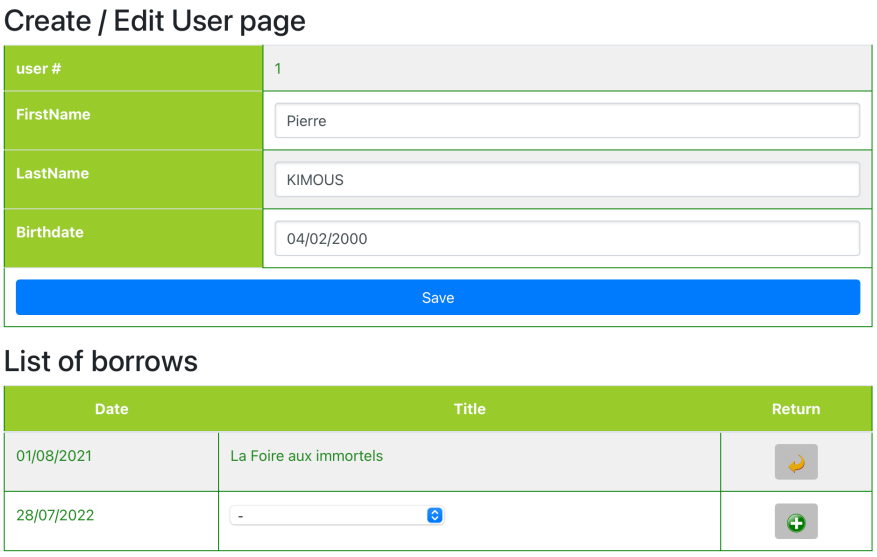


Fig. 150 : User can borrow books

Maybe it will be a bit complicated to manage the return date because Javascript functions about dates do not always return what we would like. [Fig. 151] shows the way we implemented it.

```
class BorrowReturn extends Component {
    constructor(props) {
        super(props);

        this.state = {borrow_return: this.props.borrow_return};
        this.returnBorrow = this.returnBorrow.bind(this);
    }

    returnBorrow(event) {
        event.preventDefault();
        let curDate = new Date();
        let theDate = curDate.toISOString().substring(0,10);
        const params = {borrow_id: this.props.borrow_id, borrow_return: theDate};
        postServiceData("returnBook", params).then((data) => {
            this.setState({borrow_return: curDate});
        });
    }

    render() {
        if (this.state.borrow_return !== null) {
            let theDate = new Date(this.state.borrow_return).toLocaleDateString();
            return <span>{theDate}</span>
        } else {
            return <button className="btn" onClick={this.returnBorrow}><img src="img/return.png"
        }
    }
}
```

```
class BorrowInList extends Component {
    render() {
        let borrow = this.props.borrow;
        return (
            <tr>
                <td>{(new Date(borrow.borrow_date)).toLocaleDateString()}</td>
                <td>{borrow.book_title}</td>
                <td className="text-center">
                    <BorrowReturn borrow_id={borrow.borrow_id} borrow_return={borrow.borrow_return} />
                </td>
            </tr>
        );
    }
}
```

Fig. 151 : User can borrow books

6.3.14.3 Check it works

Ok, let's try.
Edit user. Check you have the borrowed books list.
Return a book. Check the list is updated.
Borrow a book. Check it appears in the list.

6.3.15 Summary

Here is a summary of the main actions we used for the project

- Create a Web Application with npm :
  npx create-react-app prwebreact
- Create directories img and css in "public", add files in these directories
- Modify "index.html" in "public" to add js and css
- install routing modules
  npm install --save react-router-dom
- Modify "App.js" to manage routes (use Routes and Route)
- Install express and pg (or your database connector)
- Create NodeJS server with server.js
  - Create nodeJS directory "server"
  - Create nodeJS structure :
    npm init
  - Add file server.js
  - Define routes
  - launch server
    npm start
- Create your components (xxx.js)
  - Include "React" and switch your component to a class that extends "Component"
  - Create a render method that returns the way you render your component
  - define your behavior methods

Things you might think to :

- 2 servers : 1 for React, 1 for nodeJS
- nodeJS server is the only one that can reach the database
- React means "single page". So, if you have several pages in you application, use Routes, Route, Navigate, ...
- Create as many components as you need
- states are components local data
- props are parameters for the component

- behavior methods may manage forms, states, nodeJS exchanges
- use Hooks when a component mays switch to another one
- do not forget to define states in the constructor. You should also bind behavior methods.
- use token cookies to manage security
- each time you change server.js, your nodeJS server file, you must restart nodeJS server.
- javascript synchronous function uses Promise. That means that the execution of a function does not end at the instruction line. You should consider including functions to manage function execution.
- **use console.log to display informations in your browser to understand what happens**

6.4 Angular and NodeJS

Angular is a Typescript framework.

We use Angular for the FrontEnd part and NodeJS for the BackEnd part. That means NodeJS ensures the link with the database, whereas Angular manages the browser part.

To understand what happens in Angular, you may consider using "console.log". When you use it, it will tell you which line is used, and you will get variables values and types.

Your NodeJS server version might be >= 18 and <= 20.
Module punycode is deprecated since version 21 and may lead to issues.

```
nvm list
nvm use ...(version)
nvm alias default ...(version)
```

6.4.1 Install Angular tools

Maybe you can have a look to

- https ://angular.dev
- https ://angular.dev/installation

First, we download the angular environment. For that we use npm commands. That explains why we install nodejs before we install angular.

```
npm install -g @angular/cli
```

-g option is not mandatory, but it will install angular for all users.

NB :

- Sometimes some protection software (like SentinelOne) prevent installation. Remove the -g option and it should be ok.
- For Windows user, PowerShell (not cmd tool) disable some script execution. Use following command :
  Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
  Read carefully the message displayed and follow instructions.

Check Angular installed tools versions, see [Fig. 152]

```
ng version
```

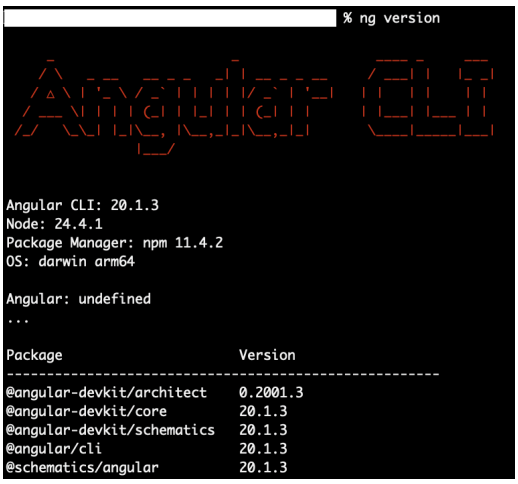


Fig. 152 : Check angular version

6.4.2 How will it work ?

Our application is made of 2 parts :

- BackEnd
  - It is managed by nodeJS. It is our “server” part.
  - **It runs in its own terminal / command tool**.
  - It deals with the databases, manage server data. It receives request and send responses back.
  - If you are not pleased with nodeJS, you can change it by any other kind of backend server. It can be, for example, a php set of scripts, a symfony application or a spring boot application.
- FrontEnd.
  - It is managed by Angular and is the “client” part.
  - This is the visible part of the project, what is displayed in the browser.
  - This part runs on any computer that connects to the application.
  - When your browser connects to the web application, it downloads the JS files of your angular application and launches it **locally** (on your computer).

That means you need 2 command tools, 1 per server you have to launch.

Never forget that, to run the application, technically you should use a computer for each element (BackEnd server, FrontEnd server, Database server). And 1 more for the browser that connects and runs a part of the angular application. In our case, we run the 4 elements on the same computer.

6.4.3 Create Angular app

6.4.3.1 Create project

First we have to create the angular project.

Use your Terminal / Command tool. Go to the parent directory of what you want to be your project location, the directory you want to put your application in.

We use the installed tools to create the project, let's call it "prwebangular". You can change your project name, but avoid space characters, and use only lowercase characters.

```
ng new prwebangular
```

This should take a few minutes. [Fig. 153] shows the result of the command. Do not panic with the warning message. It comes from differences between nodeJS and Angular versions.

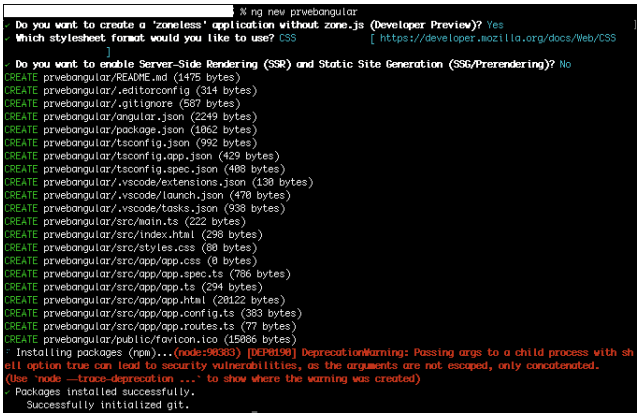


Fig. 153 : Create Angular project

[Fig. 154] shows the prwebangular directory content.

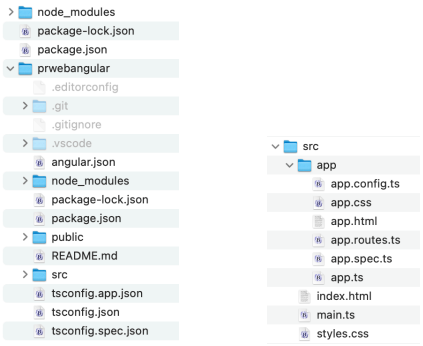


Fig. 154 : Angular directory and src directory contents

What do we find ?

- you may note GIT files / directory (hidden files starting with a dot).
- README.md, your project "README". Should be downloaded on GitHub if you use it.
- node_modules (created after the first launch) contains nodeJS modules that will be used for the project. This directory can be removed and rebuilt using the "npm update" command. Modules are uploaded according to json files.

- package-lock.json contains project and node_modules informations
- package.json contains project informations (name, modules used, ...)
- **public** directory will contain images, ...
- **src** is our Angular application
  - **app** directory contains app components
    - the app.module.ts is a Typescript file (ts=TypeScript) that manages components.
    - Each component is stored in a directory and is composed of 4 files :
      - a **css** file for the specific style
      - a **html** file for the content
      - **ts** files are Typescript files that configures the components and application
      - a **spec.ts** file configures the component
    - it contains a default component, app, with the 4 files, but it is not in a directory.
  - index.html is root file
  - tsconfig files (.app.json, .json and .spec.json) are configuration files for your app components.

6.4.3.2 First try.

In your terminal / command tool, go to the created directory (should be prwebangular).

You can check for outdated packages with "**npm outdated**" and update them with "**npm update -g**" or "**npm update**" depending on the way you installed npm.

Now, let's check everything is ok. Use one of the following commands (both will run the project, but, by default, the npm command do not launch the browser) :

```
ng serve --open
```

or

```
npm start
```

If you use the npm command, open browser with URL http ://localhost :4200
The information is in the terminal in the last generated lines.

If you want “npm start” to open your browser, you may change script in file **package.json**, line “scripts” / “start”. Add the “- -open” parameter like in [Fig. 155] (no space between the 2 “-”).

```
"scripts": {
  "ng": "ng",
  "start": "ng serve --open",
  "build": "ng build",
  "watch": "ng build --watch --configuration development",
  "test": "ng test"
},
```

Fig. 155 : modify ng serve command

Launching angular server should take some time to compile the elements, and in the terminal you should have something like in [Fig. 156]. Browser will display the default page, like in [Fig. 157].

```
% ng serve --open
Initial chunk files | Names         | Raw size
main.js             | main          | 47.78 kB |
styles.css          | styles        | 95 bytes |

                    | Initial total | 47.88 kB

Application bundle generation complete. [1.656 seconds]

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
  ➜  Local:   http://localhost:4200/
  ➜  press h + enter to show help
```

Fig. 156 : Angular compiling

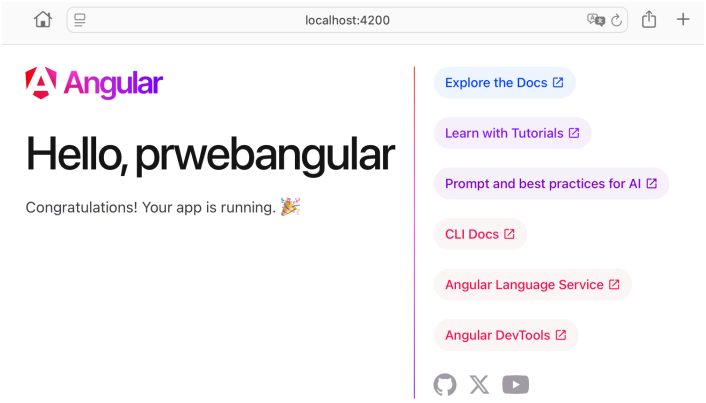


Fig. 157 : Angular default page

Stop server using CTRL-C in your terminal.

6.4.3.3 Modifying our first page

First, we will try to modify our first page to understand how it works.

Have a look to “app.html”. This is the main page, the one that is served when you start the angular server. [Fig. 158] shows its content.

Fig. 158 : Default angular page

In this file, there are :

- a **style** tag to manage local styles
- a **main** tag that displays informations
- a **router-outlet** empty tag that will manage our routes

Have a look to the page in your browser. Just after the angular line, you might have a “Hello, ...” with the name of your app.

Now, have a look around line 233 (just after the svg tag, in tag h1), the title element. Here is the way we can use parameters for a component : placing it between 2 parenthesis.

Ok, we have a parameter, but where does the title value come from ? Have a look to the file “app.ts”, the export part. You should have the answer.

Depending on the way we define the parameter, there are several ways to use it.

Replace the content of “app.html” by :

```
<main class="main">
  <p>Hello, app {{ title() }}</p>
</main>
<router-outlet />
```

Check your browser.

Now, change the title definition in app.ts by :

```
protected readonly title = 'prwebangular';
```

And in “app.html”, use {{ title }} instead of {{ title() }}.
Check your browser.

6.4.3.4 Components

In Angular, components are the elements you use in your app.

Each Component is located in a folder in src/app and consists in 4 files in this folder. If xxx is the name of your component, then folder's name is xxx, and it contains 4 files :

- xxx.css for specific css elements
- xxx.html for the html definition of the component
- xxx.ts for the component definition
- xxx.spec.ts for the component management.

Open file "app.ts" in the directory "src/app". [Fig. 159] shows the result.

```
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = 'prwebangular';
}
```

Fig. 159 : app.ts file

Have a look to @Component. Do you recognize some elements ?

- Selector : have a look to "index.html" in src.
  Do you see the tag **app-root** ?
- templateURL : have a look to app.html
  This is the html content of the displayed page
- styleUrls : have a look to the css file
- imports indicate which modules to import

Now have a look to "app.routes.ts". [Fig. 160] shows the content.
In this file, we will have to define the routes we manage and the components linked to these routes. Current file do not content any route.

```
import { Routes } from '@angular/router';

export const routes: Routes = [];
```

Fig. 160 : app.route.ts file

Do you remember the tag "router-outlet" in "app.html" ? This is the element that will manage routes. If you do not define the routes in app.route.ts, you will not be able to manage them.

6.4.4 Login page

We will start by creating a login page. And the component linked to this page.

6.4.4.1 Building a component

First, we need a component. Let's call it authenticate. To create it, use the following command :

```
ng generate component authenticate
```

When launched, you should have a result like in [Fig. 161]

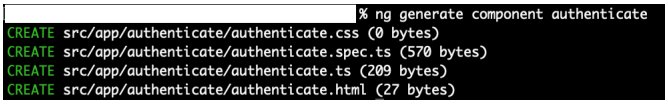


Fig. 161 : Add component "authenticate"

Now, in your src/app directory, you should have a directory "authenticate" with 4 files for the component.

Have a look to file "authenticate.ts". It is built as the app component. What do we find in the component definition ?

- selector : a way to select the component
- standalone : component can be used as a fully defined component
- imports : which modules do we have to import (the should have been imported previously)
- templateUrl : the html to use (can be a html definition or a file to use)
- styleUrl : the css to use (can be a css definition or a file to use)

[Fig. 162] shows component elements.

Fig. 162 : authenticate.ts elements

Also, do you see the line "export class", bottom of the file ? Here is the name we will use for the component.

Let's add our component and our route to the **routes**. The informations to add are in the file "app.routes.ts".

We have to import the module and declare a route that is linked to our component. To do this, have a look to [Fig. 163]

```
import { Routes } from '@angular/router';
import { Authenticate } from "./authenticate/authenticate";

export const routes: Routes = [
    { path: 'authenticate', component: Authenticate }
];
```

Fig. 163 : Add module "authenticate" to the routes in app.routes.ts

The import instruction tells where we can find module "Authenticate". Remember module name in "authenticate.ts"? Also, path to the file is given according to current file.

In the array Routes, we define the routes. To define a route, "path" indicate the path (route) to use for the url, and "component" the component to use.

Try it. Use http ://localhost :4200 then http ://localhost :4200/authenticate
What happens? You might have the content of "app.html" followed by the content of "authenticate.html".

Change route "authenticate" to route "login" in "app.routes.ts" (change "path" value). Route "authenticate" is not reachable any more, you only have the main display. Route "http ://localhost :4200/login" should work.

Let's create our login page.

We have to use the component as a starting page. For that we need to add a route for an empty path that redirects to our module. Add a new empty route in "app.routes.ts", like in [Fig 164].

```
import { Routes } from '@angular/router';
import { Authenticate } from "./authenticate/authenticate";

export const routes: Routes = [
    { path: "", redirectTo: "login", pathMatch: "full" },
    { path: 'login', component: Authenticate }
];
```

Fig. 164 : set empty path in app.routes.ts

6.4.4.2 Add login page content

Now, let's create the content of the login page. We have to define the login elements in the "authenticate" component files.

First remove everything in "app.html" except tag "router-outlet".
Next, replace content of "authenticate.html" by the content of the body content of a login page. Use the materials you downloaded from hippocampus, or what you did in the first practical works, to fill the file authenticate.html **with the body part only**. [Fig. 165] is an example of what you can build.

```
<div class="py-5">
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <h2 class="">Library Login</h2>
            </div>
        </div>
        <div class="row">
            <div class="col-md-12">
                <form>
                    <div class="form-group row">
                        <label for="inputlogin" class="col-2 col-form-label">Login</label>
                        <div class="col-10">
                            <input type="text" id="inputlogin" placeholder="login" name="login" required="required" />
                        </div>
                    </div>
                    <div class="form-group row">
                        <label for="inputpassword" class="col-2 col-form-label">Password</label>
                        <div class="col-10">
                            <input type="password" id="inputpassword" placeholder="Password" name="password" required="required" />
                        </div>
                    </div>
                    <button type="submit" class="btn btn-success">Submit</button>
                </form>
            </div>
        </div>
    </div>
</div>
```

Fig. 165 : Authenticate.html file content

In file index.html located in src, angular will replace the "app-root" tag by the current component. So in file authenticate.html we should only have the elements to be inserted in the tag body.

Ok, but what about our js files, our css files, ...
Theoretically, Angular offers some solutions like adding files in directory public, putting them in src, modifying file angular.json, but they doesn't work (some are known as issues).

In fact, the easiest way to add files is to use an old way of doing things. Create a directory "assets" in directory "public". Put directories and files you want to add in "assets".

Your public directory should be like in [Fig. 166]

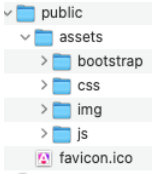


Fig. 166 : Add directories and files to public directory

And **prefix each url of images, css, js by "assets/"**.
For example, in file "index.html" located in the directory "src", you can add bootstrap and jquery to every page, you can add them located in the directory src.

[Fig. 167] shows what you can write.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Prwebangular</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">

  <link rel="stylesheet" href="assets/bootstrap/css/bootstrap.css">
  <script type="text/javascript" src="assets/js/jquery-3.3.1.min.js"></script>
  <script type="text/javascript" src="assets/bootstrap/js/bootstrap.min.js"></script>
  <link href="assets/css/main.css" type="text/css" rel="stylesheet" />
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Fig. 167 : Add assets to index.html

Let's try.
If it is not launched, launch it.
Connect to http :/localhost :4200.
You should have your login page. With colors, images, ... like [Fig. 168]

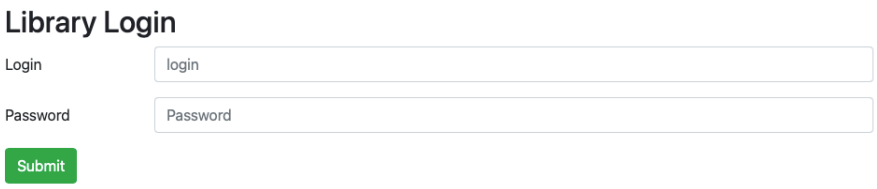


Fig. 168 : Login page

But if you try to use a login/password, you should have an error message.
Why ? Because of the informations in the form that connects to an invalid (unknown) page.
More, we have to ask Angular to manage this.

6.4.4.3 Managing authentication

Now, we have to check login / password.

We have to change the form in the component to be managed by angular, and a bit later manage authentication with from the backend. Technically we have to modify authenticate.html and authenticate.ts to manage the informations. Also, we have to create a service that manages the authentication. It will be connected to the backend later.

So, first we modify the html part of the component. We have to change the way we display / manage the form, and the component description in authenticate.html. We use these angular elements :

- **ngForm** defines a reference to a form
- **ngSubmit** ask angular to manage the form.
- **ngModel** define a field in a form

Change the form part in “authenticate.html” by the one in [Fig. 169].

```
<form #loginForm="ngForm" (ngSubmit)="authenticateUser(loginForm)">
    <div class="form-group row">
        <label for="inputlogin" class="col-2 col-form-label">Login</label>
        <div class="col-10">
            <input type="text" class="form-control" id="inputlogin"
            placeholder="login" name="login" required="required" ngModel />
        </div>
    </div>
    <div class="form-group row">
        <label for="inputpassword" class="col-2 col-form-label">Password</label>
        <div class="col-10">
            <input type="password" class="form-control" id="inputpassword"
            placeholder="Password" name="password" required="required" ngModel />
        </div>
    </div>
    <button type="submit" class="btn btn-success">Submit</button>
</form>
```

Fig. 169 : HTML form to authenticate

Next, in “authenticate.ts”, we have to tell the component we use the Form elements, and that we call authenticateUser(...). In the “imports” section, we have to tell what we need. [Fig. 170] shows what we have to include.

```
import { Component } from '@angular/core';
import { NgForm, FormsModule } from '@angular/forms';
import { Router } from '@angular/router';

@Component({
  selector: 'app-authenticate',
  imports: [FormsModule],
  templateUrl: './authenticate.html',
  styleUrl: './authenticate.css'
})
export class Authenticate {
```

Fig. 170 : authenticate component modification to include modules

Next, we have to define the scripts to manage the component informations.
The elements are defined in the ts part of the component, so in our case in authenticate.ts

We will need a service to authenticate : authenticateservice. We will see that later.

We have to create a function authenticateUser which is called when the form is submitted. We have to retrieve values and check if user authenticates. And we might not forget to import angular modules.

In “authenticate.ts” we have to define the elements we need :

- define a constructor. That will define the elements we need.
- we have to be sure it is generated for all browsers and when it should be, we also extend component OnInit (required in import too).
  That means we have to define a function **ngOnInit**, that do the same as the constructor.
- we define a function authenticateUser (remember ngSubmit). We retrieve values from the Form. We call a function checkAuthenticate in the service part to check login/password (we will define it in authenticateservice).
- we manage the result.

[Fig. 171] shows the way to do this in authenticate.ts.

```
import { Component, OnInit } from '@angular/core';
import { NgForm, FormsModule } from '@angular/forms';
import { Authenticateservice } from "../authenticateservice";
import { Router } from '@angular/router';

@Component({
  selector: 'app-authenticate',
  imports: [FormsModule],
  templateUrl: './authenticate.html',
  styleUrl: './authenticate.css'
})
export class Authenticate implements OnInit {

  constructor( private _authenticate: Authenticateservice , private router: Router) {
  }

  ngOnInit() {
  }

  authenticateUser(form: NgForm): void {
    var loginValue = form.value.login;
    var passwdValue = form.value.password;
    var res = JSON.parse(this._authenticate.checkAuthenticate(loginValue, passwdValue));
    var ok = res.ok;
    if (ok == 1) {
      // DO SOMETHING
    }
  }
}
```

Fig. 171 : new authenticate.ts content

What did we do ?

- We added import of module OnInit,
- we added import of service authenticateservice (functions not defined yet)
- We added required elements in array imports[]
- we tell the component implements OnInit
- we define the constructor and the method onNgInit
- we define parameters in constructor that implement local attributes (_authenticate, router). **note the way we can initialise a parameter in a constructor by defining it in the parameters**.
- we get data from the form (remeber ngModel) with the corresponding elements in the html part of the component.
- we implement an authenticateUser method

Next, define authenticate service.
Here is the way you can create the service “authenticateservice” :

```
ng generate service authenticateservice
```

This instruction generates 2 files : “authenticateservice.ts” and “authenticateservice.spec.ts”.
We use this service to manage authentication. Component “authenticate” is used to exchange with the browser and define what has to be displayed.

In file “authenticateservice.ts” (in src/app), we must define a first version of the authentication method checkAuthenticate. Implement it like in [Fig 172]

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class Authenticateservice {
  checkAuthenticate(login: string, password: string): string {
    var result = JSON.stringify({"ok": 0});
    if ((login === "admin") && (password === "admin")) {
      result = JSON.stringify({"ok": 1});
    }
    return result;
  }
}
```

Fig. 172 : First authentication function

Method checkAuthenticate has 2 parameters : login and password. The method replies with a JSON element {ok : xxx} that tells if it is ok or not.

Maybe, before you try it, you should add something in “authenticate.ts” so that it displays the content of variable res. Something like :

```
console.log(res);
```

You should add when you get variable res, after the call to checkAuthenticate to see the result.
NB : **this script is executed in your browser, so the console is the one in your browser**.

Now, let’s try in our browser.

Try to connect with (admin, admin) and with any other couple of values (login, password). Use your browser debugger and have a look to the console. You should have the result according to your data.

Questions

What is the app-root tag used for in index.html ?
What is app.routes.ts used for ?
What are the 4 files that define a component ?
How do you manage forms and elements in forms ?
What is a service used for ?
In a component, when you use console.log in the component.ts, where is displayed the message ? Why ?

6.4.5 BackEnd server with NodeJS

Now, we need something to really manage authentication, and connection to the database : a BackEnd server.

As a BackEnd server, we will use nodeJS.

NodeJS is a Javascript/Typescript server.
So we will have to use Javascript (TypeScript) to build the scripts that will manage requests.

6.4.5.1 Creating server

Create a directory “server” in the same directory as “prwebangular”.

Go inside this directory with a Terminal / Command tool.
We will initialise the node server files.

```
npm init
```

Validate each question for which you are asked for an answer (default answer is ok). Of course, you can change default answer if you want to, but be sure of what you do. Maybe you can call your server file server.js instead of index.js, both will work.

That creates a file “package.json” that contains server configuration elements. Main informations, especially to run the server are in this file.

Now, we need 2 modules :

- pg to connect to the postgresql server
- express to manage some server elements for us.

Use following commands :

```
npm install express
npm install pg
```

This will create files “package.json” and “package-lock.json”, and a directory “node_modules” and install the requirements.

Now, we need the server script.
Create a file with the name you gave previously (might be index.js or server.js, or whatever you chose). If you do not remember the file name, have a look to package.json, the attribute “main”.

The server file will contain our BackEnd scripts and will be used by nodeJS. We have left an initial version of the server in the materials (called server.js).

[Fig. 173] shows the content of the file.

```
const express = require('express');
const pg = require("pg");
const app = express();

var conString = 'postgres://prweb:prweb@localhost:5432/prweb';

app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Default to accept all requests
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  next();
});

// Must be LAST instruction of the file
// Listen to port 8000
app.listen(8000, () => {
  console.log('Server started!')
});
```

Fig. 173 : NodeJS server script in server.js

What do we find in the server file ?

- 2 first lines are library inclusion. express to manage some elements, pg for postgresql.
- line 3 is used to tell we have an app that uses express.
  We manage nodeJS routes through “app”.
- line 5 with conString tells what will be our connection URL for the database (will be used later) :
  - protocol is postgresql,
  - user connects with login prweb and password prweb,
  - postgresql server is in localhost on port 5432,
  - database is prweb
- 2 next lines (app.use ...) explain that the parser should be able to encode responses, and should be able to encode data as JSON data.
- next line explains we don’t care where the request comes from (nodeJS do not manage Angular application, but might answer requests coming from it, but not on the same port). Technically this is called a cross-platform call. These instructions will allow NodeJS to reply even if the call comes from somewhere else. We use “next” to tell we go to the next valid “use”.
- last lines tell the server to listen to port 8000
  This instruction **MUST ALWAYS BE THE LAST ONE**.

OK, let’s try it. Open a new terminal / Command tools to launch the NodeJS server.

```
npm start
```

Check it compiles and there is no error when launched.
You can stop it with CTRL-C.

Check it works with http ://localhost :8000.
As there is no route to reply to, you should have a message that tells the server can't handle the URL.

6.4.5.2 First function in NodeJS

In your server file, add a first function after app.use and before app.listen, like in [Fig. 174].

```
// Default to accept all requests
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  next();
});

/**
 * Only to check it works
 */
app.get("/", function(req, res) {
  var jsonString;
  jsonString = JSON.stringify({ok:1});

  res.setHeader('Content-Type', 'application/json');
  res.send(jsonString);
});


// Must be LAST instruction of the file
// Listen to port 8000
app.listen(8000, () => {
```

Fig. 174 : First function in NodeJS

Stop the node server (CTRL-C to stop it) and relaunch it.

Once launched, the server does not check if you modify the file that contains your script. So, **each time you modify the script file, you will have to stop server and relaunch it**.

Now, you should have a JSON response.

6.4.5.3 Writing authentication function

Right now, our server does nothing.

Let's write a first replier for our server. We have a login page and we would like the server to identify our user. So we start by defining a function in our server that manages authentication.

To reply to a request, we use express, and the variable app we defined.

- **app.get** is used for a GET request
- **app.post** is used for a POST request

Both of these functions have 2 parameters :

- the route we manage
- a function that manages the request and the response for this route.

To get parameters, if req is the parameter that represents the request, then

- **req.query** contains GET parameters
- **req.body** contains POST parameters

So, what do we have to do in our function ?

- get parameters (login and password)
- check (login / password) is ok, and give the answer as a reply. We will use "JSON.stringify" that translates a JSON object to a JSON string, so that we send it back.
- send the response as a JSON string.

[Fig. 175] shows how we can implement such a function in the script file. Our route is "/authenticate". Remember, this function has to be placed **before** the "listen" part. Currently we implement it for a GET request to be able to test it.

```
app.get("/authenticate", function(req, res) {
  var login = req.query.login;
  var passwd = req.query.passwd;

  var jsonString;
  if ((login === "admin") && (passwd === "admin")) {
    jsonString = JSON.stringify({ok:1});
  } else {
    jsonString = JSON.stringify({ok:0});
  }
  res.setHeader('Content-Type', 'application/json');
  res.send(jsonString);
});
```

Fig. 175 : NodeJS server response to an authentication

Stop the node server and relaunch it.

With your browser, check it works with that URL :
http ://localhost :8000/authenticate ?login=admin&passwd=admin

Do you have a response that tells ok is 1 ?
Ok, now change login and/or password.
What is the response ?

Now, let's change the script to POST :

- change app.get to app.post
- change req.query to req.body

You may not be able to use the script with your browser because, now, it is in POST mode.

6.4.5.4 Checking Login Password from the nodeJS server

As our Angular application will often interact with our nodeJS server, maybe it should be a good idea to write a function that manages this interaction. We will ensure our function exchange with a POST method so that we can protect our nodeJS server from unwanted requests.

This function will have 2 parameters : the route we call in the nodeJS server, and data to be send as a JSON object.

We will use the "fetch" function that can send a request to a server.
"fetch" is an **asynchronous** function, so we will have to manage its response with the right method.

The response from the nodeJS server is a string that contains a JSON string.
So first, we will have to change the string response to an object.

Next, our response is a **Promise**, a specific JS object that tells "I will give you the result when it will be filled". A promise is often linked to asynchronous functions. So, never consider a JS instruction is ended when you go to the next one. If it is an asynchronous function, call is launched and a background function is launched and in charge of managing the result. That means the instruction is ended but the result is not arrived yet. So, if you try to use the Promised result and is not filled yet, you process with undefined data.

That means you will often have to take care of the way you manage data when coming from another source. And for all asynchronous functions.

Let's do it with the service.

To call NodeJS from Angular, we use the service.
We use a http request to nodeJS, so we have to declare it.

Calling http request is asynchronous, therefore, the answer we can give is not completed. We can only give back an object that will tell our caller that a response will arrive soon. in TypeScript, the kind of object that manages that is an **Observable**.

The **Observable** response have to be managed by the component.

So, as a service, we only have to send the HTTP request to nodeJS and send back the Observable to the component.

We have to import HttpClient, declare an attribute in the constructor to build it, and call the nodeJS server with the appropriate data. Now, our method checkAuthenticate should tell it returns an Observable. That means we have to include the module and set checkAuthenticate return type.
[Fig. 176] shows how we can implement it in authenticateservice.js.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from "rxjs";

@Injectable({
  providedIn: 'root'
})
export class Authenticateservice {
  private _serverURL = 'http://localhost:8000';
  constructor(private _httpClient: HttpClient) { }
  checkAuthenticate(login: string, password: string): Observable<any> {
    var theObject = {"login": login, "passwd": password};
    var result = this._httpClient.post(this._serverURL+"/authenticate", theObject);
    return result;
  }
}
```

Fig. 176 : Auth service calls nodeJS backend server

As we use HttpClient, we also have to declare it.
In “app.config.ts”, add import for provideHttpClient like in [Fig. 177]

```
import { ApplicationConfig, provideBrowserGlobalErrorListeners, provideZonelessChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideZonelessChangeDetection(),
    provideRouter(routes),
    provideHttpClient()
  ]
};
```

Fig. 177 : Add HttpClient module to app.config.ts

Our authenticate service returns an Observable.

An Observable manages the response when it is (will be) available.
Method "subscribe" used on an Observable uses a callBack function that manages the event "response is available" and let us manage this response.

[Fig. 178] shows how we can implement it in authenticate.js.

```
authenticateUser(form: NgForm): void {
  var loginValue = form.value.login;
  var passwdValue = form.value.password;
  this._authenticate.checkAuthenticate(loginValue, passwdValue)
  .subscribe(
    (value) => {
      var ok = value.ok;
      if (ok == 1) {
        // DO SOMETHING
        console.log("Switching");
      }
    }
 );
}
```

Fig. 178 : Auth component uses Observable

You do not have to relaunch angular : each time you change a source file, it is recompiled.

Check in your terminal, that your code is ok.
Refresh your browser with the login page and check it works.
Have a look to the debugging console in the browser to check the answer.

Questions

What is a BackEnd server used for ?
Why do you use method subscribe to get a result from BackEnd server ?
Why do you have to restart NodeJS server when you modify server.js ?

6.4.6 Application pages

Now we will deal with data. We need :

- A page that displays users
- A page that displays 1 user
- A page that displays books
- A page that displays 1 book
- managing 'borrowing books' by users

For our 4 pages, we need 4 components.
Maybe 1 service for users and 1 service for books would be sufficient.

Typescript can also manage objects types.
By default, we can use "**any**" to tell typescript we manage an object that we don't want to define.
But shouldn't it be better to manage a **Person** class and a **Book** class defined as the ones in database ?
Maybe for Borrow too.

Oh, maybe we should link the authentication page to users page or to books page.
And navigating through the nav bar.

6.4.7 Managing people.

6.4.7.1 Creating pages

Create a users component to manage the list of users, a user component to manage 1 user and a user service.

```
ng generate component users
ng generate component user
ng generate service userservice
```

That should create 2 folders with the components (users and user) and 2 files (userservice.ts and userservice.spec.ts).

Now, create a person class :

```
ng generate class person
```

That should create 2 files (person.ts and person.spec.ts).

Add Person description in "person.ts" like in [Fig. 179].
And manage specifications in "person.spec.ts" like in [Fig. 180].

```
export class Person {
  person_id: Number;
  person_firstname: String;
  person_lastname: String;
  person_birthdate: Date;

  constructor(person_id: Number, person_firstname: String, person_lastname: String, person_birthdate: Date) {
    this.person_id = person_id;
    this.person_firstname = person_firstname;
    this.person_lastname = person_lastname;
    this.person_birthdate = person_birthdate;
  }
}
```

Fig. 179 : class Person in person.ts

```
import { Person } from './person';

describe('Person', () => {
  it('should create an instance', () => {
    expect(new Person(1, "firstname", "lastname", new Date())).toBeTruthy();
  });
});
```

Fig. 180 : Person spec in person.spec.ts

Copy the page with the list of people from your first script and place it in “users/users.html”.
Copy the person page that manage a user and place it in the “user/user.html”.

Add routes in “app.routes.ts” like in [Fig. 181].

```
import { Routes } from '@angular/router';
import { Authenticate } from "./authenticate/authenticate";
import { Users } from './users/users';

export const routes: Routes = [
    { path: "", redirectTo: "login", pathMatch: "full" },
    { path: 'login', component: Authenticate },
    { path: 'users', component: Users }
];
```

Fig. 181 : Add route for users

You can check it works with your browser using http ://localhost :4200/users

6.4.7.2 Routing to users from authentication

Now, let’s try to use users route when authentication is ok.

The file where we authenticate users is “authenticate.ts”. Do you remember that we checked login/password but we did nothing when it is ok ? This is where we have to add something.

We need a Router (already defined in constructor), to import the Router Module, and route to “users”.

The Router is a component that manages routes. So let's use it and navigate to another component. [Fig. 182] shows how we can implement it in "authenticate.js".

```
import { Component, OnInit } from '@angular/core';
import { NgForm, FormsModule } from '@angular/forms';
import { Authenticateservice } from "../authenticateservice";
import { Router, RouterModule } from '@angular/router';

@Component({
  selector: 'app-authenticate',
  imports: [FormsModule, RouterModule],
  templateUrl: './authenticate.html',
  styleUrl: './authenticate.css'
})
export class Authenticate implements OnInit {

  constructor( private _authenticate: Authenticateservice , private router: Router) {
  }

  ngOnInit() {
  }

  authenticateUser(form: NgForm): void {
    var loginValue = form.value.login;
    var passwdValue = form.value.password;
    this._authenticate.checkAuthenticate(loginValue, passwdValue)
    .subscribe(
      (value) => {
        var ok = value.ok;
        if (ok == 1) {
          // DO SOMETHING
          this.router.navigate(["users"]);
        }
      }
    );
  }
}
```

Fig. 182 : Set route from authenticate to users

Now, restart to your login page.
Give right login / password.
Now, you should have been switched to users page.

6.4.7.3 Managing data

We have to load data from the database.

When component is loaded, we have to ask data we need from the BackEnd.
Relation between FrontEnd and BackEnd is ensured by the service.
Component initialisation is in the file users.ts
The html part must be changed to take into account angular data management.

6.4.7.3.1 BackEnd response

Let's start with the backend.

We have to get data from table person and send an array with the lines. As we will need very often to execute a request and send back a list of values, maybe we could build a function that manages a query and its parameters and that send a JSON object as a response.

[Fig. 183] shows how we can implement it in the server. We create a connection to the database server. If it is ok, we launch the request with its parameters. Then we build the result as a JSON string and send it back as a response.

```
function getSQLResult(req, res, hasReturnedValue, sqlRequest, values) {
  var client = new pg.Client(conString);
  client.connect(function (err) {
    if (err) {
      // Cannot connect
      console.error('cannot connect to postgres', err);
      res.status(500).end('Database connection error!');
    } else {
      // Connection is OK
      client.query(sqlRequest, values, function (err, result) {
        if (err) {
          // Request fails
          console.error('bad request', err);
          res.status(500).end('Bad request error!');
        } else {
          // Build result array from SQL result rows
          if (hasReturnedValue) {
            var results = [];
            for (var ind in result.rows) {
              results.push(result.rows[ind]);
            }
          } else {
            var results = {ok: 1};
          }
          // Convert object to a JSON string and send it back
          res.setHeader('Content-Type', 'application/json');
          res.send(JSON.stringify(results));
        }
        client.end();
      });
    }
  });
}
```

Fig. 183 : BackEnd query function

Then we have to build the function that handles the route to NodeJS and that calls our function.
It may look like the one in [Fig. 184]. If you have an error, maybe you didn't import the pg module, or you didn't stop the server and relaunch it.

```
app.post("/users", function (req, res) {
  var sqlRequest = "SELECT * FROM Person ORDER BY Person_LastName, Person_FirstName";
  var values = [];
  getSQLResult(req, res, true, sqlRequest, values);
});
```

Fig. 184 : BackEnd find all people

We built the function as a POST request, but of course,you can add a GET request to check it works.

6.4.7.3.2 User service and component

Our “user” service have to call the backend. So it is quite similar to service “authenticateservice”.

In “userservice.ts”, we define a method “findAll” with no parameters.
The method returns an Observable <Person >.

We call the backend route “/users” that we already defined.

We have to import HttpClient, Observable and Person.
Also, do you remember that a constructor can initialise the HttpClient ?
It may look like the one in [Fig. 185]

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from "rxjs";

@Injectable({
  providedIn: 'root'
})
export class Userservice {
  private _serverURL = 'http://localhost:8000';
  constructor(private _httpClient: HttpClient) { }

  findAll(): Observable<any> {
    var theObject = {};
    return this._httpClient.post(this._serverURL+"/users", theObject);
  }
}
```

Fig. 185 : User service to find people

In our “Users” component, we have to declare UserService and initialise a connector to the service in the constructor to define it. Maybe you can add the router too, because we will need it a bit later.

Next, we need an object that keeps the list of people. Lets call it “users”.
We have to define users as an attribute, initialize it in the constructor, and get its value in ngOnInit.

As we will ask angular to iterate on this object, our attribute might be a **NgIterable** object.

We have to import NgIterable from angular/core and CommonModule from angular/common.
NgIterable is for the NgIterable object.
CommonModule offers tools to use NgIterable objects.

[Fig. 186] shows a way to build it.
The data we get from the service is stored into our users object.

```ts
import { Component, OnInit, NgIterable, ChangeDetectorRef } from '@angular/core';
import { NgForm, FormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
import { Userservice } from "../userservice";
import { Router, RouterModule } from '@angular/router';

import { Person } from '../person';

@Component({
  selector: 'app-users',
  imports: [FormsModule, RouterModule, CommonModule],
  templateUrl: './users.html',
  styleUrl: './users.css'
})
export class Users implements OnInit {
  public users: Person[];

  constructor( private _user: Userservice , private router: Router, private changeDetector: ChangeDetectorRef) {
    this.users = new Array();
  }

  loadData():void {
    this._user.findAll()
    .subscribe(
        (data) => {
          this.users = data;
          this.changeDetector.markForCheck();
        }
    );
  }

  ngOnInit() {
    this.loadData();
  }
}
```

Fig. 186 : Users component users.ts

Now that have loaded users, we have to display them in the html part.

In the Typescript part, we defined an attribute users.

In the HTML part, we have to create a loop on this array to display the lines.
Angular uses "*ngFor" to create a loop. When we use "ngFor", we define which array we iterate on, and the iteration object.
The attribute value looks like "**let** variable **of** iterableObject".
In our case, the iterableObject is users (in users.ts). We use variable "user".

To get the object value, we use double braces. ngFor is a angular/HTML attribute that should be placed in the tag you iterate on, "TR" for us. We also need to format date.

[Fig. 187] shows a way to build it. We do not manage buttons on each line yet.

```html
<tr *ngFor="let user of users">
    <td scope="col">{{ user.person_id }}</td>
    <td>{{ user.person_firstname }}</td>
    <td>{{ user.person_lastname }}</td>
    <td>{{ user.person_birthdate | date: 'dd/MM/yyyy' }}</td>
    <td class="text-center">
        <button name="edit" class="btn"><img src="assets/img/edit.png" alt="edit" class="icon" /></button>
        <button name="delete" class="btn" ><img src="assets/img/delete.png" alt="delete" class="icon" /></button>
    </td>
</tr>
```

Fig. 187 : Angular loop to display people in users.html

Have a look to your browser, the list of people should be the one in your database.

6.4.8 Managing a user

Now, we have to edit 1 user.

We have to switch from the users list to the page that edit 1 user.
We could manage it through a form, but in our case, it is not required because we manage components with the browser. So, when we click on a button, we only have to launch a method that switches to component user.
The route might be called with the user's ID.

In component user, we have to retrieve the ID, load the corresponding person and display loaded data. Data should be retrieved from BackEnd server.

6.4.8.1 Get person from BackEnd

Let's start with BackEnd server.

We need a post method that retrieves a person from the ID. We only have to process the request with the ID value as a parameter. The method getSQLResult should send the result.

[Fig. 188] shows a way to retrieve 1 person with the ID. Route is "/user".

```
app.post("/user", function (req, res) {
  var id = req.body.id;
  var sqlRequest = "SELECT * FROM Person WHERE person_id=$1";
  var values = [id];
  getSQLResult(req, res, true, sqlRequest, values);
});
```

Fig. 188 : BackEnd loads 1 user in server.js

6.4.8.2 Load data in user component

Then, we need the service user to call the server to retrieve the value.

Add a method in userservice.ts that calls route /user with the id as a parameter as in [Fig. 189].

```
getUser(id: number): Observable<any> {
  var theObject = {"id": id};
  return this._httpClient.post(this._serverURL+"/user", theObject);
}
```

Fig. 189 : FrontEnd loads 1 user in userservice.ts

Next, we have to load this from user.ts when angular starts the page.
For that, we need another component of angular/router : "ActivatedRoute".

This allows us to retrieve parameters from the URL.
We have to initialise the ActivatedRoute in the constructor, and in the method ngOnInit, to retrieve the ID value and load user from the user service.

[Fig. 190] shows a way to get the ID from the URL and retrieve user from the service. Remember that the server (through method getSQLResult) send us back a list of something, so the user is the first one in the array (index 0).

```
import { Component, OnInit, NgIterable,ChangeDetectorRef } from '@angular/core';
import { NgForm, FormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
import { Userservice } from "../userservice";
import { Router, ActivatedRoute, RouterModule } from '@angular/router';

import { Person } from '../person';

@Component({
  selector: 'app-user',
  imports: [FormsModule, RouterModule, CommonModule],
  templateUrl: './user.html',
  styleUrl: './user.css'
})
export class User implements OnInit {
  user: Person;

  constructor( private _user: Userservice , private router: Router, private route: ActivatedRoute, private changeDetector: ChangeDetectorRef) {
    this.user = new Person(-1, "", "", new Date());
  }

  ngOnInit() {
    this.route.paramMap.subscribe(
      (value) => {
        var personId = Number(value.get('id'));
        if (personId > 0) {
          this._user.getUser(personId)
          .subscribe(
            (data) => {
              this.user = data[0];
              this.changeDetector.markForCheck();
            }
          );
        } else {
          this.user = {person_id: -1, person_firstname: "", person_lastname: "", person_birthdate: new Date()};
        }
      }
    );
  }
}
```

Fig. 190 : FrontEnd loads 1 user in user.ts

6.4.8.3 Display user data in user component

Our html file for component user displays 1 person.

Our component knows attribute “user”. So we can change the file user.html to display user “user” as {{ user.xxx }} to get user data. You only have to take care that input type date uses “YYYY-MM-dd” format for data.

[Fig. 191] shows haw you can set it.

```
<tr>
    <th scope="col">user #</th>
    <td>{{ user.person_id }}
        <input type="hidden" class="form-control" name="personId" id="personId"
        value="{{ user.person_id }}"/></td>
</tr>
<tr>
    <th scope="col">FirstName</th>
    <td><input type="text" class="form-control" name="FirstName" id="FirstName"
        value="{{ user.person_firstname }}"/></td>
</tr>
<tr>
    <th scope="col">LastName</th>
    <td><input type="text" class="form-control" name="LastName" id="LastName"
        value="{{ user.person_lastname }}"/></td>
</tr>
<tr>
    <th scope="col">Birthdate</th>
    <td><input type="text" class="form-control" name="Birthdate" id="Birthdate"
        value="{{ user.person_birthdate | date: 'dd/MM/yyyy' }}"/></td>
</tr>
```

Fig. 191 : FrontEnd html script for 1 user in user.html

6.4.8.4 Link component Users to component User

Now we have to link the users page to the user page when we want to edit 1 user.

For that, we only have to use the angular function **(click)** as an attribute in "button" to explain want we want to do and add a method in the typescript part to switch to user. Our function is "onEdit" and has 2 parameters : the event (the click) and the user we want to edit. Change your file users.html to the one in [Fig. 192]. Important part is in the red rectangle.

If you kept the form, we do not need it any more, so you can remove it.

```
        <td>{{ user.person_lastname }}</td>
        <td>{{ user.person_birthdate | date: 'dd/MM/yyyy' }}</td>
        <td class="text-center">
            <button name="edit" class="btn" (click)="onEdit($event, user)"><img src="assets/img/edit.png" alt="edit" class="icon" /></button>
            <button name="delete" class="btn" ><img src="assets/img/delete.png" alt="delete" class="icon" /></button>
        </td>
    </tr>
</tbody>
```

Fig. 192 : Button switches to edit 1 user in users.html

Next, we have to add the method onEdit in the typescript part of the component of users. As we did to navigate from the authentication page to the users page, we only have to navigate from users to user. There is a little difference, we have to send user id to edit user.

[Fig. 193] shows how you can add a parameter to navigate.

```
onEdit(event:Event, user:any) {
  var userid = user.person_id;
  this.router.navigate(["user", {id: userid }]);
}
```

Fig. 193 : onEdit switches from Users tu User in users.ts

6.4.8.5 Try it

Come back to your identification page. Authenticate. You should have the users list. Click on the button pencil near an user, you should switch to the edition page of that user.

If data are not displayed correctly, maybe you should have a look to the html file for user.

6.4.8.6 Saving user

Ok. Let's save an user.

We need :

- In the FrontEnd :
  - to add a button to save user in the user component
  - to add a call to the BackEnd to save user, through the userservice
- In the BackEnd :
  - to build a function that saves user in database

6.4.8.6.1 BackEnd part

First our NodeJS file "server.js". We have to define a method to manage a route "saveUser" in POST mode. This method retrieves values from a JSON input and launches the insert / update user. [Fig. 194] shows a way you can manage it.

```
app.post("/saveUser", function(req, res) {
    var person_id = req.body.person_id;
    var person_firstname = req.body.person_firstname;
    var person_lastname = req.body.person_lastname;
    var person_birthdate = req.body.person_birthdate;

    var sqlRequest = "";
    var values = [];
    // We build a request that returns ID value to be able to returns its value (Person_ID)
    if (person_id < 0) {
        sqlRequest = "INSERT INTO Person(Person_FirstName, Person_LastName, Person_BirthDate)"
        + " VALUES ($1, $2, $3)";
        + " RETURNING Person_ID";
        values = [person_firstname, person_lastname, person_birthdate];
    } else {
        sqlRequest = "UPDATE Person SET"
        + " Person_FirstName=$1, Person_LastName=$2, Person_BirthDate=$3"
        + " WHERE Person_ID=$4"
        + " RETURNING Person_ID";
        values = [person_firstname, person_lastname, person_birthdate, person_id];
    }
    getSQLResult(req, res, false, sqlRequest, values);
});
```

Fig. 194 : BackEnd methods saveUser in server.js

6.4.8.6.2 FrontEnd part

One of the purpose of editing an user is to be able to change values.

Create a form around the table in user.html
We need to manage ngSubmit in the form to redirect to a method saveUser.
We need a FormGroup to manage data.

Have a look to [Fig. 195] shows a way you can manage it.

```
<form (ngSubmit)="saveUser()" [formGroup]="userData">
    <table class="table table-striped">
    <tbody>
        <tr>
            <th scope="col">user #</th>
            <td>{{ user.person_id }}
                <input type="hidden" class="form-control" name="personId" id="personId"
                value="{{ user.person_id }}"
                formControlName="personId" ></td>
        </tr>
        <tr>
            <th scope="col">FirstName</th>
            <td><input type="text" class="form-control" name="FirstName" id="FirstName"
                value="{{ user.person_firstname }}"
                formControlName="FirstName" /></td>
        </tr>
        <tr>
            <th scope="col">LastName</th>
            <td><input type="text" class="form-control" name="LastName" id="LastName"
                value="{{ user.person_lastname }}"
                formControlName="LastName" /></td>
        </tr>
        <tr>
            <th scope="col">Birthdate</th>
            <td><input type="text" class="form-control" name="Birthdate" id="Birthdate"
                value="{{ user.person_birthdate | date: 'dd/MM/yyyy' }}"
                formControlName="Birthdate" /></td>
        </tr>
    </tbody>
    <tfoot>
        <tr>
            <td scope="col" colspan="2" class="text-center"><button type="submit" class="btn btn-block btn-primary">Save</button></td>
        </tr>
    </tfoot>
    </table>
</form>
```

Fig. 195 : Frontend script inn user.html

Next, we have to add some importation of modulesj and write some scripts in user.ts

- Add FormGroup, FormControl, FormsModule, ReactiveFormsModule import from @angular/forms,
- Add FormsModule, ReactiveFormsModule in the imports of the component,
- Add attribute “userData” to the attributes of the component,
- Add “userData” definition in the constructor, and when we get data
- Create method saveUser that calls a service that saves user

[Fig. 196] show the way you have to change user.ts

Fig. 196 : Add elements to manage FormGroup and saveUser in user.ts

6.4.8.6.3 FrontEnd service part

Next, add a method “saveUser” in userservice.ts

This method creates a JSON object with the parameters and calls route “/saveUser” in POST mode on our NodeJS Server. [Fig. 197] show the method saveUser in userservice.ts

```
saveUser(IDValue: number, FirstName: string, LastName: string, Birthdate: string): Observable<any> {
  var theObject = {"person_id": IDValue, "person_firstname": FirstName, "person_lastname": LastName, "person_birthdate" :Birthdate};
  return this._httpClient.post(this._serverURL+"/saveUser", theObject);
}
```

Fig. 197 : saveuser service in userservice.ts

6.4.8.6.4 Try it

Restart your NodeJS server. As we modified it, you have to restart it.

With your browser, restart to the authentication page. Authenticate with the right login/password. You should get the list of users. Edit any user. Change something. Save. The user should have been modified.

6.4.8.7 Adding user.

Next, the add button. We know how to save an existing user, but we also have to save a new user.

Add a (click) attribute to the “add” button in the HTML users component part. This will call a method “onAdd” in the TS part. Use the event as a parameter. You do not need any other parameter.

In the TS users component part, create a method onAdd that navigates to user. Set the user id to -1. And navigate to user.

In the TS user component part, ensure that, in ngOnInit, user is correctly initialized.

In the NodeJS server file, ensure route “/saveUser” launches an update request when ID is positive, and an INSERT request when ID is negative or equal to 0.

Ok, let’s try. Click on the add button, give first name and last name, and a birthdate. Save. Your list should have 1 more user.

If you have any problem when displaying the birthdate, you can change type to “date” in the corresponding input in user.html

6.4.8.8 Deleting user

Next, the delete button (the trash button for each line in users.html).

Add a (click) attribute to the delete button in the HTML part of users component. This will call a method "onDelete" in the TS part. Maybe you should give the event and the user as parameters.

In the TS part of users component, create a method "onDelete" that retrieves the ID and sends it to a method "deleteUser" in the user service element. If it is ok, we only have to reload the component with "this.ngOnInit() ;". That will also ensure that if someone else added a user, we have the new user in the list.

In the user service file, create a method "deleteUser()" that calls route "/deleteUser" in NodeJS Server.

In the NodeJS server file, manage route "/deleteUser". Retrieve the ID parameter and execute a SQL request that deletes the corresponding person.

Let's try.
Remove the user you previously added.

6.4.9 Navigating

We defined a navbar in the first practical work. Maybe we could define it as a component and use it in our pages.

Create new component "navbar" (ng generate component ...).

Copy the html part of the navbar of the first practical work and put it into navbar.html.
Add a (click) attribute to the "a" tag and link it to a method as you did for the buttons. Remove attribute href. Link "Users" to method "navUsers" and "Books" to method "navBooks". Use event as a parameter for the 2 methods.

in navbar.ts, create methods "navUsers" and "navBooks". For "navUsers", navigate to "Users". Keep "navBooks" void, or with a comment to the navigation to "Books". Do not forget to import Router and RouterModule, add RouterModule to imports in @Component. And at last, declare a Router in the constructor.

Add "navbar" in the imports of "users.ts" and in the imports of the @Component, like in [Fig. 198].

Fig. 198 : add navbar to users.ts

Now, we have to call component Navbar in users.html
Have a look to navbar.ts, the "@component" part. Do you see the line "selector"? It defines how we can call this component.
Add '<app-navbar ></app-navbar >at the beginning of the file "users.html", like in [Fig. 199].

Fig. 199 : add navbar to users.html

6.4.10 Books and book.

Create a class for books, let's call it book (ng generate class ...). Fill it as you did for Person. You only have to manage book_id, book_title and book_authors.

Create components books and book (ng generate component ...).
Fill books.html with the file books.html you created in the first part of the practical work. Do the same for book.html and book.html.
Fill books and book HTML and TS components files according to what you did for user.

Create service for books, let's call it bookservice (ng generate service ...). Fill bookservice as you did for userservice.

Update NodeJS server to manage books. You have to implement methods for route books, book, saveBook, deleteBook.

Add the link to navigate to books in your component navbar (in the html and in the ts parts).

Do not forget to add routes in app.routes.ts

Try what you did.

Check you can switch from users to books. You should have the list of books.

You might be able to edit a book, change any value, add a book, and remove a book.

6.4.11 Borrowing books

When editing an user, we want to be able to borrow a book for this user.

That means that when, we edit an user, we must display the borrowed books and being able to borrow a new one. Therefore, when we initialise the user component, we need the list of borrowed books for the user and the list of free books.

6.4.11.1 Display borrowed books

First, we need a borrow class. Fill it with the attributes and a constructor. Consider you could add the book_title as an attribute.

Next, create a borrow service to manage book borrowing : borrowservice. In your borrow service add a method to get user's borrowed books from the user ID. Add a method in book services to get borrowable books. These 2 methods may call BackEnd server with appropriate routes.

Now, the links in "user.ts". Import borrow service. Add a Borrow array to manage borrowed books and a Book array to get borrowable books. Do not forget to create them in the constructor. In ngOnInit, after you retrieved user, call a method to initialize user's borrowed books (with user ID as a parameter) and the list of borrowable books, like in [Fig. 200]. You do not need to wait the result of a service to call the next one, so you can call the services without chaining them.

```
if (personId > 0) {
  this._user.getUser(personId)
  .subscribe(
    (data) => {
      this.user = data[0];
      this.setDataForm();
      this.changeDetector.markForCheck();
    }
  );
  this._borrow.getBorrows(personId)
  .subscribe(
    (data) => {
      this.borrows = data;
      this.changeDetector.markForCheck();
    }
  );
} else {
  this.user = {person_id: -1, person_firstname: "", person_lastname: "", person_birthdate: new Date()};
  this.borrows = new Array();
  this.setDataForm();
}
this._book.getBorrowable()
.subscribe(
  (data) => {
    this.books = data;
    this.setDataForm();
    this.changeDetector.markForCheck();
  }
);
```

Fig. 200 : initialize borrowable books and borrowed books in user.ts

In your file server.js, implement the 2 routes you defined in bookservice. Add a method to get borrowed books from a user, and a method to get borrowable books (not currently borrowed books). [Fig. 201] shows a way you can implement them.

```
app.post("/borrows", function (req, res) {
  var person_id = req.body.person_id;
  var sqlRequest = "SELECT Borrow.*, Book_Title FROM Borrow JOIN Book USING (Book_ID) WHERE Person_ID=$1 ORDER BY Borrow_ID";
  var values = [person_id];
  getSQLResult(req, res, true, sqlRequest, values);
});

app.post('/borrowableBooks', function (req, res) {
  var sqlRequest = 'SELECT book.* FROM book WHERE book.book_id NOT IN (SELECT book_id FROM borrow WHERE borrow_return IS NULL)';
  var values = [];
  getSQLResult(req, res, true, sqlRequest, values);
});
```

Fig. 201 : BackEnd borrowing books

Next, HTML part. Copy the borrowed html code from your html work to the user component. Change your HTML component to manage the borrowed books with a *ngFor loop on the borrowed books.

When you display the returned date, it might be undefined (null) if book is not returned yet, or you may have a date if it is returned. To display the returned date or a button to return a book, we need a command “ngSwitch” to select which kind of element to display. [Fig. 202] shows how to use it.

```
<tbody>
    <tr *ngFor="let borrow of borrows">
        <td scope="col" class="text-center">{{ borrow.borrow_date | date: 'dd/MM/yyyy' }}</td>
        <td>{{ borrow.book_title }}</td>
        <td class="text-center" [ngSwitch]="borrow.borrow_return">
            <span *ngSwitchCase="null"><button class="btn" (click)="onReturn($event, borrow)"><img src="assets/img/return.png" alt="return"
            <span *ngSwitchDefault>{{ borrow.borrow_return | date: 'dd/MM/yyyy' }}</span>
        </td>
    </tr>
</tbody>
```

Fig. 202 : borrowing books

6.4.11.2 Returning a borrowed book

When you click on the return button, you call a method in the TS user component. This method gets the borrow ID, and ask the borrow service to “return” it, that means it might fill the borrow_return attribute. [Fig. 203] shows how to return a book.

```
onReturn(event:Event, borrow:Borrow) {
  var borrowId = borrow.borrow_id;
  var when = new Date();
  this._borrow.returnBorrow(borrowId, when)
  .subscribe(
    (data) => {
      borrow.borrow_return = when;
      this.changeDetector.markForCheck();
    });
}
```

Fig. 203 : borrowing books return method

6.4.11.3 Borrowing a new book

Next, to borrow a new book, add a loop on the borrowable books to display these books.

Add a (click) attribute in the add button in the HTML user file.

In the TS user component, create the corresponding method “onBorrow”.
Method onBorrow might get the selected ID value (maybe adding an ID to the select tag and getting its value through GetElementByID should be easier) and ask a borrow service to add a new borrow book to user. That method calls a method borrowBook in the borrow services.
The NodeJS server might implement a borrowbook route that insert a new borrowed book. It needs the book to borrow, its ID, the user, its ID too, and the date is today (use new Date()).

[Fig. 204] shows how to do this.

```
<tr>
    <td scope="col"></td>
    <td>
        <select name="bookId" id="bookId" class="form-control form-select form-select-lg mb-3">
            <option  *ngFor="let book of books" value="{{ book.book_id }}">{{ book.book_title }}</option>
        </select>
    </td>
    <td class="text-center">
        <button class="btn" (click)="onBorrow($event, user)"><img src="assets/img/plus.png" alt="add" class="icon" /></button>
    </td>
</tr>
```

```
onBorrow(event:Event, user:Person) {
  var personId = user.person_id;
  var bookid = Number((document.getElementById('bookId') as HTMLInputElement).value);
  this._borrow.borrowBook(personId, bookid)
  .subscribe(
    (data) => {
      this._borrow.getBorrows(personId).subscribe(data => {
        this.borrows = data;
        this.changeDetector.markForCheck();
      }
    );
    this._book.getBorrowable().subscribe(
      (data) => {
        this.books = data;
        this.changeDetector.markForCheck();
      }
    );
  });
}
```

Fig. 204 : borrowing new book

Then, try it.

Questions

What do you use components, services, and classes in ANgular ?
How do you manage lists of objects with Angular ?
How do you create a loop on an Array in the html part of a component ?
How do you manage routes ? How do you switch to another page ?
How do you force Angular to refresh data in the screen when you get new values ?

6.4.12 Summary

Here is a summary of the main actions we used for the project

- Create FrontEnd
  - Install angular :
    npm install -g @angular/cli
  - Create angular app
    ng new myApp
  - Create components, services and classes
    ng generate component aComponent
    ng generate service aService
    ng generate class aClass
  - Define routes in app.routes.ts
  - Add static elements in public/assets
  - Modify "index.html" in "public" to add js and css
  - Launch server
    npm start
- Initialise BackEnd server
  - Create NodeJS server directory (same level ass your Angular app)
  - Initialise server
    npm init
  - Install express and pg (or your database connector)
    npm install express --save
    npm install pg --save
  - Create NodeJS server file with server.js
  - Add route listeners (app.get and app.post)
  - launch server
    npm start

Things you might think to :

- 2 servers : 1 for Angular, 1 for nodeJS
- Angular means “single page”. So, if you have several pages you might use routes and navigate through routes
- Create as many components and services as you need
- Classes help to define data
- Use components to manage informations and services to connect to the BackEnd.
- Backend server is the only one that can reach the database
- Each time you change server.js, your nodeJS server file, you must restart nodeJS server.
- javascript asynchronous function uses Promises and Observables, that means an instruction might launch a callback function to get the response. Therefore, You do not get the response when the instruction is ended, so do not try to use the response just after the instruction, it might not be available yet. Use subscribe to wait and get the answer when it is ready.
- **use console.log to display informations in your browser to understand what happens**

7 Annexes

7.1 Tools for every practical works

First there are tools you will use for each practical work

- A text editor. For example Visual Studio Code, NotePad++, BBEdit, Atom, Bracket, or Sublime Text, ...
- A Web browser, with activated degugging tools : Firefox, Chrome, Safari, Opera.
  You should avoid Edge, because of poor debugging tools.
- A Database server : PostgreSQL, at least version 9. Prefer version 11 or 12 if possible.
- JQuery library (should be downloaded locally)

You will also need specific tools for each sequence

- For the first practical works (HTML, CSS, JS)
  - HTTP server - apache, XAMPP, wamp, ...
- For PHP-Symfony
  - PHP 8.1 connected to your web server but should also be used from the terminal
  - Symfony 6.3 or above
- For Spring
  - Java 8 to 20 - JDK version. Maybe >= 15
    You can keep multiple versions on your disk.
  - Netbeans 18
  - Tomcat 9. Not Tomcat 10.
- For React/Angular & NodeJS
  - NodeJS with express, express-generator, pg
  - npm (should be installed with nodeJS)
  - React/Angular will be downloaded by npm.

7.2 Data

For most of our web applications, we will use data. [Fig. 205] is the Physical Schema we will use.

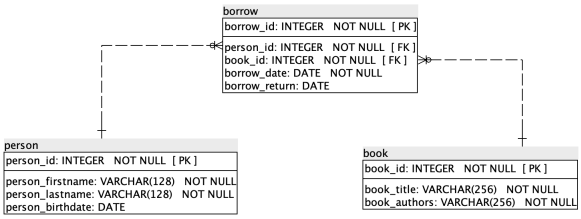


Fig. 205 : The database schema

The files “CreateBase.architect” and “CreateBase.sql” in the materials may help you to create your database.

7.3 Debugging

7.3.1 Browser tools

Usually, on every browser, there are tools that can help you to understand what happens.

7.3.1.1 Browser console

Console is usually available on every browser. It displays informations when the page is loaded and when a JS script uses console.log(something).

Console can be used :

- By programs like Angular, React or VueJS to display informations (because they run in the browser)
- Directly by browser's user to launch commands.

7.3.1.2 Browser debugger

Very often, on browser, there are some tools for debugging.
These tools displays html source code, JS code.

In the JS code (and the HTML one if you use scripts in HTML code), you can add breakpoints. If, during the page use, the script with a breakpoint is used, it will stop there and allow you to go step by step, check variables, ...

7.3.2 Debugging tools on Safari

To activate SAFARI debugger, Go to the “Preferences”. Select tab “advanced”. Click on “activate debugging tools” bottom of the window. Restart Safari.

To access debugger tool, in menu “Development”, call “Web Inspector”. You can also right click in the window and ask for Source code.

[Fig. 206] shows SAFARI debugger.

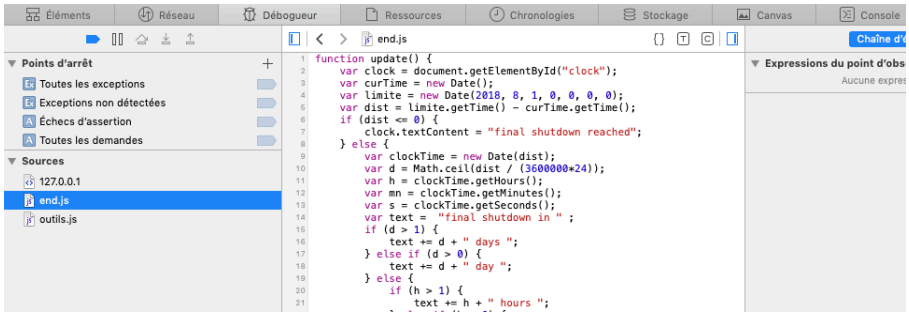


Fig. 206 : The SAFARI debugger

- Tab “Elements” shows the CURRENT DOM. That reflects what is currently displayed.
- Tab “Console” is the console. It may display errors. You can also launch JS commands.
- Tab “Sources” is the initial source code. You can access every used file for this page, add you breakpoints, manage debugging.

7.3.3 Debugging tools on Firefox

If you want to use FIREFOX debugging tools, use the icon at the top right with the 3 horizontal bars. Go down to “Web Development”, select “Inspector”.

[Fig. 207] shows FIREFOX debugger.

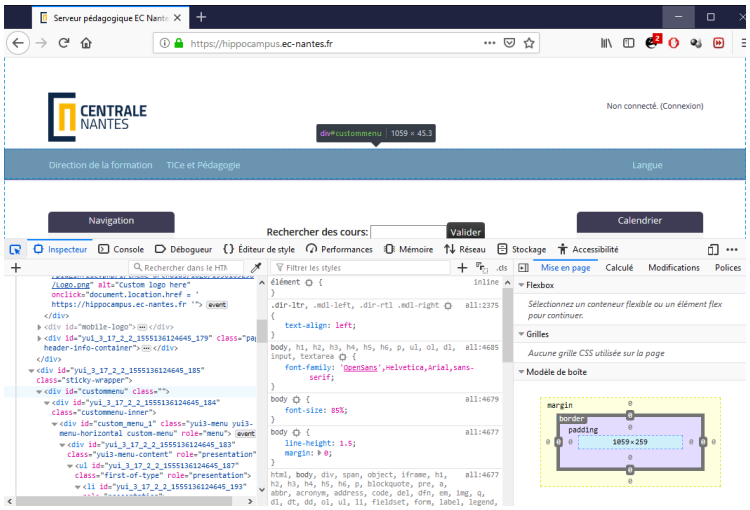


Fig. 207 : The FIREFOX debugger

- Tab “Inspector” shows the CURRENT DOM. That reflects what is currently displayed.
- Tab “Console” is the console. It may display errors. You can also launch JS commands.
- Tab “Debugger” allows breakpoints, check values, step by step execution, ...

7.3.4 Debugging tools on Chrome

If you want to use CHROME debugging tools, use the icon at the top right with the 3 vertical dots. Go down “More tools”, select “Development tools”.

[Fig. 208] shows CHROME debugger.

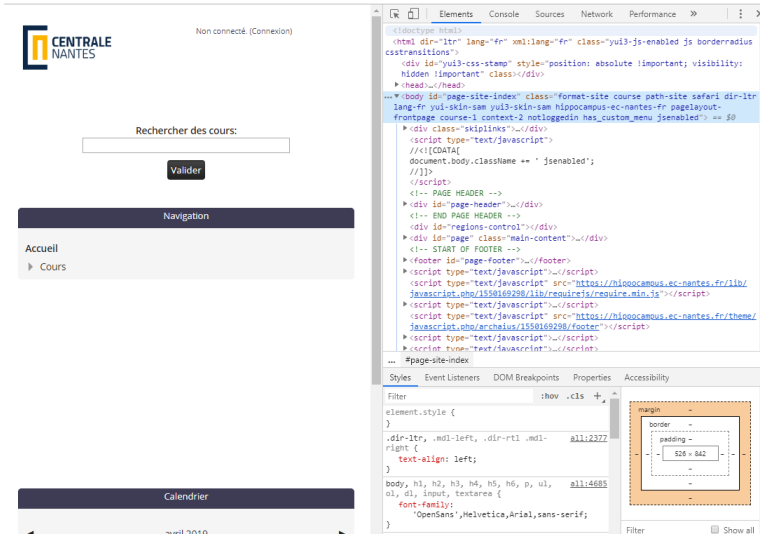


Fig. 208 : The CHROME debugger

7.3.5 Web browser debugger check

Open your browser with this URL : https ://www.ec-nantes.fr.
Select debugging tools. Navigate using tabs.
Check you can add breakpoints in the JS source code.

7.4 PostgreSQL

PostgreSQL is a relational database server.

7.4.1 PostgreSQL version

We will use versions, 11 or more. If possible version 14 or 15. You can choose the one you want.

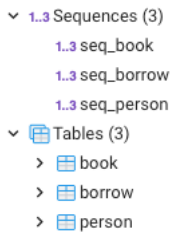
You may use another kind of server if you want, but our slides are written for it. If you use something like mysql or mongoDB, you will have to use the right modules.

Downloading, Installing and Configuring PostgreSQL is described in the slides “Install Party”.

7.4.2 PostgreSQL importing data

- Create a user “prweb”. This user might be able to connect.
- Create a new database. Call it “prweb”. Set user “prweb” as owner.
- Open Query tool. Import “CreateDatabse.sql”. Launch it.

You should have a database with data.



7.5 HTTP server Installation

This section is **only for Windows users**. MacOS and Linux users already have one HTTP server installed.
Also, if you already installed XAMPP, WAMP or MAMP, you do not have to install another one.

Use your web browser and download XAMPP at the following URL :
https ://www.apachefriends.org/fr/index.html

If you prefer WAMP or XAMPP, it is up to you to use them.

Install it.
From XAMPP server, we will only need HTTP server and PHP.
You are not required to install something else but it is up to you.

7.5.1 HTTP server config location

Server configuration is in the file httpd.conf.

- Windows
  - MAMP, XAMPP : in your application directory, in “htdocs”
  - WAMP : in your application directory, in Apache2\conf directory
- MacOS - linux
  in /etc/apache2

7.5.2 HTTP server root location

server root location tells where your html files should be located on your disk. That means, when you use protocol http in your browser, root access is that directory.

- Windows
  - MAMP, XAMPP : in your application directory, in “htdocs”
  - WAMP : in your application directory, in Apache2\htdocs directory
- MacOS
  /Library/WebServer/Documents
- Linux
  Depend on your distribution, but should be in /var/www

7.5.3 HTTP server check

- Run your server
  - Windows : Launch the admin tool (XAMPP) of your web dev environment (WAMP, MAP, ...) and click to start server.
    Sometimes there is a problem with directory rights. Fix it.
  - MacOS, Linux : should be launched
    if not, use “sudo apachectl start” from your console.
- Check it works fine by using following URL :
  http ://localhost

7.6 PHP Installation

For the practical work, we recommand PHP 8. It may work with PHP 7 but instruction might be sometimes a bit different.

- Windows
  PHP 8 should have been installed with XAMPP/ WAMP / MAMP
- MacOS
  - Before MacOS "Big Sur" (MacOS 11) : Should already been installed by Apple. Check it is version >= 8.1
  - Since MacOS "Big Sur" (MacOS 11) : You may encounter several problems when using PHP and Apache due to unsigned modules.
- Linux
  If not installed may be you should use something like "apt-get install php8". Depends on your distribution.

7.6.1 PHP and MacOS "Big Sur" (MacOS 11) and above

With MacOS "Big Sur", MacOS announced they will not support PHP any more.

Since Mac OS "Monterey", PHP is not installed by default. If you upgraded MacOS from older versions, and you already installed PHP, you may have kept it installed, but depending on the way you installed your version, you can have some problems when using it with Apache http.

If you use a HomeBrew version, you will have a "code signing" problem. According to Apple, Homebrew PHP version is not signed (understand not checked by Apple). So, you will have to tell your OS your PHP version is ok.

If PHP is not yet installed on your computer, we suggest you use Homebrew, then you can install PHP with brew.

7.6.1.1 Installing Homebrew

HomeBrew is a software installer for many tools. It manages pre-configured tools install them, allows version management, upgrades, ... Very often, such an installer is useful because it has been checked by experimented users.

HomeBrew is used through your terminal with the "brew" command.

You can find Homebrew installation process on this site : https ://brew.sh/index_fr

Some commands you may know :

- brew update
  upgrades Homebrew
- brew upgrade
  update installed packages
- brew upgrade
- Once brew is installed, add :
  - **wget** : brew install wget
  - **curl** : brew install curl

7.6.1.2 Installing php with Homebrew

Open your terminal / command tool.
The command to install php is :

```
brew install php
```

It will install the latest available php version.
Also, you can use “brew install php@8.1” if you want to install the 8.1 version. And the same kind of command for other versions.

7.6.1.3 Check if PHP module has to be signed

If you use php with your terminal, everything will we ok. It is different when using it with the Apache HTTP server on your Mac. PHP module is considered as unsigned, so we have to sign it.

To check it, with your terminal, use following commmand :

```
sudo apachectl -t
```

Required password is your macos password (not the root one if you created a root account).

If the answer tells there is an error with your php module, then you probably have to sign it.

7.6.1.4 Signing PHP module

You should follow these instructions if, and only if, you use MacOS and you have a problem with the PHP module in Apache HTTTP server.

First, we have to create a local certificate, and a Certificate Autority to validate it. Then we sign the PHP module with this certificate. At last, we tell Apache HTTP that the PHP module is signed by our certificate.

7.6.1.4.1 Create a local Certificate Authority

If you already created a local certificate authority, skip this phase.

Thanks to Mohd Shakir Zakaria for the process.
https ://www.simplified.guide/macos/keychain-ca-code-signing-create

Launch your “Keychain Access” Application. In the “Keychain Access” menu, use “Certificate Assistant”, then “Create a Certificate Authority”. Have a look to [Fig. 209].

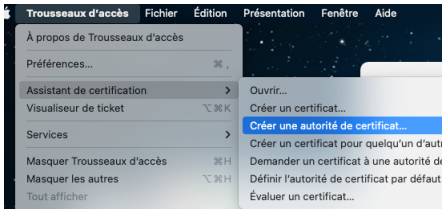


Fig. 209 : Call Create Certificate Autority

A window should appear like [Fig. 210].

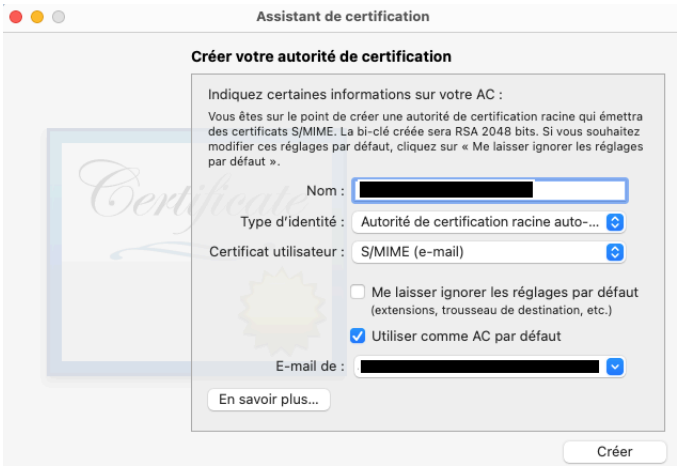


Fig. 210 : Create Certificate Autority

Give a name to your certificate authority (For example “CA for ... your name””).
For”User Certificate”, select “Code Signing”.
Check the “Let me ignore defaults”.
Check/Fill your email adress in the appropriate field.

You should have something like [Fig. 211].

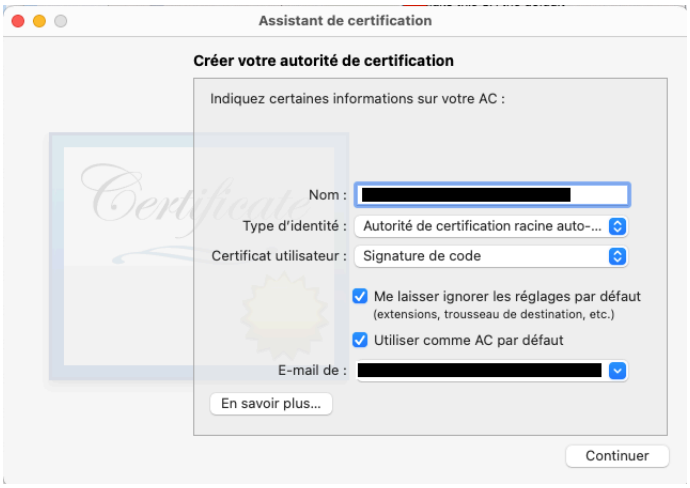


Fig. 211 : Create Certificate Autority

Click “Continue”. Validate warning message (if it appears) to continue.

Click “Continue” to validate certificate informations [Fig. 212].
Validate warning message (once more if it appears) to continue.

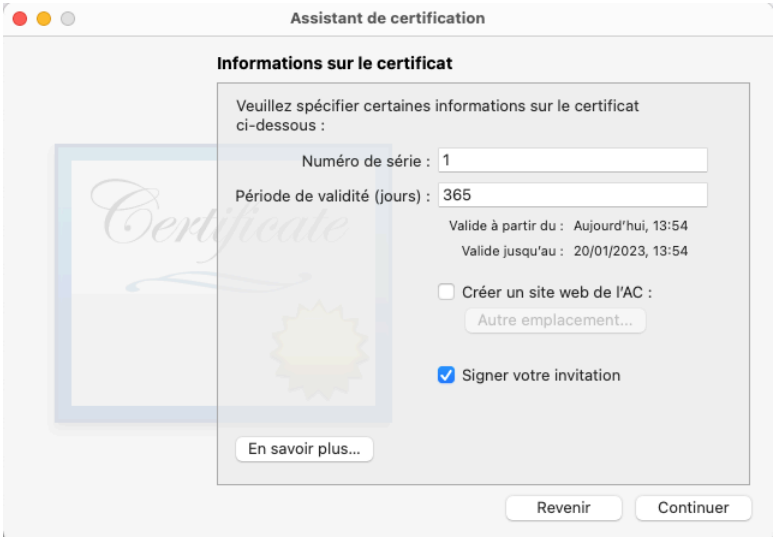


Fig. 212 : Create Certificate Autority

In the next screen check/add your personal informations and your CA informations [Fig. 213].
Click “Continue” to validate informations.

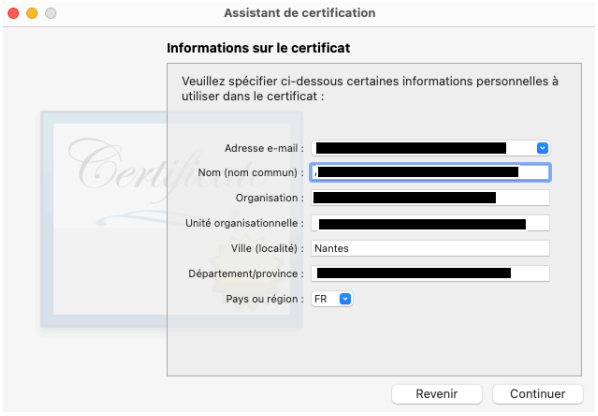


Fig. 213 : Create Certificate Autority

You should be proposed an encryption algorithm and a key size [Fig. 214].
Click “Continue” to generate a Key Pair Information. That will define your encryption algorithm and private/public key for your certificate.

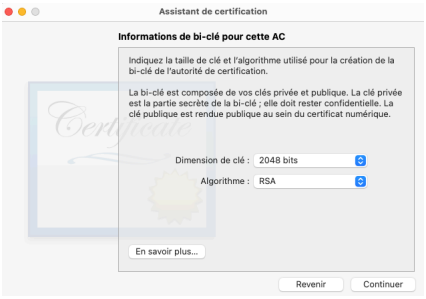


Fig. 214 : Create Certificate Autority

Next is the same for a specific key pair information. [Fig. 215]. Click “Continue”.

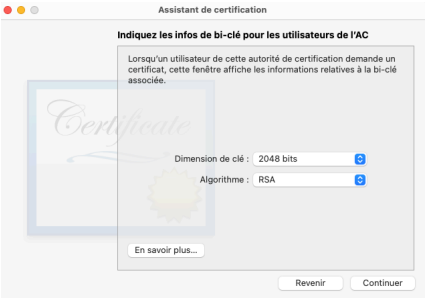


Fig. 215 : Create Certificate Autority

Check key usage extension for your CA. [Fig. 216].
“Include Key Usage Extension” should be checked.
Also, you should have “signature” and “Certificate signing” checked.
Click “Continue” to validate.

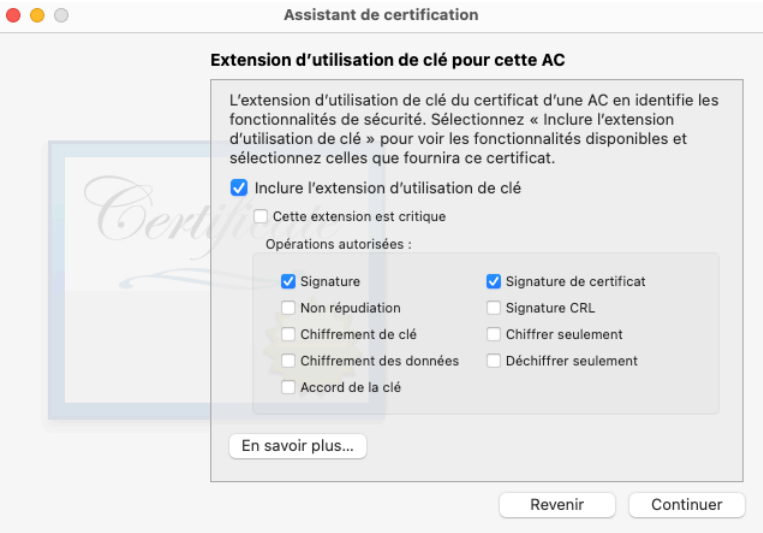


Fig. 216 : Create Certificate Autority

Then same for Users of the CA. [Fig. 217].
“Include Key Usage Extension” should be checked.
“This extension is critical” too. Also, you should have “signature” and “Certificate signing” checked.
Click “Continue” to validate.

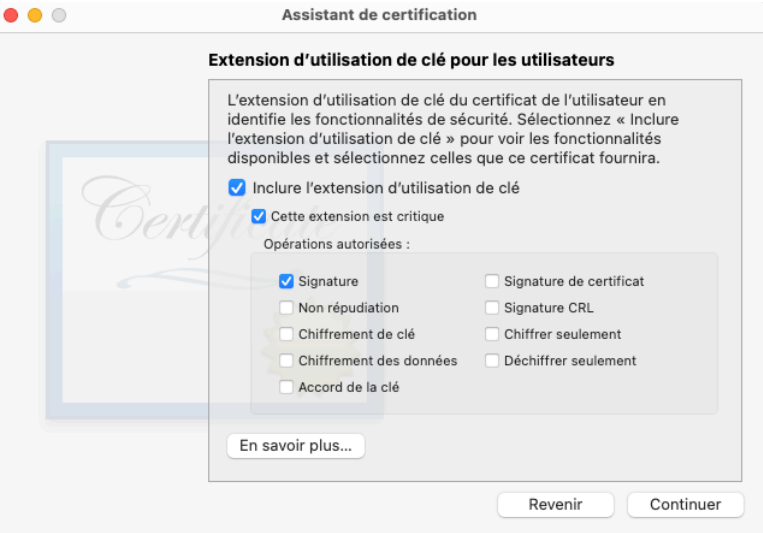


Fig. 217 : Create Certificate Autority

Then a screen about Extension Key Usage. [Fig. 218].
Check “Include Extension Key Usage Extension”.
Options appears. Check “Code signing”.
Click “Continue” to validate.

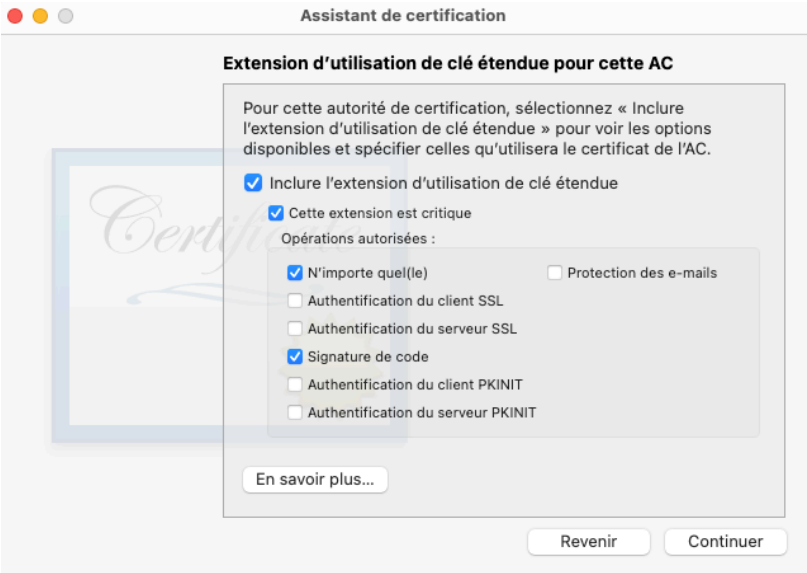


Fig. 218 : Create Certificate Autority

Then, nearly same screen. [Fig. 219].
Click “Continue” to validate.

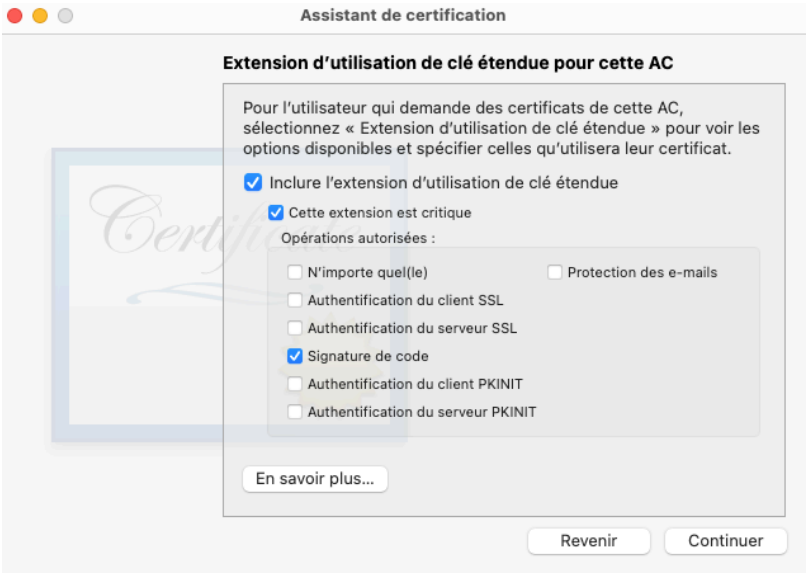


Fig. 219 : Create Certificate Autority

Next, about basic constraints. [Fig. 220].
Click “Continue” to validate.

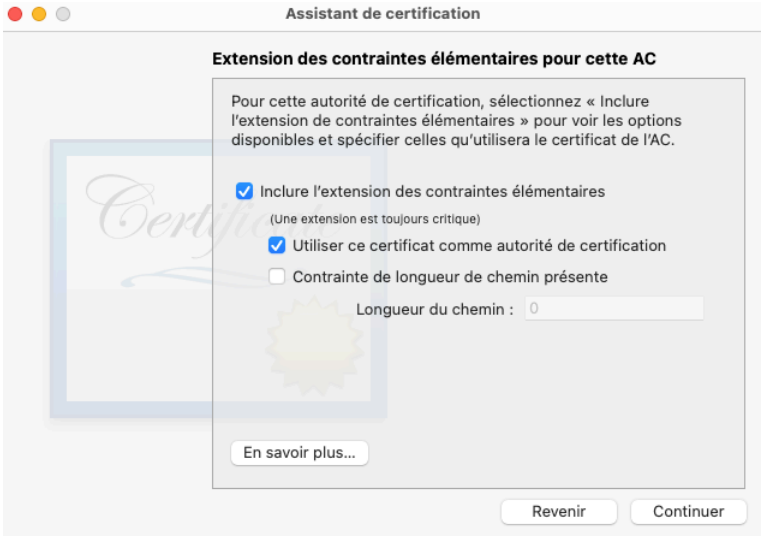


Fig. 220 : Create Certificate Autority

Validate next screens by clicking on “Continue”.

Finally, you can create the Certificate Authority [Fig. 221].
Click “Create” to create it.

Fig. 221 : Create Certificate Autority

You can close certificate autority assistant.

Now, we have to tell we use it. Back to the Keychain Acces window. Use tab “My Certificates” [Fig. 222].

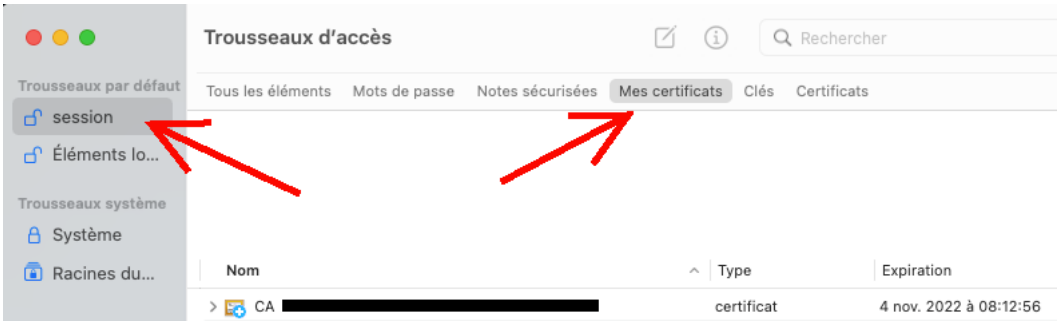


Fig. 222 : Using Certificate Autority

Select your Certificate Autority and Double Click on it to get informations [Fig. 223].

Fig. 223 : Using Certificate Autority

Open “Trust” by clicking on “>”, and switch “When using this certificate” to “Always Trust” [Fig. 224].

Fig. 224 : Using Certificate Autority

Close window.
We created our Certificate Authority.

7.6.1.4.2 Create a local Certificate

To create our certificate we need a local Certificate Authority.
So, ensure you already created it.

Launch your “Keychain Access” Application. In the “Keychain Access” menu, use “Certificate Assistant”, then “Create a Certificate”. Have a look to [Fig. 225].

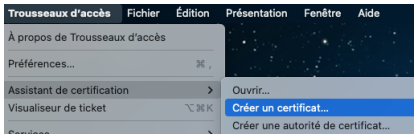


Fig. 225 : Call Create Certificate

A window should appear like [Fig. 226].

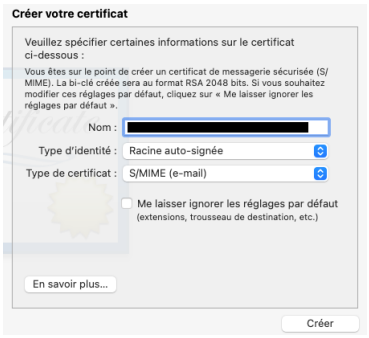


Fig. 226 : Create Certificate

Give a name to your certificate (For example “Certificate for … your name””).
For”Identity Type”, switch to “Leaf”.
For “Certificate Type”, switch to “Code Signing”.
If your window looks like [Fig. 227], create your certificate data.

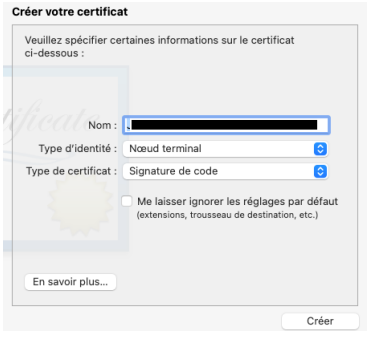


Fig. 227 : Create Certificate

Now we have to choose a Certificate Authority to validate our certificate.
We choose our local Certificate Authority, like in [Fig. 228].

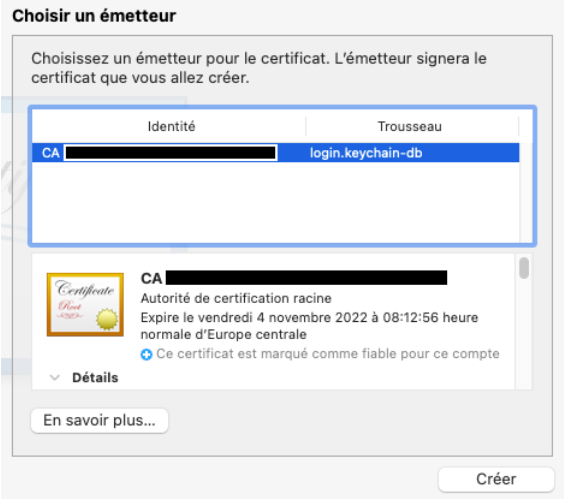


Fig. 228 : Create Certificate

Ensure your local Certificate Authority is selected and click on “Create” to create it.

If everything is OK, your certificate is created.
You should have a confirmation window like in [Fig. 229].

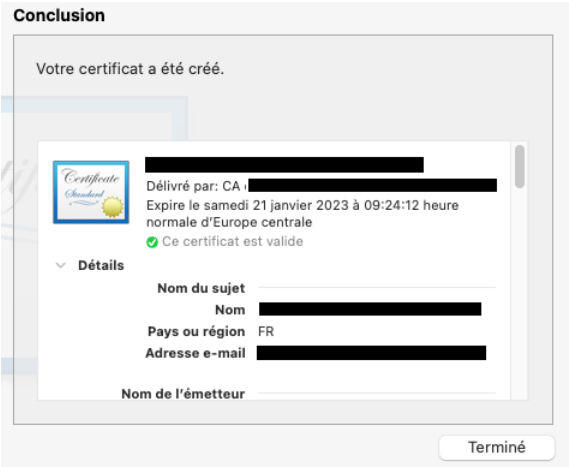


Fig. 229 : Create Certificate

Close window by clicking on “Done”.

Ensure your certificate is created in Keychain Access.

7.6.1.4.3 Sign PHP module

To sign our PHP module we need a local Certificate. So, ensure you already created it.

Launch XCode once and install terminal tools, or use your terminal and manually install them with this command :

```
xcode-select --install
```

Locate your PHP module in the Apache config file. You can use this command that tells you in which file is the module configuration, and the corresponding line

```
grep -nir "^loadmodule.*php" /etc/apache2
```

The result should be something like in [Fig. 230].

Fig. 230 : Search for PHP module configuration line

First part of the result gives the file that contains the configuration line, then you have the line number in this file. Last part gives the current command. Do you notice the file location after "LoadModule php_module"? It gives the php module location in your hard disk.

Now that we know where is located the file, we sign it with our certificate.
This command (on a single line) sign your file.

```
codesign --sign "Your certificate name" --force --keychain ~/Library/Keychains/login.keychain-db Your php module location
```

Of course, you must replace the red elements by their value on your computer.

Now, we have to tell apache which certificate signs the php module.

Open the apache configuration file that mounts the php module (remember the grep command we launched ?). Go to the line that mounts the php module (the line we've got with grep).
Add your certificate name at the end of the line.

```
LoadModule php_module Your php module location "Your certificate name"
```

Relaunch apache.

```
sudo apachectl -k restart
```

7.6.2 PHP check installation for HTTP server

PHP should be linked to your HTTP server.

Open the file “httpd.conf” in the http config directory.
Find line that contains “LoadModule php7_module”. if the line starts with “#”, remove the # character to activate php. Restart your http server.

7.6.3 PHP check installation for your terminal

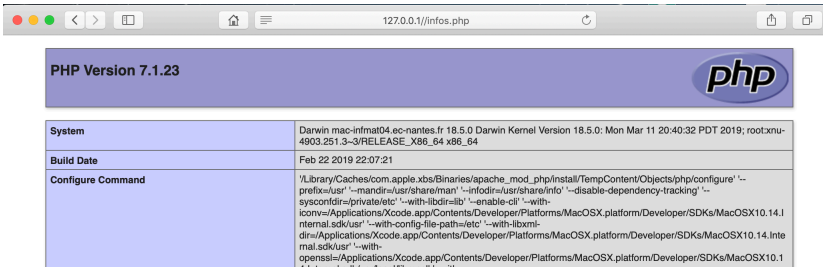
Open your terminal / command tool.
Use command :
php -v

The result should be a version >= 7.2, if possible 8.

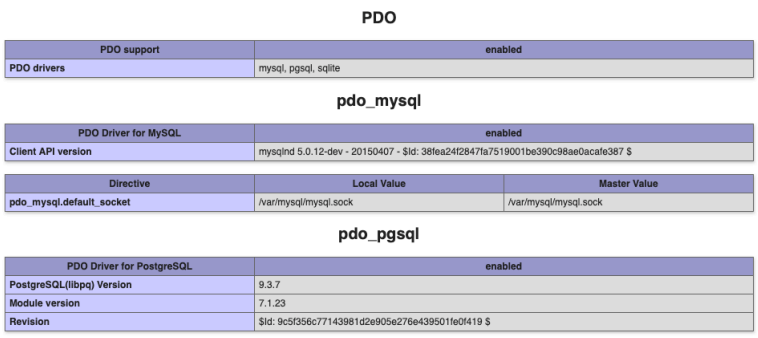
7.6.4 PHP configuration

Copy the file infos.php to your http server root location. Use your browser and call this URL : http ://localhost/infos.php

You should have something like this :



In this page, find “PDO” You should have something like this :



if “PDO PGSQL” is not active, you have to activate it.

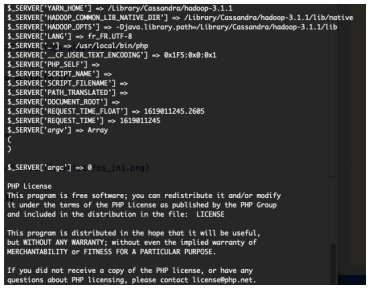
Only if PDO PGSQL is not active :

Still in this page, try to find “php.ini”

Configuration File (php.ini) Path	/etc
Loaded Configuration File	/etc/php.ini

Now, open the ini file (probably php.ini).
Find “php_pdo_pgsql” in this file
Remove “;” at the beginning of the line.
Restart http server.

Open your terminal / command tool.
Use command : php -i
You should get something like that.



In the results line, locate the PDO section

If **pgsql** is not in the list, you have to add it.
Locate file php.ini in the results line (it is possible it is not the same the HTTP server one).

7.7 Symfony Installation

Have a look to https ://symfony.com/doc/current/setup.html

You have to install 2 tools

- composer
- symfony-cli

7.7.1 Install Composer

Composer is a tools that helps to manage file creation

Have a look to https ://getcomposer.org/download/
You might have something like in [Fig. 231]

Fig. 231 : Get Composer

Copy the instruction one by one to your terminal / Invite de commande.
That should install Composer and remove installation files.

7.7.2 Install Symfony-cli

Symfony-cli manages symfony project creation, compilation, building, running.

Have a look to https ://symfony.com/download You might have something like in [Fig. 232]

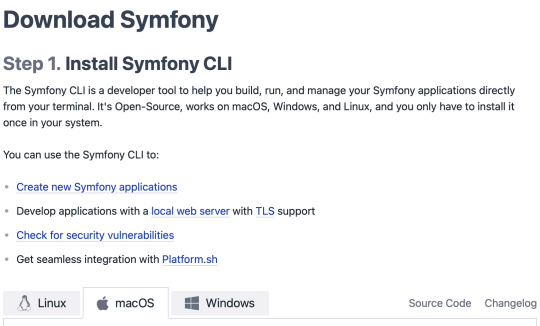


Fig. 232 : Get Symfony-cli

First, select you r system. It should be selected by default, but maybe nott.

7.7.2.1 Windows Installation

For windows, you have to install scoop first. That tool will install specific tools and bypass some windows security permissions.
SO DO NOT USE THIS TOOL IF NOT required.

You will need to start the **PowerShell** Tool. Not the "Command tool".
Have a look to https ://scoop.sh and follow instructions.

Once done, you can use your "Command tool" or go on with "Powershell".
Downlod Symfony-cli with scoope.

```
scoop install symfony-cli
```

7.7.2.2 MacOS Installation

Use the Terminal.

You can use the instruction you prefer, if the tool is installed of course.
If Homebrew is installed, it is easier to use and manage versions.

7.7.2.3 Linux Installation

Use the Terminal.

wget and curl should be installed, so you can use one of the 2 commands.
If you use a Debian/Ubuntu distribution, maybe using apt is a good idea. Same for Fedora/RedHat/Suse distribution for dmf.

7.8 Java Installation

You should already have install Java, JDK version 8 or more.

7.8.1 Check version

Use terminal / Command tool and use these command lines :

- java -version
  - If you have an error message, java should not have been installed. So install it, JDK version.
  - If a message tells you that variable JAVA_HOME is not define, fix it.
  - If your version is <8, update it.
- javac -version
  - If first command was ok, and this one not, maybe you installed JRE instead of JDK. Install JDK.
  - If this command is ok but the version is not the same than the first line, your last installation was a JRE. Install JDK.
- echo $JAVA_HOME (or echo %JAVA_HOME% for windows)
  - If this command does nothing (nothing appears as a PATH), your JAVA_HOME variable is not defined. Add it with the path to the JRE in your environment variables.

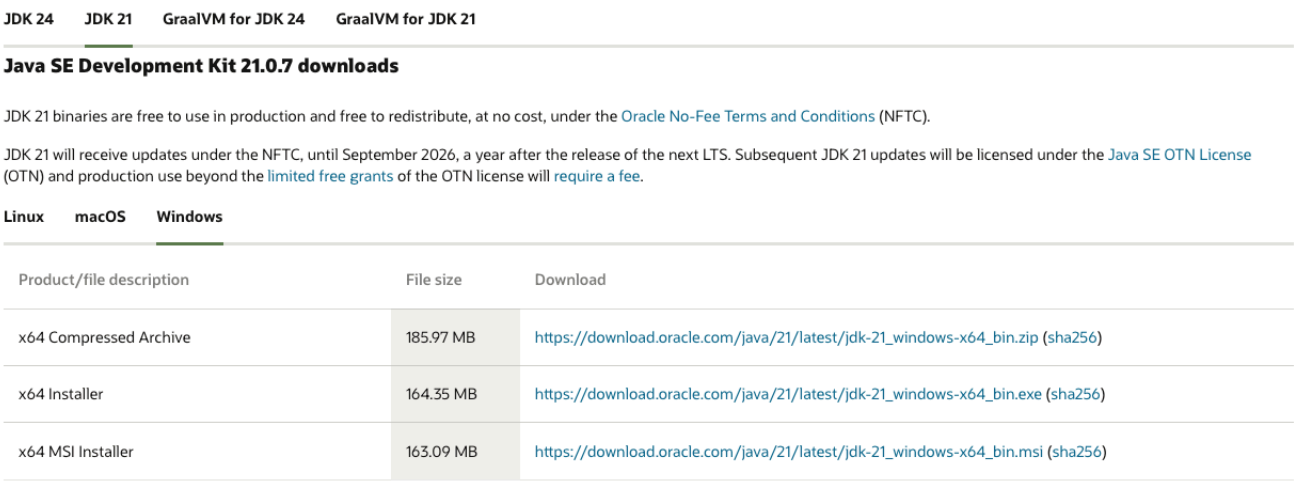
7.8.2 installation

With your browser, connect to :
https ://www.oracle.com/fr/java/technologies/javase-downloads.html

With Windows or Linux, do not use any other site than the Oracle one to download JAVA. For linux you can also install OpenJDK.

Go to the version you want to install. On the right side of the window, you should find “Oracle JDK” This is the one we want to install.

Figure [Fig. 233] shows the Oracle Installation page.



JDK 24 JDK 21 GraalVM for JDK 24 GraalVM for JDK 21

Java SE Development Kit 21.0.7 downloads

JDK 21 binaries are free to use in production and free to redistribute, at no cost, under the Oracle No-Fee Terms and Conditions (NFTC).

JDK 21 will receive updates under the NFTC, until September 2026, a year after the release of the next LTS. Subsequent JDK 21 updates will be licensed under the Java SE OTN License (OTN) and production use beyond the limited free grants of the OTN license will require a fee.

Linux macOS Windows

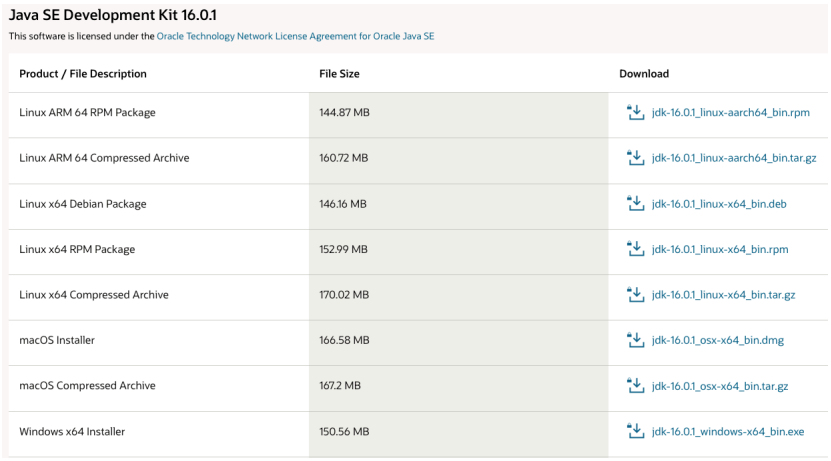
Product/file description	File size	Download
x64 Compressed Archive	185.97 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.zip (sha256)
x64 Installer	164.35 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.exe (sha256)
x64 MSI Installer	163.09 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.msi (sha256)

Fig. 233 : JAVA Installation JAVA - selection page

Cick on “Download” in the JDK section. You should have a new page.

Take a bit of time to read the licence.

Go down to “Java SE Development Kit”. Find the line for your OS, and click on the download link. Figure [Fig. 234] shows the link to the installation tool.



Java SE Development Kit 16.0.1

This software is licensed under the Oracle Technology Network License Agreement for Oracle Java SE

Product / File Description	File Size	Download
Linux ARM 64 RPM Package	144.87 MB	jdk-16.0.1_linux-aarch64_bin.rpm
Linux ARM 64 Compressed Archive	160.72 MB	jdk-16.0.1_linux-aarch64_bin.tar.gz
Linux x64 Debian Package	146.16 MB	jdk-16.0.1_linux-x64_bin.deb
Linux x64 RPM Package	152.99 MB	jdk-16.0.1_linux-x64_bin.rpm
Linux x64 Compressed Archive	170.02 MB	jdk-16.0.1_linux-x64_bin.tar.gz
macOS Installer	166.58 MB	jdk-16.0.1_osx-x64_bin.dmg
macOS Compressed Archive	167.2 MB	jdk-16.0.1_osx-x64_bin.tar.gz
Windows x64 Installer	150.56 MB	jdk-16.0.1_windows-x64_bin.exe

Fig. 234 : JAVA Installation - select OS

A new window appers. **Check the checkbox** and download the file. Button is disabled as long as the checkbox is not checked.

Take care : versions <require to create an account on the Oracle site to download files. If you have to, use your ECN email address for that.

7.9 Tomcat Installation

Tomcat is a servlet server. That means it can handle java for web applications.
Java (JDK version) **must** be installed.

Download software through your web browser with this URL :
https ://tomcat.apache.org/download-90.cgi
Do not use Tomcat 10, because development librairies have changed significatively.

Download the lastest version 9 (find it in the Quick navigation section).
Select ZIP version from core. Unzip it (should be done on MacOS).
Move tomcat downloaded directory in your app directory, or where you want.
Windows users : Do not download it as a service.

7.9.1 Tomcat configuration

Go to your tomcat directory.
Open the conf directory.
Edit the tomcar-users.xml file. Add following lines just before the last line of the file :

```
<role rolename="tomcat"/>
<user password="admin" roles="tomcat,manager-gui,admin,manager-script" username="admin" />
DO NOT COPY THESE LINES, THEY MAY CONTENT UNCORRECT CHARACTERS.
```

Fell free to change login and password values.
You should have something like in [Fig. 235].

```
    <role rolename="tomcat"/>
    <user password="admin" roles="admin,manager,tomcat,manager-gui,manager-script" username="admin"/>
</tomcat-users>
```

Fig. 235 : tomcat_users.xml admin user definition

7.9.2 Tomcat check

Use your terminal / Command tool.
Go to your tomcat directory. Open the bin directory.
launch the "startup" script (.bat for windows, .sh for MacOS and Linux).

If there is an error, fix it.
Maybe a problem of rights in the Tomcat directory or for the log directory in Tomcat directory.
Also sometimes on unix and MacOs, shell scripts are not executable. Use "chmod a+x *.sh" for that.

Now, use your browser and call http ://localhost :8080
You should have something like [Fig. 236].

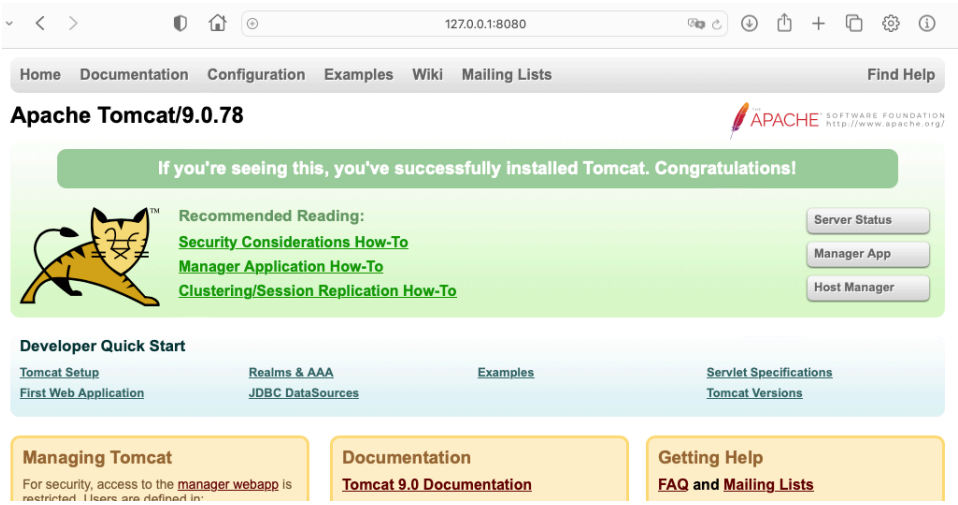


Fig. 236 : Tomcat startup page

This means tomcat can launch.

From the tomcat page, click on the “Manager App” button.
You should be asked for a login/password. Values are the ones you gave in the tomcat-users.xml file.
Then you should have the page in [Fig. 237].

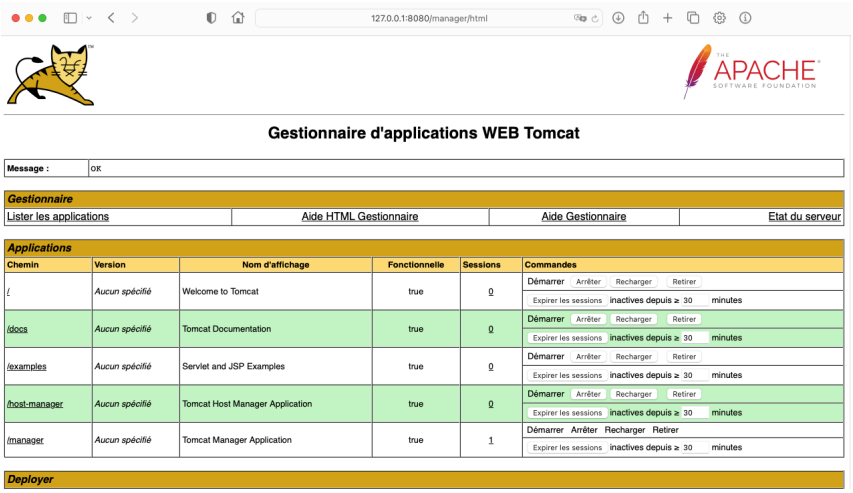


Fig. 237 : Tomcat manager page

You can shutdown it with the “shutdown” script in the terminal / Command tool.

7.9.3 Tomcat tools

In the Tomcat admin page, you can find :

- The list of deployed applications. You can stop them, launch/relaunch them, or remove them.
- A Deploy zone, deploy alternative directorys with web apps
- A Deply WAR zone. WAR files are Web ARchive files, that means compressed web applications. You can deploy war files from this zone, or by copying the waar file in the webapps directory of the tomcat directory.
- Diagnostics zone.
- Information zone.

7.10 Netbeans

We use Netbeans version 22/23, developed by Apache. We are not aware of all apache developments, so maybe there is a more recent version available.

Prerequisite : **Java must be installed on your computer, the JDK version**. Version 11 or more.

You can find the Apache Netbeans version here :
https ://netbeans.apache.org/download

Figure [Fig. 238] shows the download link.

Fig. 238 : Netbeans Download

Download version.

Start Netbeans to install required components. [Fig. 239] shows Netbeans first start.

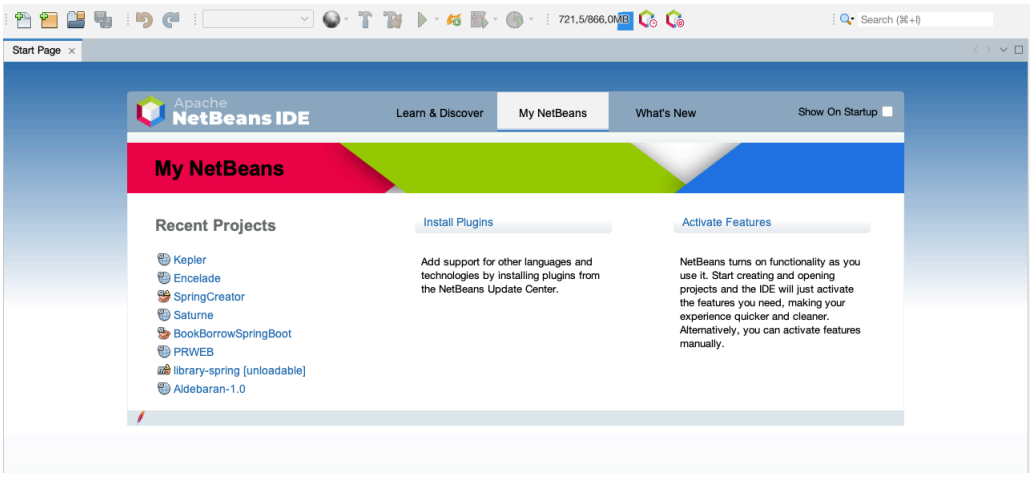


Fig. 239 : Netbeans first start

7.11 Symfony

Symfony is a PHP framework.

With your web browser, use this URL : https ://symfony.com/download

Select your system.
Read and follow the instruction to download “symfony-cli”.

symfony-cli is a tool to manage symfony projects.

Close your Terminal / Command tool if you opened it. Reopen it.
That will reload the PATH, which may now include path to the application “symfony”.
Check it works. If not, maybe your symfony path is not set. Fix it.

```
symfony -v
```

We will also need “composer”, a PHP symfony module.
With your web browser, have a look to https ://getcomposer.org/download/

To install composer, you have to run commands in your Terminal / Command tool.
Use the commands listed in the site, one by one.
If there is a bug when installing it may come from the hash checking. Do not take care of this instruction and go on with the installation.

Even for Windows (there is an installer), it should be a good idea to run these commands because it creates some files on your disk. Close your Terminal / Command tool if you opened it. Reopen it.

Run composer to check it works. [Fig. 240] shows the result. Maybe the version is different for you.

```
composer -v
```

Fig. 240 : Run composer

7.12 JQuery download

JQuery is a Javascript library.

Download it from http ://jquery.com/download/.
Get compressed compressed production file, the last version.

Keep this file. You will have to copy it in your javascript directory to use it.

7.13 Node JS

Node JS is a javascript application server.

7.13.1 Install nodeJS and npm

With your web browser, use this URL : https ://nodejs.org/.

You should have the instruction in the downloaded page like in [Fig. 241], with the right system and version. Choose the current one.

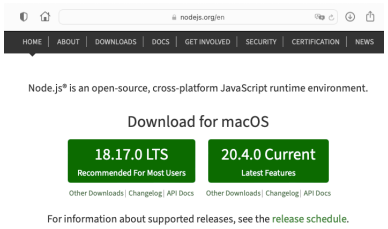


Fig. 241 : Install nodeJS and npm

Download the installer for your system.
Launch it and follow instructions.
You should only have to validate screens.

For linux, you can also use an “apt-get” or any package installer.
Just remember installing nodeJS and npm.

For MacOS, you can also use Homebrew, but this installer seems to be better.

At the end of the installation you should be told what nodeJS and npm versions are installed.

Open a terminal / command tool, check nodeJS and npm versions.
That will check they are correctly installed, and give you the version.
Be sure nodeJS is at least in version 18.
[Fig. 242] shows the result.

```
node -v

npm -v
```

Fig. 242 : Check nodeJS and npm

7.13.2 Install node modules

npm is the NodeJS package manager. We use it through terminal / command tools.

npm install is the instruction to install a module.
The installation can be :

- local
- global (for each modeJS server installed on the compuuter)

We use parameter “location” to tell with one we use.
If you choose a global installation (recommanded), **it requires to be root, or Admin for Windows**.
Linux / macOS user should use sudo npm install ...

There are many modules. We need the PostgreSQL one. Its name is “pg”. [Fig. 242] shows the installation of the postgres module for nodeJS.

```
npm install --location=global pg
```

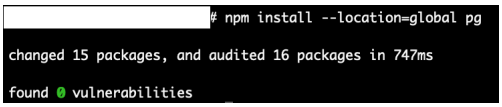


Fig. 243 : Check nodeJS and npm

7.14 PHP - Symfony Framework

DO NOT USE THIS ON MACOS.
Current PHP version on recent MacOS systems do not include Posgresql driver.
Brew version are refused because they are not signed.

Before we write something in Symfony, let's write a bit of PHP.

7.14.1 Introduction to PHP

The idea of this practical work is to understand PHP language.

7.14.1.1 Required tools

For this part, you will need :

- A web server to serve your pages
- **php 8** interpretor connected to your web server.
- a text editor to create pages
- JQuery file

7.14.1.2 PHP overview

PHP is a programming language. An interpreted language.
When you launch a file, it is compiled on the fly.

There are 2 ways of using it :

- With an HTTP server. Scripts will be called by HTTP server to produce HTML files.
- Standalone. Scripts are launch with the terminal.

7.14.1.2.1 PHP basics

A php script is included in a tag :

```
<?php
... Here is the PHP script
?>
```

Syntaxically, PHP is close to PERL, C, ...

- instructions end with ;
- use { ... } to define blocks.
- variables start with $ followed by the variable name.
- variables types depends on the context, and their previous use. Since PHP 7 it is possible to force variable types definition.
- Your are not obliged to declare a variable. Using it is considered as a declaration.
- to set a variable value, use =
- comments on 1 line can use //
- comments on more that 1 line are set with /* ... */

[Fig. 244] is an example of a PHP script.

```
<?php
$myVariable = 5; // Set myVariable to 5
$myVariable = $myVariable + 1; // Add 1
$myVariable2 = "aString";
?>
```

Fig. 244 : PHP script

7.14.1.2.2 Using types

You can manipulate standard types of data

- integers (1, 123, -42, ...)
- reals (12.34, -10.11, ...)
- boolean (true, false)
- characters, strings
  - '...'
  - "..."
- arrays
- objects

Strings can be set with '...' or "...", but they do not have the same effects.

If the "..." string contains a variable ($...), the variable is replaced by its value.
'...' keeps variables in the string. $... will not be replaced.

You can concatenate strings using operator .

```php
<?php
$i = 1;
$s1 = "i = $i";  // $s1 contains "i = 1"
$s2 = 'i = $i';  // $s2 contains "i = $i"
$s3 = $s1 . $s2; // $s3 contains "i = 1i = $i"
?>
```

7.14.1.2.3 PHP operators

PHP implements

- standard arithmetic operations
  - +, -, *, /
  - %, **
- usual functions
  - sin, cos, ..., asin, acos, ..., log, exp, ...
  - ceil, floor, round, ...
  - min, max, ..
  - rand, srand, ...
- standard boolean operations
  - &&, ||, !
- comparison instructions
  - ==, !=
  - ===, !== : compare value and type

And more.

7.14.1.2.4 Arrays

Arrays are created by their use or by an explicit declaration. Multi dimensionnal arrays are build as arrays of arrays. That means lines are not obliger to have the same dimension. At last, arrays are indexed by integers, reals, strings...

[Fig. 245] give examples of using PHP arrays.

```php
<?php
$myArray = array();
$myArray[0] = 1;
$myArray[1][2] = 3 + $myArray[0];
$myArray[2]["new"] = "aString";
?>
```

Fig. 245 : Using PHP Arrays

But if you try to take an undefined index from an array, you will have a warning error.

7.14.1.2.5 Control instructions

Control instructions cover :

- tests : if (...) { ... } else { ... }
- switch : switch ($expression) { case ... : .... break ; default : ... }
- for loops : for ($var = ... ; $var <... ; $var ++) { ... }
- while loops : while (...) { ... }

it also includes “foreach” to browse an array

7.14.1.2.6 Output

To print to the output flow you can use :

- echo ...
- print ...

There are only minor differences between these instructions, so you can use both.

- print_r(...) displays an array

```php
<?php
$i = 1;
print "Result is $i";
?>
```

7.14.1.2.7 Functions

functions look a bit more like JS ones.

```
function myFunction(...parameters...) {
  ...
  return result; // eventually
}
```

- Using & before a parameter make it passed by address.
- You can set default values to parameters

And the way to call them looks like every function call

```
$variable = myFunction(... args ...)
```

7.14.1.2.8 Objects and Classes

Since PHP 5, PHP can manage objects. Syntax is a bit more clear since version 7.

```
class myClass {
  ... Attributes
  ... Methods
}
```

Use **new** to create a new instance of a class.
Use **->** to call a method or access to an attribute.

```
class myClass {
  ...
}
$myVar = new myClass();
$myVar->doSomething();
```

By default, attributes are public. You can define them as public, protected or private. That limits which class can directly access attributes.

- public : any class can access the attribute
- protected : acces for the class and any class that heritate the current one.
- private : acces limited to the class.

For protected and private access, you should add getters and setters for the attributes.

```
class myClass {
  ... private attribute;
  ...
}
```

As for classes in all languages, you can define methods that can be applied to instances of the class.

```
class myClass {
  public function myFunction(...) {
    ...
  }
  ...
}
```

$this is the current object
$self for the current class. Can be used for static methods or attributes.

Constructors are called when an instance is created.
Destructors are called when objects are removed.

```
function __construct() {
    ...
}
function __destruct() {
    ...
}
```

7.14.1.2.9 Including files

Usually, programs are not in a single file. You split your program in several files, and include the ones you need.

In PHP, there are 4 ways of importing files :

- require
- require_once
- include
- include_once

If file is missing, require will stop your program, include will not.
"_once" ensure the inclusion is done only once.

```
Require_Once("MyFile.php");
```

7.14.1.2.10 Libraries

There are many functions and many available libraries.

However you have to take care of some points :

- they should not be deprecated
- your PHP version has to be compiled with the right library
- the library is activated in the .ini file

7.14.1.2.11 HTTP exchanges

PHP exchanges with browser through variables. More specifically arrays.

- $_GET contains data according to a GET call
- $_POST contains data according to a POST call
- $_SERVER contains context elements

[Fig. 245] is an exemple of the way you can get parameters from a request. The URL set parameters. As it is a GET call, result can be found in array $_GET.

Browser call URL :

```
http ://myserver.com/action.php ?myAction=1&name=DUPOND
```

This is a GET call. So your file action.php should retrieve data like this :

```
<?php
$myAction = $_GET["myAction"]; // will be set to 1
$personName = $_GET["name"];  // will be set to DUPOND
?>
```

Fig. 246 : PHP get parameters from HTTP exchange

Reply to the browser is made through the output flow (print)

By default, with a HTTP exchange, reply will be a HTML stream. [Fig. 247] is an example of the way you can return a response.

```
<?php
...
print '<html lang="fr-fr">\n';
print "  <head>\n"; \\
print "    <title>}Hello World</title>\n";
print '    <meta charset="UTF-8"/>\n';
print "  </head>\n";
print "  <body>\n";
print "    <p>Hello World</p>\n";
print "  </body>\n";
print "</html>";
?>
```

Fig. 247 : PHP response

Sometimes, it is usefull to output something else than HTML code. It can be an XML document, a JSON document, ... By default, output is considered as HTML output, but you can change it by changing the headers response. [Fig. 248] is an example of header for a JSON response.

```
...
Header("Content-Type: application/json");
print "{'ok':1}";
```

Fig. 248 : PHP JSON response

[Fig. 249] is an example of header for a file response. It can be used when user wants to download data.

```
Header("Content-Type: application/msword");
Header("Content-disposition: attachment; filename=the-file-name.doc");
print $fileContent;
```

Replace "application/msword" by the mime-type you want.

Fig. 249 : PHP FILE response

Here are some examples of mime types you can use.

- text/html : a HTML file
- application/octet-stream : a text/binary stream
- application/msword : a word file
- ...

https ://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types give most of the mime types you can use.

7.14.1.2.12 Some usefull functions

- isset($variable) checks if a value has already been put in this variable
- is_array(...), is_int(...), ... check if variable are arrays, int, ...

7.14.1.2.13 Writing your own PHP pages

Let's start with the usual hello world. Here is the hello.php file :

```
<?php
print "Hello World !!";
?>
```

Or you can use this one if you want something more HTML compliant.

```
<?php \\
print "<html lang=\"fr-fr\">\n";
print "  <head>\n";
print "    <title>Hello World</title>\n";
print "    <meta charset=\"UTF-8\"/>\n";
print "  </head>\n";
print "  <body>\n";
print "    <p>Hello World</p>\n";
print "  /body>\n";
print "</html>";
?>
```

Put your PHP file in your HTTP server files location (htdocs, ...).
Call it with your browser : http ://localhost/.../hello.php

Ok, now let's try someting a bit ore complex.

In the materials for this practical work, you will find a HTML file called index.html.
This file aims at checking login/password, using a call to action.php with a method GET.
You will have to write the file action.php.

- You should get data from the form through the $_GET array.
- Check $_GET contains the "user" data
- Check if "user" data is "admin"
  - if true, reply with an "ok"
  - if false reply with a "wrong".

Create the file action.php and the PHP script.
Check it works.
Note the structure of the URL when you click on the button to call action.php.

Second step, switch to POST method. You should change it in index.html and in you action.php

When you check it, do you see a difference with the GET method in the URL.

7.14.1.3 Going a bit further...

We will use a function.
Copy CheckLDAP.php from the materials to your HTTP server doc location.

In CheckLDAP.php, depending on your configuration for php, the authenticate function will decide if it calls ldap functions or if it uses a fake authentication. However the authentication part is not defined.

We will use file “index.html” that we already wrote and that displays a login/password page. Change “action” attribute in the “form” so that it calls “action.php”.

Create a file “action.php”.

- Beginning of “action.php” add a PHP script that collects information from the form in index.html To collect informations, have a look to the attribute “method” in the form and select the rights way to collect them.
- Call the authenticate method in checkLDAP.php, and do not forget to include the file beginning of action.php
- according to the result, display informations. Remember you should respond with an HTML file.

Check it works.

Advertisement

Due to internal policy, it is possible LDAP is not usable any more.

7.14.2 About Symfony

Symfony, as Laravel, are PHP frameworks. [Fig. 250] is Symfony's logo.

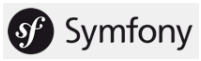


Fig. 250 : Symfony Logo

Symfony vas created in 2005 by SensioLabs. It is one of the most popular and used in France.

Symfony is free, but you can also pay to gain support. Symfony current version is version 7 since 2024. A new major version is available every 2 years. Have a look to https ://symfony.com/releases for the releases.

Symfony can use 2 ORMs, Doctrine and Propel. By default, the version we use is designed for Doctrine. Doctrine will manage the database through objects. So we will have to define objects that map the tables in the database.

7.14.3 Required tools

For this practical work, you will need :

- a PHP 8 interpretor (terminal configuration)
- Composer (php tools)
- Symfony
- a text editor to create pages
- JQuery file
- a PostgreSQL server

You should already have installed php and postgreSQL. JQuery should already been downloaded. You should install Symfony. Have a look to the annexes for that.

We will not need any HTTP server : Symfony will use its own server.

7.14.4 Checking tools

7.14.4.1 Check PHP configuration

Check your php interpretor in the terminal (your web and terminal version could be different) :

```
php -v
```

Then, check if php is compiled with the right modules

```
php -i
```

Check the pdo_pgsql module is activated (it should be in the listed modules). [Fig. 251] shows what you should have. Of course version can be different.

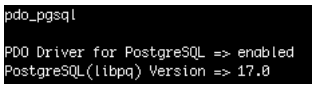


Fig. 251 : PHP with postgresql module activated

If it is not activated, activate it in the ini file. **Take care, the ini file for the web server and for the terminal are not always the same**. You can find the ini file location with this command :

```
php --ini
```

7.14.4.2 Install and Check Symfony

You will find Symfony installation instructions in the annexes.
You should install symfony-cli and composer.

Symfony implements tools to check if the environment is ok. Take care, there are no space char around colons chars. [Fig. 252] shows the result of this command.

```
symfony local :check :requirements
```

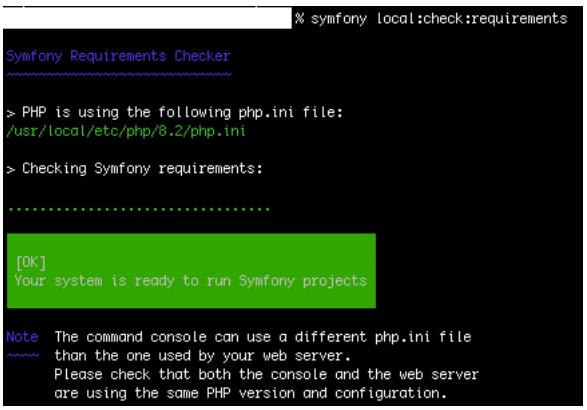


Fig. 252 : Symfony check requirements

7.14.5 The Symfony project

7.14.5.1 Creating project directory

Open your terminal/ command tool, go to the location where you want to create your project and ask Symfony to create the project structure. Depending on available versions, you can change it in the command line.

```
symfony new prwebSYMFONY --version="6.3.*" --webapp
```

[Fig. 253] shows the result of the command output. On the right of the figure there is also the project directory content you may have.

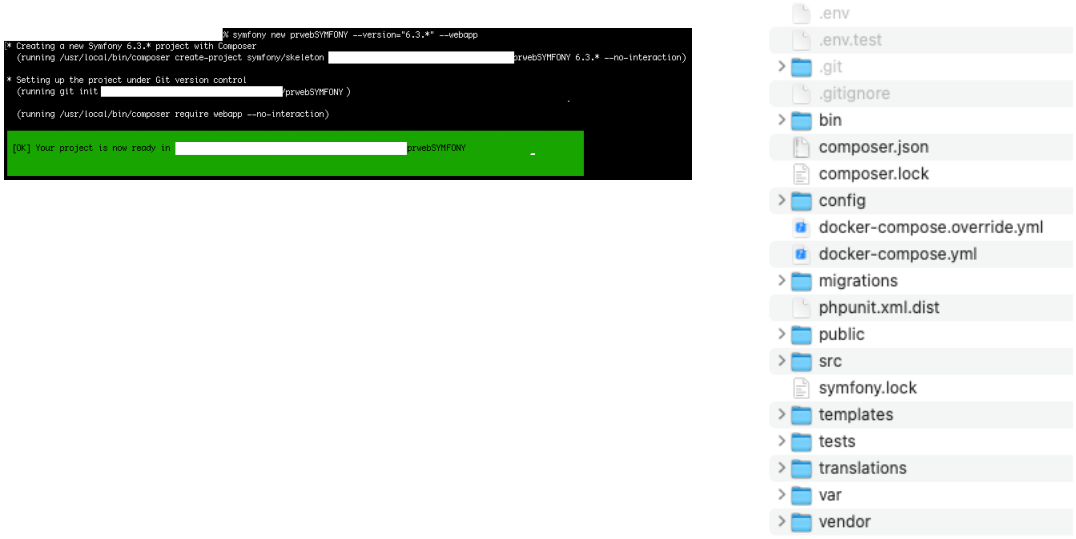


Fig. 253 : Composer creates project

What are the files/folders used for ?

- .env : is a hidden global file for configuration
- bin : for Symfony commands
- composer.json : the installed modules for the project
- config : configuration files
- migrations : database informations
- public : where we will put css, images, js files
- src : your files. The project source files. What you develop.
- templates : "HTML" templates for the application. It is not really HTML, call it TWIG.
- tests : for tests purpose

- translations : for multi-languages applications
- var : Symfony temporary files, like caches
- vendor : Symfony libraries

Symfony provides tools to manage authentication.
That means create users, manage password encryption, ...
For that, we need the Symfony security bundle.

In your terminal, go to your project, add doctrine annotation and the security bundle. Import them and check nothing is missing.

```
cd prwebSYMFONY
composer require doctrine/annotations
composer require symfony/security-bundle composer install
```

7.14.5.2 Get project informations

Maybe, having some informations about the project (Symfony version, ...) would be nice. Try this :

```
php bin/console about
```

This should give to you some informations about Symfony version, your app (App/Kernel) and PHP version, like in [Fig. 254],

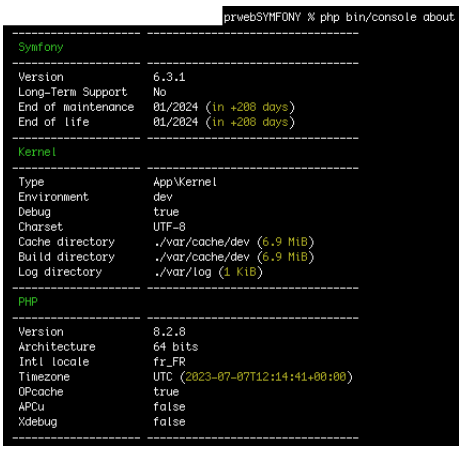


Fig. 254 : Check Symfony elements versions

Depending on your Symfony version, you may have an “Expired” error. That means that the Symfony Team do not maintain this version any more. You will not be able to upgrade it, for security reasons for example, even if your program will still go on running.

7.14.5.3 Running project

If everything is ok, we can start the server and check it is ok. There are 2 ways of doing this :

Launch server :

```
symfony server :start
```

(no space char between server and :)
And stop it with "CTRL C".

Run server in background :

```
symfony server :start -d
```

And stop it with :

```
symfony server :stop
```

In both case, Symfony compiles the project, check modules, and if everything is ok, launch an http server. [Fig. 255] shows some elements of the compilation.

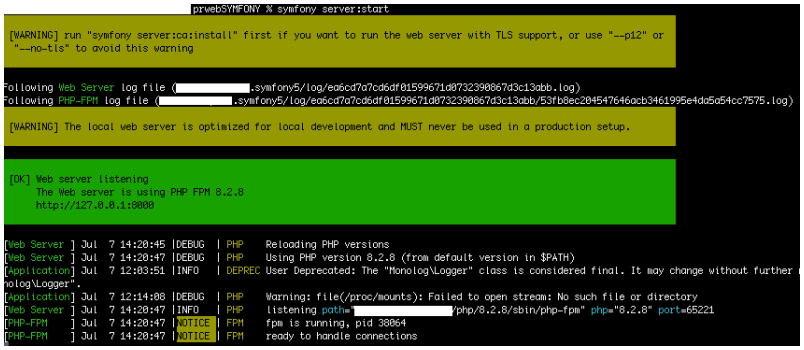


Fig. 255 : Start server

Have a look to the green area in [Fig. 255]. It explains which URL you should use to run the project.

Now, open your web browser with the following URL : http ://localhost :8000
[Fig. 256] shows the kind of result you may have. Maybe your version is a bit newer or older. That is not a problem. Do you see the message top of your page. It tells you that there is nothing defined for the project (that is true).

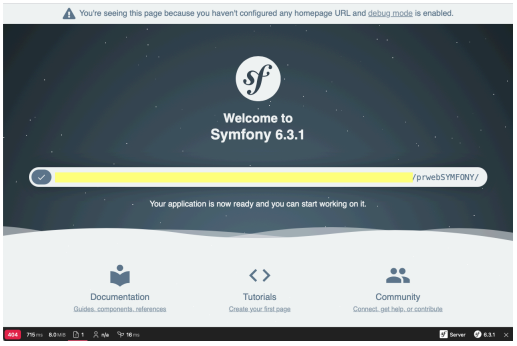


Fig. 256 : Connect to HTTP server and your Symfony project

7.14.6 Database connection configuration

Before defining entities, we need to configure our database connection.

First, we have to know which postgresql version we use. Use PgAdmin and connect to your postgreSQL server. Select your PostgreSQL connection in the left area. In the right area, in the menus, select the tab "Properties". This displays your server properties. You should find the server version in the "Version" line.
[Fig. 257] shows an example of a PostgreSQL server version in PgAdmin.

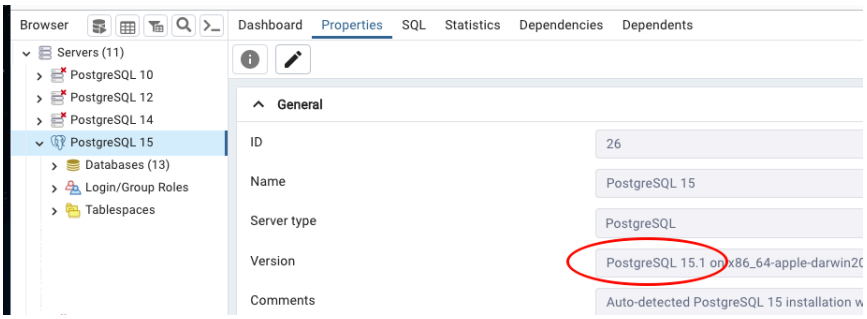


Fig. 257 : Get PostgreSQL server version

Ok, now we configure the Symfony environment file.
File **.env** is an hidden file. It is located at the root of your project. Open this file with your text editor. Have a look to the **DATABASE_URL** instruction. This is what you should change.

```
DATABASE_URL="postgresql ://login :password@localhost :5432/database ?serverVersion=version&charset=utf8"
```
DO NOT COPY/PASTE THIS LINE, change data in the file. There are no space chars.

where

- login is your database login (maybe prweb)
- password is your database password (maybe prweb)
- database is your database name (maybe prweb)
- version depends on your postgresql server version.

login, password, database's name are used to connect. server version is used to know which commands Symfony may use according to this version number.

Do not forget to save the .env file.

It is not required, but you can change the file doctrine.yaml in config/packages too.

7.14.7 Creating projet entities and repository

We want to manipulate the elements in the database. This manipulation can be done by using SQL request, getting data, and manipulating the informations, but this is not the way we want to use it.

First, ensure your database is created and available (SGBD server running, and database created).

We want to use an MVC model. That means we have to represent elements in the database as objects and manipulate these objects. For that, we have several things to do :

- we need to create entities in our application
- we need repositories to manage our entities
- we need views to display informations
- we need controllers to interact with the user

7.14.7.1 Entities and repositories - principles

First, we will create the link with the database by creating entities and repositories.

There are 3 ways of doing this :

- create entities files and ask Symfony to create tables in the database according to your definition
- create xml or yaml files describing your database and ask Symfony to create entities and tables
- use an existing database and ask Symfony to create entities

Very often, developers prefer the first way or the second way of doing this. For a small database it can be a nice solution (and we could have use it). However, when the database is a little bit more complex, it is often easier to create entities according to an existing database.

Let's talk about entity files contents.
Here is the structure of a php file to define an entity :

```php
<?php
namespace App\Entity;
use Doctrine\ORM\Mapping as ORM;
/**
* EntityName
* @ORM\Table(name="tableName")
* @ORM\Entity(repositoryClass="App\Repository\RepositoryName")
*/
class EntityName {
        ...
}
```

Some comments about this file :

- "namespace" explains the file content is an entity
- "use" tells we will use Doctrine as our ORM
- Now have a look to the comment located before the class definition :
  - First @ORM element indicates the name of the table in the database
  - Second @ORM element tells the entity will be managed by a repository called "RepositoryName"
- And finally the class, with the entity name.

Next, in a table, there are columns. They are defined as attributes in the entity class definition.
We have to take into account :

- Standart declaration for most of the attributes
- IDs
- External links to other entities / classes

Maybe you can have a look to :
https ://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/basic-mapping.html

Here is an example of column declaration :

```
class EntityName {
    /**
     * @var string|null
     * @ORM\Column(name="title", type="string",
                length=255, nullable=true)
     */
    private $title;
    ...
}
```

Some explanations about this example :

- "@var string|null" : the column is a string, and it may contain null values.
- Second line if for the database. Column name is "title", it is a string, its length is max 255 characters and it may contain null.
- last line is the attribute definition, as private. It's name is title.

Now let's try with an ID :

```
class EntityName {
    /**
    * @var int
    * @ORM\Column(name="id", type="integer",
                nullable=false)
    * @ORM\Id
    * @ORM\GeneratedValue
    */
    private $id;
    ...
}
```

In this example :

- "@ORM\Id", tells the column is the ID.
- "GeneratedValue", defines auto-increment.

GeneratedValue uses the default strategy to use auto-incrementation. You can change this strategy, but be sure of what you do.

And a last one : building a link with another entity.

```
class EntityName {
    /**
     * @var \OtherEntity
     * @ORM\OneToOne(targetEntity="OtherEntity",
                cascade={"persist"})
     */
    private $otherEntity;       ...
}
```

What do we add ?

- OtherEntity is another entity class
- "OneToOne", tells this attribute is linked to an unique OtherEntity entity. And vice-versa.

Here are the links you can create :

- OneToOne : one entity linked to another one, and this entity linked to the first one. The 2 linked entities use a OneToOne link.
- ManyToOne : one entity linked to another one, and this entity linked to many entities. The entity uses ManyToOne, and the other one uses OneToMany.
- OneToMany : one entity linked to many other ones, and these other ones linked to the first one. The entity uses OneToMany, and the other one uses ManyToOne.
- ManyToMany : one entity linked to many other ones, and these other ones linked to many entities too. The 2 linked entities use a ManyToMany link.

And the elements you will find in the declarations

- targetEntity="..." defines the class that manages the entity we are linked to
- cascade={"persist"} ensure that if we apply on operation on the targetEntity entity, also it is mapped to current attribute. Take care, cascade do not solve every link problem.
- mappedBy="..." defines the name, in targetEntity, of the attribute we are linked to
- inversedBy="..." is the reverse link of mappedBy
- @JoinColumns and @JoinColumn define columns names in the database

Keep in mind the use of cascade. Maybe it can be usefull in some declarations.

Most often, you do not have to define the link in both linked entities.

- If you use Unidirectional declaration, you have to declare the link in a file, the other link is implicit.
  In that case, you should not use mappedBy nor inversedBy.
- If you use Bidirectional declaration, you declare the link in both files.
  In that case, you have to use mappedBy and inversedBy.

Most often Unidirectional link can be used.
However, if you use reverse link, a bidirectional implementation may help you.

You can also define some informations.

JoinColumns will define informations about implementation.
It should be located at the end of the description, just before the attribute name definition.

```
* @JoinColumn(name="firstEntityId", referencedColumnName="id")
```

This declaration means current attribute will be implemented as a column named "firstEntityId". It is linked to a column "id" in the other entity.

Here are some examples you can use.

- [Fig. 258] and [Fig. 259] shows OneToOne examples
- [Fig. 260] shows a ManyToOne and a OneToMany example
- [Fig. 261] and [Fig. 262] shows a ManyToMany example

```php
class FirstEntity {
    ...
    /*
     * @ORM\OneToOne(targetEntity="SecondEntity")
     ...
    private $secondEntity;
    ...
}
```

```php
class SecondEntity {
    ...
}
```

Fig. 258 : OneToOne Unidirectional PHP example

```php
class FirstEntity {
    ...
    /*
     * @ORM\OneToOne(targetEntity="SecondEntity",
                mappedBy="firstEntity")
     ...
    private $secondEntity;
    ...
}
```

```php
class SecondEntity {
    ...
    /*
     * @ORM\OneToOne(targetEntity="FirstEntity",
                inversedBy="secondEntity")
     ...
    private $firstEntity;
    ...
}
```

Fig. 259 : OneToOne Bidirectionnal PHP example

```php
class FirstEntity {
    ...
    /*
     * @ORM\ManyToOne(targetEntity="SecondEntity",
                inversedBy="firstEntities")
     ...
    private $secondEntity;      ...
}
```

```php
use Doctrine\Common\Collections\ArrayCollection;
class SecondEntity {
    ...
    /*
     * @ORM\OneToMany(targetEntity="FirstEntity",
                mappedBy="secondEntity")
     ...
    private $firstEntity;      ...
    public function __construct() {
        $this->firstEntities = new ArrayCollection();
    }
    ...
```

Fig. 260 : ManyToOne and OneToMany PHP example

```php
use Doctrine\Common\Collections\ArrayCollection;
class FirstEntity {
    ...
    /*
    * @ORM\ManyToMany(targetEntity="SecondEntity")
    * @JoinTable(name="joinTableName",
    * joinColumns={@JoinColumn(name="firstEntityId",
            referencedColumnName="id")},
    * inverseJoinColumns={@JoinColumn(name="secondEntityId",
            referencedColumnName="id")})
    */
    private $secondEntities;
    ...
    public function __construct() {
        $this->secondEntities = new ArrayCollection();
    }
    ...
}
```

```php
class SecondEntity {
    ...
    }
}
```

Fig. 261 : ManyToMany PHP example

```php
use Doctrine\Common\Collections\ArrayCollection;
class FirstEntity {
    ...
    /*
    * @ORM\ManyToMany(targetEntity="SecondEntity",
    inversedBy="firstEntities")
    * @JoinTable(name="joinTableName")
    */
    private $secondEntities;
    ...
    public function __construct() {
        $this->secondEntities = new ArrayCollection();
    }
    ...
}
```

```php
use Doctrine\Common\Collections\ArrayCollection;
class SecondEntity {
    ...
    /*
    * @ORM\ManyToMany(targetEntity="FirstEntity",
    mappeddBy="secondEntities")
    */
    private $secondEntities;
    ...
    public function __construct() {
        $this->secondEntities = new ArrayCollection();
    }
    ...
}
```

Fig. 262 : ManyToMany PHP example

In these figures :

- @JoinTable tells
  - the link between the 2 tables is implemented through a table called joinTableName
  - joinTableName has 2 columns : firstEntityId and secondEntityId
  - there is an external link between firstEntityId and the primary key in FirstEntity : id
  - there is an external link between secondEntityId and the primary key in SecondEntity : id
- SecondEntitys entities will be managed in FirstEntity by $secondEntities
- $secondEntities is an array initialized when entity is created, through __construct()

There are 4 methods to implement the php files for the entities :

- Method 1 : Manual - From PHP to Database -
  You write php files, then you map them to the database
- Method 2 : Semi-Automatic - From PHP to Database -
  You ask help to Symfony to build php files. Then you map them to the database
- Method 3 : Semi-Automatic - From PHP to Database -
  You use YAML or XML files to describe data, Symfony build them. Then you map the files to the database
- Method 4 : Automatic - From Database to PHP -
  Using reverse engineering, you build PHP files from database.

We will use the last one, but you should have a look to the first ones too.

Questions

What are the 4 links you can use to implement foreign keys and link tables ?
Why do ManyToOne and OneToMany work together ?
Which link represents link tables ?
What is JoinTable used for ?

7.14.7.1.1 Creating manually entity files

This is not the method we will use. It is given only for documentation.

In src/Entity, you have create 1 file for each entity, or for each table it is the same : Book.php, Person.php and Borrow.php
Use the infrastructure of the tables to build entities Book, Person and Borrow by adding attributes and annotations.

[Fig. 263] shows an entity file content.

```php
/**
 * Book
 *
 * @ORM\Table(name="book")
 * @ORM\Entity(repositoryClass: BookRepository::class)
 */
class Book
{
    /**
     * @var int
     *
     * @ORM\Column(name="book_id", type="integer", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="SEQUENCE")
     * @ORM\SequenceGenerator(sequenceName="book_book_id_seq", allocationSize=1, initialValue=1)
     */
    private $bookId;

    /**
     * @var string
     *
     * @ORM\Column(name="book_title", type="string", length=256, nullable=false)
     */
    private $bookTitle;

    /**
     * @var string
     *
     * @ORM\Column(name="book_authors", type="string", length=256, nullable=false)
     */
    private $bookAuthors;


}
```

Fig. 263 : Entity

ORM commands tell Doctrine what to do.
You might take carre of :

- the object class name : Book, Person or Borrow
- @ORM\Table= the table name in database : book, person or borrow
- @ORM\Entity= defines it is an entity
- RepositoryName= the repository class name : BookRepository, PersonRepository or BorrowRepository
- in the class you should have attributes definitions
  @ORM\Column defines each column in the database

7.14.7.1.2 Ask Symfony to create files and to map them to the database

This is not the method we will use. It is given only for documentation.

Use this command to start the interactive creation :

```
php bin/console make :entity Item
```

Then answer the questions to build attributes in the database structure. id is built by default.
[Fig. 264] is an example of the attribute creation for Book. Do the same for Person and Borrow.
Note that this method also creates a Repository for the generated entity.

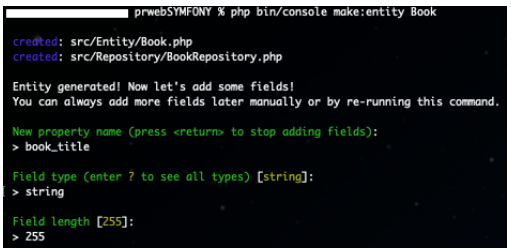


Fig. 264 : Interactive creation of an entity

7.14.7.1.3 Creating entity files with YAML or XML file

This is not the method we will use. It is given only for documentation.

We create a YAML file for each entity. Definitions are located in src/Resources/config/doctrine.
For example, for an entity FirstEntity, the description is in file FirstEntity.orm.yml
YAML definition describe fields to implement. Here is the example for Book :

```
App\Entity\Category :
    type : entity
    repositoryClass : App\Repository\BookRepository
    table : book
    id :
        book_id :
            type : integer
            generator :
                strategy : AUTO
    fields :
        book_title :
            type : string
            length : 255
        book_authors :
            type : string
            length : 255
```

When you've created the YAML files, you can generate the entity files with a Symfony command :

```
php bin/console doctrine :schema :update --force
```

7.14.7.1.4 Creating entities by importing database informations

This is the method we will use.
But do not use commands right now. We will use them a bit later.

NOTE : your database server must be running, and your database should be created. **Ensure your tables owner is the user you defined in the file .env, otherwise the import procedure will not complete**.

As our connection to the database is defined, we can ask Symfony to create entities. You can use the following command to create them.

```
php bin/console doctrine :mapping :import "App\Entity" annotation --path=src/Entity
```

This command creates entities according to the database schema. It generates 1 file / entity with the right annotations, except Repository link.

7.14.7.2 Create Entities and Repositories

We have to create entities from database. For that we will ask symfony to import them from database.

We also need a user to connect to the application. This user is not currently available in the database. So we will have to create this authentication user.

Note : For bug fixing reason, we will create authentication user first. If you create authentication user after creating entities, there might be a bug which cancels this creation. If you create authentication user first, bug does not happen.

7.14.7.2.1 Creating User for authentication

We do not want anybody to acces data. We need a user in the database and pages to authenticate, create user, delete user, ...

So, first create user. Symfony has a specific command for that :

```
php bin/console make :user
```

You have to give the entity name : User.
We store it in the database, otherwise we will have to manage files in the application to manage users.

Then we have to give the user authentication property. Let's choose "username". And we tell Symfony that passwords will be hashed because we use them to authenticate and we store them in database (otherwise it will not been encrypted in database).

[Fig. 265] shows the user creation.

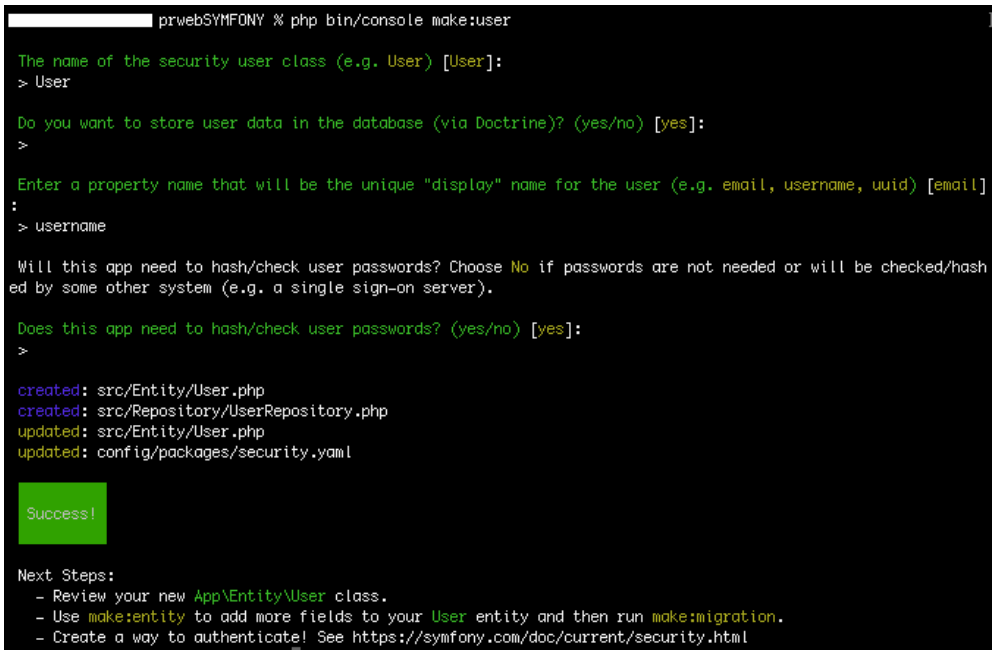


Fig. 265 : Symfony creates User

Let's have a look to the result :

- in src/Entity, you should have a new entity : User.
  This take into account the authentication field (the one you defined) and the password.
- in src/Repository, a UserRepository is created.
  To manage User.
- in config/packages/security.yam, file has changed to take user into account.
  - it added security section with password_hashers
    To manage password encryption
  - it added user in provider and firewalls
    To define authentication User to use.

We will manage authentication pages a bit later.

7.14.7.2.2 Create Entities and Repositories from database

Ok, now we can generate entities and repositories from existing tables in the database.

Import them from database.

```
php bin/console doctrine :mapping :import "App\Entity" annotation --path=src/Entity
```

[Fig. 266] shows the entities creation with the Symfony command.

```
[redacted] prwebSYMFONY % php bin/console doctrine:mapping:import "App\Entity" annotation --path=src/Entity
Importing mapping information from "default" entity manager
  > writing src/Entity/Book.php
  > writing src/Entity/Borrow.php
  > writing src/Entity/Person.php
```

Fig. 266 : Doctrine creates entities

Open the directory src/Entity. Open the entities files with a text editor and have a look.
[Fig. 267] shows a generated entity file content.

```php
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Book
 *
 * @ORM\Table(name="book")
 * @ORM\Entity
 */
class Book
{
    /**
     * @var int
     *
     * @ORM\Column(name="book_id", type="integer", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="SEQUENCE")
     * @ORM\SequenceGenerator(sequenceName="book_book_id_seq", allocationSize=1, initialValue=1)
     */
    private $bookId;

    /**
     * @var string
     *
     * @ORM\Column(name="book_title", type="string", length=256, nullable=false)
     */
    private $bookTitle;

    /**
     * @var string
     *
     * @ORM\Column(name="book_authors", type="string", length=256, nullable=false)
     */
    private $bookAuthors;


}
```

Fig. 267 : Doctrine generated entity

What do we find in this file ?

- namespace is the one defined in Doctrine
- we use Doctrine as an ORM. We use @ORM to define ORM informations.
- Then we have the Entity book definition and its link to the database.
  - @ORM\Table gives the table name,
  - @ORM\Entity confirms it is an entity.
- The Book class contains book informations
  - @var gives the column type for Symfony
  - @ORM\Column gives informations about the column in the table
  - @ORM\Id tells the column is the primary key
  - @ORM\GeneratedValue tells incrementation mechanism
  - @ORM\SequenceGenerator tells sequence informations

Also ensure that the entity namespace declaration (should be around line 3) looks like this :

```
namespace App\Entity;
```

What about entity links ?

Have a look to Borrow.php to understand how it works.

In Borrow.php, you should find the same kind of elements than for books. To manage links, like in [Fig. 268], @ORM \ManyToOne tells this entity is linked to another entity (Book or Person) and that the linked entity may be linked to as many borrows as needed.

Have a look to “About Frameworks” to understand meaning of OneToOne, ManyToOne, OneToMany and ManyToMany.

```
/**
 * @var \Book
 *
 * @ORM\ManyToOne(targetEntity="Book")
 * @ORM\JoinColumns({
 *   @ORM\JoinColumn(name="book_id", referencedColumnName="book_id")
 * })
 */
private $book;
```

Fig. 268 : entity attribute with link to other entity

7.14.7.2.3 Entities sequence management

With Doctrine / Symfony there are several ways to manage auto increment of IDs.

- AUTO (default) : Use what should be the best strategy, according to Doctrine, depending on the database server. Usually :
  - IDENTITY for MySQL, SQLite and MsSQL
  - SEQUENCE for Oracle and PostgreSQL.
- SEQUENCE : Use database sequence for ID generation.
- IDENTITY : Use special identity columns in the database.
  - MySQL/SQLite/SQL Anywhere =>AUTO_INCREMENT
  - MSSQL =>IDENTITY
  - PostgreSQL =>SERIAL
- TABLE : Would use a separate table for ID generation. Not yet implemented!
- NONE : Generated, by your code.
- UUID (Doctrine {>}=2.3, and database server that manages UUID) : use the built-in generator.

By default, when we asked Symfony to generate entities, They were generated as SEQUENCE.

7.14.7.2.4 Updating entities to use repositories

By default, there is no link defined between entities and repositories.
We have to tell each entity that they are managed by a repository.

In each entity, juste before the class definition, in the line @ORM\Entity, in the header section before the class definition, add the repository link like this :

```
@ORM\Entity(repositoryClass="App\Repository\RepositoryName")
```

Take care : do not copy the line from PDF : generated quotes might not be the right ones.

Where RepositoryName is the name of the repository you want to use for the entity. That name should be the Entity Name followed by "Repository".
For example, for entity Book, repository name should be BookRepository.
Result should look like in [Fig. 269].

```
/**
 * Book
 *
 * @ORM\Table(name="book")
 * @ORM\Entity(repositoryClass="App\Repository\BookRepository")
 */
class Book
```

Fig. 269 : Link Entities and Repositories

7.14.7.2.5 Generate getters and setters in entities

We must be able to use the entities attributes. As they are defined as private, you can't. So, we have to define getters and setters for that.

Of course, you can generate them by yourself, but Symfony includes a command to generate them. [Fig. 270] shows the result of the command.

```
php bin/console make :entity --regenerate
```

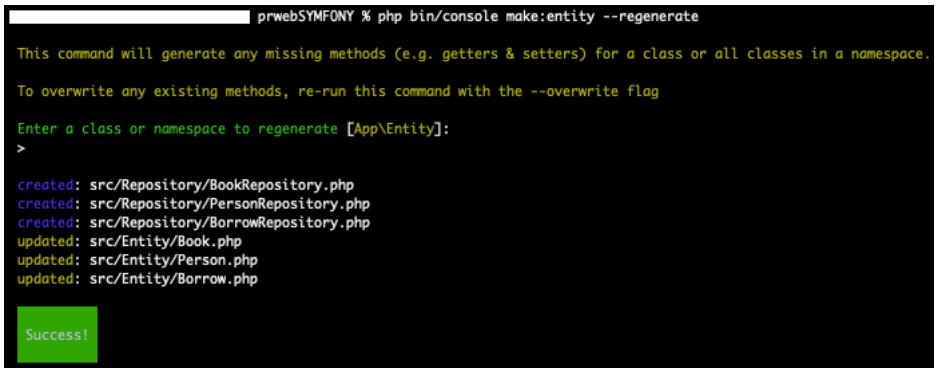


Fig. 270 : Generate getters and setters

Have a look to your entities, and check getters and setters where generated.

7.14.7.3 Creating repositories

Did you use the command to generate Getters and Setters in your entities ?
If so, have a look to the repository directory.
Do you notice the new files ?

The new files in "Repository" are the ones you defined in your entities.
They were created when you asked to regenerate entities.

When you generated the getters and setters, it also generated the repositories (have a look to your command results). It tooks the name in the annotations and generated the required functions.

Open directory "repository".
Open file BookRepository.php with a text editor. You should have something like [Fig. 271].

```php
<?php

namespace App\Repository;

use App\Entity\Book;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\ORM\Exception\ORMException;
use Doctrine\ORM\OptimisticLockException;
use Doctrine\Persistence\ManagerRegistry;

/**
 * @extends ServiceEntityRepository<Book>
 *
 * @method Book|null find($id, $lockMode = null, $lockVersion = null)
 * @method Book|null findOneBy(array $criteria, array $orderBy = null)
 * @method Book[]    findAll()
 * @method Book[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class BookRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Book::class);
    }

    /**
     * @throws ORMException
     * @throws OptimisticLockException
     */
    public function add(Book $entity, bool $flush = false): void
    {
        $this->_em->persist($entity);
        if ($flush) {
            $this->_em->flush();
        }
    }

    /**
     * @throws ORMException
     * @throws OptimisticLockException
     */
    public function remove(Book $entity, bool $flush = false): void
    {
```

Fig. 271 : A repository

What do we find in this file ?

- file inclusion refers to thee managed entity
- class header defines default accessible methods for the repository with @method. If you add a method, add its reference in the header.
- a default constructor
- 2 methods : add and remove. They save / remove an entity to/from the database
- examples of “how to build methods” in the comments

7.14.7.4 Using migrations

Now, we have the main infrastructure.

What happens if you modify the database ?
Could it be possible to define the database data and reload them when you want ?
For instance when you are creating your app and you want to do some tests ?

Symfony implements mechanisms to manage database structure and default data.

First we will build a map of the database structure. Use this command for that :

```
php bin/console make :migration
```
Once more, there are no space chars around colons

Have a look to your project directory.
You should find a directory "migrations". This directory contains every database evolution. Files are named as Version and the creation timestamp.
If you add a column, remove something, ... in your project you should also use a migration command. Modifications will be saved in that directory, in a new file.

Have a look to the file in this directory. In this file, function **down(Schema $schema)** is called to remove the database, function **up(Schema $schema)** is called to create it.

To apply migrations use :

```
php bin/console doctrine :migrations :migrate
```

This applies the migration files to your database. Have a look to your database.
Maybe some things have change. For example, default incrementation is removed on IDs. There are also new tables and sequences that manages migration versions, and other informations.

If you want to compare your migration files and the current database (this is not required right now), you can use this :

```
php bin/console doctrine :migrations :diff
```

This should add a new migration file that takes into account modifications you did. That is why itt is not required, you didn't change anything since last migration.

7.14.7.5 Fixtures

Then, we want to manage data. This mechanism is called “fixtures”. Let’s import the module.

```
composer require --dev doctrine/doctrine-fixtures-bundle
```

Have a look in your src directory. You should have a new directory “DataFixtures”.
In this directory, you should find a file AppFixtures.php, which will be used to manage our data.
Open this file. To manage data, we have to add instructions to **load(ObjectManager $manager)**.

First, import entities and repositories. Top of the script, add imports like this :

```
namespace App\DataFixtures;
use App\Entity\Person;
use App\Entity\Book;
use App\Entity\Borrow;
use App\Repository\PersonRepository;
use App\Repository\BookRepository;
use App\Repository\BorrowRepository;
use Doctrine\Bundle\FixturesBundle\Fixture;
```

Then, modify function **load** to add your data.

For that, we can create a list of data to insert.
Then we loop on data, create objects, set data, and save data to the database.
To save an object, we **persist** it in the manager.

The **ObjectManager** is our ORM. It manages the link between objects and database.

Here is what it should look like (feel free to change data or add some) :

```
public function load(ObjectManager $manager) {
    $people = [ ['Pierre', 'KIMOUS', '2000-02-04'],
        'Jean-Yves', 'MARTIN', '1963-08-12'],
        'Jean-Marie', 'NORMAND', '1991-04-16'] ];
    foreach ( $people as $index =>$aPerson) {
        $aDate = \DateTime ::createFromFormat('Y-m-d', $aPerson[ 2 ]);
        $person = new Person(); // Create object
        $person ->setPersonFirstname($aPerson[0]);
        $person ->setPersonLastName($aPerson[1]);
        $person ->setPersonBirthdate($aDate);
        $manager ->persist($person); // save object to database
        $people[ $index ][ 'object' ] = $person; // Keep created object for future use
    }
    $manager->flush();
...
}
```

NB : we keep the object $person in the column ‘object’ of the array for future use.

Do the same for books :

```
$books = [ [ 'Astérix chez les Bretons', 'René Goscinny, Albert Uderzo' ],
           [ 'La Foire aux immortels', 'Enki Bilal' ],
           [ 'Les Passagers du Vent, Volume 1', 'François Bourgeon' ],
           [ 'Fairy Tail, Vol 1', 'Hiro Mashima' ],
           [ 'Reincarnated as a Sword, Vol 1', 'Yuu Tanaka' ] ];
foreach ( $books as $index =>$aBook) {
    ...
    $manager ->persist($book);
    ...
}
$manager->flush();
```

To create Borrow objects, we use the objects we kept in arrays to build the links.

```
$borrows = [ [2, 4, '2021-07-15', '2021-09-01'],
             [1, 2, '2021-08-01', NULL],
             [3, 3, '2021-10-01', NULL],
             [2, 1, '2021-10-02', NULL] ];
foreach ( $borrows as $index =>$aBorrow) {
    $borrow = new Borrow();
    $borrow ->setPerson($people[ $aBorrow[ 0 ]-1 ][ 'object' ]);
    $borrow ->setBook($books[ $aBorrow[ 1 ]-1 ][ 'object' ]);
    $aDate = \DateTime ::createFromFormat('Y-m-d', $aBorrow[ 2 ]);
    $borrow ->setBorrowDate($aDate);
    if ($aBorrow[ 3 ]!= NULL) {
        $aDate = \DateTime ::createFromFormat('Y-m-d', $aBorrow[ 3 ]);
        $borrow ->setBorrowReturn($aDate);
    }
    $manager ->persist($borrow);
}
$manager->flush();
```

Ok, now check it works.

```
php bin/console doctrine :fixtures :load
```

Reply “yes” to reinitialize database data.

The auto-increment elements (sequences) are not reinitialised, so your IDs may increase each time you launch it.

Questions

What are repository used for ?
How do you link entities and repositories ?
What are migrations ?
How do you reload data to test you app ?

7.14.8 Creating controllers and views

To interact with the app user (through the browser), we need scripts that manages requests (controllers), and views to display something. There are 2 ways to create controllers :

- manually : You create by yourself the controller files, methods, ...
- with Symfony : You use Symfony commands to create the controllers and views.

Of course, we use the second one.

7.14.8.1 Creating default files for controllers and views

Ok, Let's start with the page that manages table "person", or entity "Person".

7.14.8.1.1 Generate controllers and views

We want to display the table list, add a person, modify a person, ...
Do you remember what means **CRUD** ?
Ok, Let's try to generate all that.

```
php bin/console make :crud Entity
```

So, that should look like :

```
php bin/console make :crud Person
```

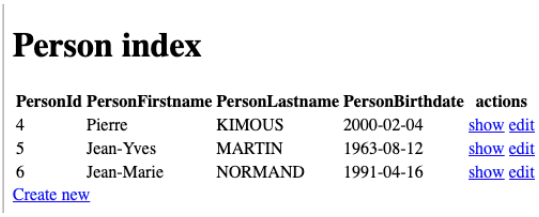
Validate controller's name. Then generate tests if you want (but you are not required to).
Have a look to [Fig. 272] for the instructions sequence.

Fig. 272 : Generate CRUD with Person

What is the result ?

- We generated a controller to manage the entity Person.
- We generated a default form to manage a person in Form/PersonType.php
- We created a directory “person” in “templates” and many twig files (views) in this directory.

Ok, let’s try.
Use URL http ://localhost :8000/person
[Fig. 273] is the result you may have.



Person index

PersonId	PersonFirstname	PersonLastname	PersonBirthdate	actions
4	Pierre	KIMOUS	2000-02-04	show edit
5	Jean-Yves	MARTIN	1963-08-12	show edit
6	Jean-Marie	NORMAND	1991-04-16	show edit

Create new

Fig. 273 : Entity Person default list

Ok. Now, have a look to the controller to understand what it does. Open file “PersonController.php” in controllers. [Fig. 274] shows the beginning of the class definition.

```php
#[Route('/person')]
class PersonController extends AbstractController
{
    #[Route('/', name: 'app_person_index', methods: ['GET'])]
    public function index(PersonRepository $personRepository): Response
    {
        return $this->render('person/index.html.twig', [
            'people' => $personRepository->findAll(),
        ]);
    }
```

Fig. 274 : Person Controller

What do we find ?

- have a look to the line before the class definition. It tells you every route in this class (understand URL you can call) starts with “/person”.
- Next, have a look to the line before the “public function index”. It tells you :
  - route is “/”. Understand this function will be called when route is “/person/” or “/person”
  - name is “app_person_index”, that means when referring to it, we can use this name instead of an URL.
  - method is GET. That means it replies to GET calls, and only to GET calls.
- Now have a look to the function body.
  - Function name doesn’t matter. You can call it dummy if you want, nothing will change.

- This function returns a Response (understand HTTP response).
- It uses a parameter : a PersonRepository, that means an instance of a repository that manages Person.
  You can use/add the parameters you need (Request for example) and Symfony will try to give you what you need when it compiles the script.
- As a response it calls render (understand "I give you a view to display"), with the twig file name, and an array as a parameter list for the view.
  For this function, there is 1 parameter. It is called "people" and contains the lines in table "Person" (use findAll() in personRepository).

If you want to check available routes (the ones in controllers, and more), you can use this command :

```
php bin/console debug :router
```

Now, have a look to PersonRepository in "src/Repositories".
You will find its content in [Fig. 275].
Do you see the methods defined just before the class definition? These methods are defined by default by the ServiceEntityRepository. That means you don't have to define them.
Next, in the class, we find some useful methods. Of course, you can use them too.
And of course implement yours.

```php
/**
 * @extends ServiceEntityRepository<Person>
 *
 * @method Person|null find($id, $lockMode = null, $lockVersion = null)
 * @method Person|null findOneBy(array $criteria, array $orderBy = null)
 * @method Person[]    findAll()
 * @method Person[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class PersonRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Person::class);
    }

    public function save(Person $entity, bool $flush = false): void
```

Fig. 275 : Person Repository

Now open directory "templates" located at the root of the project.
Have a look to the file "base.html.twig".

"base.html.twig" is a templatefile. It is used to produce HTML pages.
You can find its content in [Fig. 276].

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org,
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
        {% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Fig. 276 : Base TWIG file

This is a HTML file, with twig commands.
These commands define blocks ({% block xxx %}...{% endblock %}) to be inserted in this file.
That means symfony will create the resulting HTML content by filling this file with the blocks values.

At last, have a look to the file index.html.twig in templates/person.
You will find its content in [Fig. 277].

```
{% extends 'base.html.twig' %}

{% block title %}Person index{% endblock %}

{% block body %}
    <h1>Person index</h1>

    <table class="table">
        <thead>
            <tr>
                <th>PersonId</th>
                <th>PersonFirstname</th>
                <th>PersonLastname</th>
                <th>PersonBirthdate</th>
                <th>actions</th>
            </tr>
        </thead>
        <tbody>
        {% for person in people %}
            <tr>
                <td>{{ person.personId }}</td>
                <td>{{ person.personFirstname }}</td>
                <td>{{ person.personLastname }}</td>
                <td>{{ person.personBirthdate ? person.personBirthdate|date('Y-m-d') : '' }}</td>
                <td>
                    <a href="{{ path('app_person_show', {'personId': person.personId}) }}">show</a>
                    <a href="{{ path('app_person_edit', {'personId': person.personId}) }}">edit</a>
                </td>
            </tr>
        {% else %}
            <tr>
                <td colspan="5">no records found</td>
            </tr>
        {% endfor %}
        </tbody>
    </table>

    <a href="{{ path('app_person_new') }}">Create new</a>
{% endblock %}
```

Fig. 277 : Index TWIG file

In file index.html.twig, instructions start by telling it extends “base.html.twig”. When we load index.html.twig, we refer to base.html.twig, and we give the blocks value for this file.
Then it defines the blocks (“title” and “body”), some of the ones referred in base.html.twig.

Have a look to [Fig. 276] to see where they will be placed.

Now, still in index.html.twig, have a look to the way a loop is built. We use {% ... %} to define an instruction like if, for, ...
The loop instruction uses variable "person" to iterate on array "people". Do you remember the name of the parameter in PersonController ?
To use a value, we use {{ ... }}. In our case the value is our variable (person) and the name of the requested field.

Still in index.html.twig, have a look to the href attribute of tag "a".
This is the way we can call a method in a controller with its name (app_person_show for instance) and give to this method a parameter (the person to show).

Ok.
Now, you can generate CRUD for entities Book and Borrow too.

7.14.8.1.2 View for authentication

User is an entity but it is used to authenticate users in our app. So, thee way we manage it is a bit different because we do not need the same methods and views.

There is a specific command to manage authentication views and manage users.

```
php bin/console make :aut
```

Anser you want a login page (choice [1]).
Give a class name. For example LibraryAuthenticator (a name is required, there is none by default).
Maybe we can select the default name for thee security controller (SecurityController).
Let it generate a logout URL.
You can manage support for "remember me", or not. If you choose it you will have to define how it works.

[Fig. 278] shows the make :auth command.

Fig. 278 : make :auth command

7.14.8.2 Customise twig pages

Now a bit of presentation. Wouldn't it be nice to use Bootstrap library ?

7.14.8.2.1 Static elements and base

Static files (images, css, js, ...) should be placed in "public", at the root directory of your project.

Use what you did for the HTML Bootstrap practical work and copy directories "bootstrap", "css", "js" and "img" in the directory "public". They are also in the "materials" in hippocampus.
[Fig. 279] should be the result.

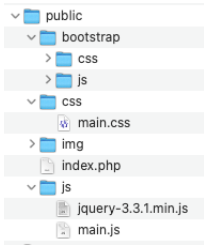


Fig. 279 : public Directory

Now, we want our bootstrap files to be used by every html generated file.
The best way is to include them in the main template : base.html.twig.
[Fig. 280] shows how you can do that.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">

        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="/bootstrap/css/bootstrap.css">
        <script type="text/javascript" src="/js/jquery-3.3.1.min.js"></script>
        <script src="/bootstrap/js/bootstrap.min.js"></script>
        <link href="/css/main.css" type="text/css" rel="stylesheet" />

        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.or
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
        {% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Fig. 280 : Base TWIG file modified

This is the same kind of inclusion than the one we used in the bootstrap practical work.
There is one main difference, URLs start with a /.
That will avoid, when there is a route prefix (like "/person" in PersonController) to get a wrong route for JS files, CSS files, ...

7.14.8.2.2 Customize index

Of course, in index.html.twig, you might also change table headers, and other elements.
[Fig. 281] shows how you can do it.

```
{% block body %}
        <div class="py-3">
            <div class="container">
                <div class="row">
                    <div class="col-md-12">
                        <h2 class="">List of users</h2>
                    </div>
                </div>
                <div class="row">
                    <div class="col-md-12">
                        <div class="table-responsive">
                        <table class="table table-striped">
                            <thead>
                                <tr>
                                    <th>User #</th>
                                    <th>Firstname</th>
                                    <th>Lastname</th>
                                    <th>Birthdate</th>
                                    <th></th>
                                </tr>
                            </thead>
                            <tbody>
                            {% for person in people %}
                                <tr>
                                    <td>{{ person.personId }}</td>
                                    <td>{{ person.personFirstname }}</td>
                                    <td>{{ person.personLastname }}</td>
                                    <td>{{ person.personBirthdate ? person.personBirthdate|date('Y-m-d') : '' }}</td>
                                    <td class="text-center">
                                        <a href="{{ path('app_person_show', {'personId': person.personId}) }}">show</a>
                                        <a href="{{ path('app_person_edit', {'personId': person.personId}) }}">edit</a>
                                    </td>
                                </tr>
                            {% else %}
                                <tr>
                                    <td colspan="5">no records found</td>
                                </tr>
                            {% endfor %}
                            </tbody>
                        </table>
                        <a href="{{ path('app_person_new') }}">Create new</a>
                        </div>
                    </div>
                </div>
            </div>
        </div>
{% endblock %}
```

Fig. 281 : Person index twig file

The point you have to take care of, is the way you can use paths. Have a look to the links in your page. They look like this :

```
href="{{ path('app_person_edit', {'personId' : person.personId}) }}"
```

Do you remember the routes name in Person controller ?
Find the function with route "app_person_edit". This route has a parameter, personId.

In the twig file, when you use path{...}, you ask symfony to call the right route with the parameter you give, and to generate the appropriate URL with these elements.

[Fig. 282] shows the result.

List of users

User #	Firstname	Lastname	Birthdate	
1	Pierre	KIMOUS	2000-02-04	show edit
2	Jean-Yves	MARTIN	1963-08-12	show edit
3	Jean-Marie	NORMAND	1991-04-16	show edit

Create new

Fig. 282 : Person index with bootstrap

Could we use such method with buttons ?
Yes. But buttons must be in forms. And, for each button, we have to use the attribute **formaction** to give the route. [Fig. 283] shows how you can do that.

```
<form>
<button name="show" class="btn" formaction="{{ path('app_person_show', {'personId': person.personId}) }}"><img src="/img/show.png" alt="edit" class="icon" /></button>
<button name="edit" class="btn" formaction="{{ path('app_person_edit', {'personId': person.personId}) }}"><img src="/img/edit.png" alt="edit" class="icon" /></button>
<button name="delete" class="btn" formaction="{{ path('app_person_delete', {'personId': person.personId}) }}"><img src="/img/delete.png" alt="delete" class="icon" /></button>
</form>
```

Fig. 283 : Buttons with route's call

And maybe you can use presentation elements you did, like using nav (in base.html.twig), add containers, css table responsive elements, replace the link "create new" by a button in footer, ...
[Fig. 284] shows our current result.

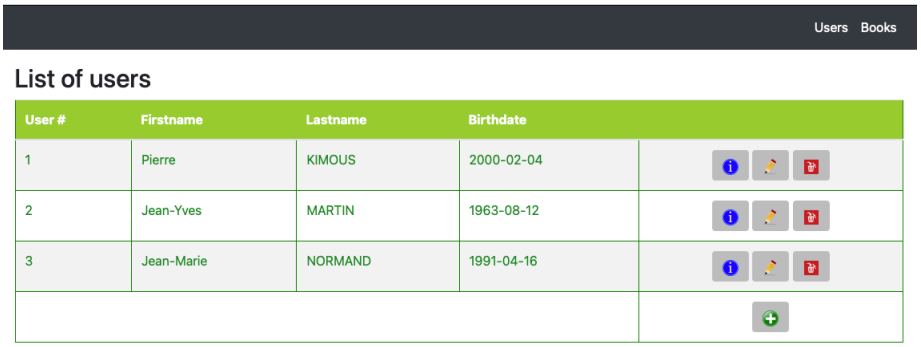


Fig. 284 : Person view

If your server is stopped, run it. Then try your pages with http ://localhost :8000/person.
Show person info, edit then, go back to list, ...
Do not try to add, remove or update a person, we didn't implement these functions yet.

Now we have to customise the other files.

7.14.8.2.3 Customise show

Let's continue with show.

Customise it with bootstrap containers, row, ... [Fig. 285] shows what you can do.

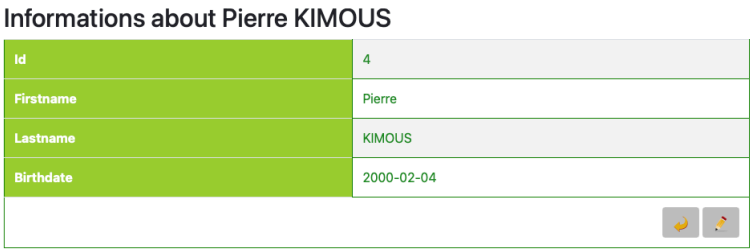


Fig. 285 : Person infos page

7.14.8.2.4 Customise edit

Let's continue with edit.
Oh, this is not the same kind of page building. It calls '_form.html.twig'.

So, we have both files to customize.

Let's start with edit.html.twig
Main structure is the same as the show file. So feel free to customise it as you want. The main difference is the call to '_form.html.twig'. Do not remove it. You can add parameters that will be used in _form twig file.
[Fig. 286] shows what can be done in edit.html.twig

```
{% block body %}
        <div class="py-3">
            <div class="container">
                <div class="row">
                    <div class="col-md-12">
                        <h2 class="">Informations about {{ person.personFirstname }} {{ person.pers
                    </div>
                </div>

                {{ include('person/_form.html.twig',
                    {'button_label': 'Save',
                     'button_image': '/img/save.png',
                     'button_alt': 'save',
                     })
                }}

                <div class="row">
                    <div class="col-md-12">
                        <form>
                        <button class="btn" formaction="{{ path('app_person_index') }}"><img src="/
                    </div>
                </div>
            </div>
        </div>
{% endblock %}
```

Fig. 286 : Person edition twig file

Now, have a look to '_form.html.twig'. Some commands you may use :

- {{ form_start(form) }} defines the form tag, and {{ form_end(form) }} ends it.
- {{ form_widget... }} defines default elements to display a form, an entity field, ...
- {{ form_error... }} defines tags to display an error message, ...
- {{ form_rest(form)) }} defines tags for... what you forgot in the form.

[Fig. 287] shows what you can do as a first version. Wwe use bootstrap to customize elements.

```
{{ form_start(form) }}
                <div class="row">
                    <div class="col-md-12">
                        {{ form_errors(form) }}
                    </div>
                </div>

                <div class="row">
                    <div class="col-md-12">
                        <div class="table-responsive">
                        <table class="table table-striped">
                            <tbody>
                                <tr>
                                    <th scope="row">Firstname
                                    {{ form_errors(form.personFirstname) }}</th>
                                    <td>{{ form_widget(form.personFirstname) }}</td>
                                </tr>
                                <tr>
                                    <th scope="row">Lastname
                                    {{ form_errors(form.personLastname) }}</th>
                                    <td>{{ form_widget(form.personLastname) }}</td>
                                </tr>
                                <tr>
                                    <th scope="row">Birthdate
                                    {{ form_errors(form.personBirthdate) }}</th>
                                    <td>{{ form_widget(form.personBirthdate)  }}</td>
                                </tr>

                            </tbody>
                            <tfoot>
                                <tr>
                                    <td colspan="2" class="text-center">
                                    <button class="btn"><img src="{{ button_image }}" alt="{{ button_alt }}" class="icon" /></button>
                                    </td>
                                </tr>
                            </tfoot>
                        </table>
                        </div>
                    </div>
                </div>
                {{ form_rest(form) }}
{{ form_end(form) }}
```

Fig. 287 : _form.html.twig customisation

And the result is in [Fig. 288]

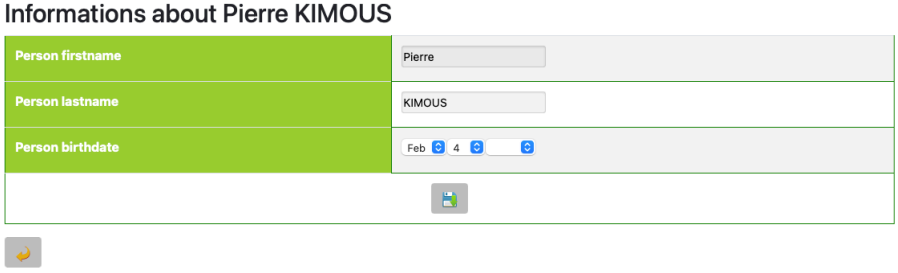


Fig. 288 : Edit person

Oh no. What is this date display format ?
Let's change it in src/Form/PersonType.php
In function buildForm, are defined field to display in the Person forms. By default, the field uses Symfony default definition, that depends on field type.

We want the date to be displayed as a single text, not as a selection.
Change personBirthdate definition like the one in [Fig. 289].
You will also have to include "DateType" in the "use" part of the file.

```php
namespace App\Form;

use App\Entity\Person;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\OptionsResolver\OptionsResolver;

class PersonType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('personFirstname')
            ->add('personLastname')
            ->add('personBirthdate', DateType::class, ['widget' => 'single_text',])
        ;
    }

    public function configureOptions(OptionsResolver $resolver): void
    {
        $resolver->setDefaults([
            'data_class' => Person::class,
        ]);
    }
}
```

Fig. 289 : PersonType redefinition

Refresh the page. Result is in [Fig. 290]

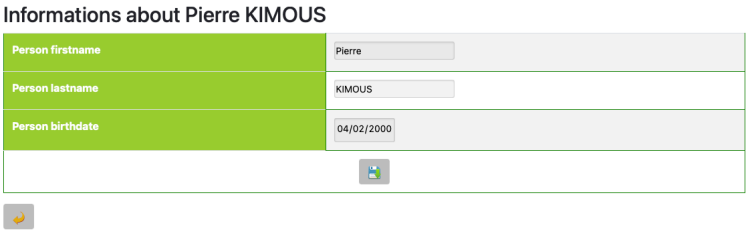


Fig. 290 : Person infos page

Make some change. Click on the save button.
If it is ok, you should return to the list.
Modifications you've done should have been saved.

WHAT ? But we didn't write anything for that !!!
Who decided to save my data and when ?

Have a look to the PersonControl, and more specifically to the edit function. This method is called when we ask to edit a person. It is also used when you submit the form by clicking on the “save” button.
Have a look to the parameters. We have the person we are editing and the repository as parameters.
Now have a look to the instruction. What happens if form is submitted and submission is valid?
We add (understand save) the person, the redirect to the index (the list).

7.14.8.2.5 Customise new

Let’s continue with new.
Have a look to the script. It is quite similar to the edit one. It calls ‘_form.html.twig’ too.

You already did it for edit. It should be the same kind of work. Result is in [Fig. 291].

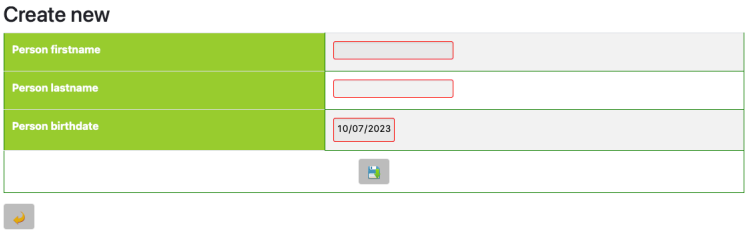


Fig. 291 : New Person

Fill the form, save user. You should then come back to the list with new user.
[Fig. 292] and [Fig. 293] show the result.

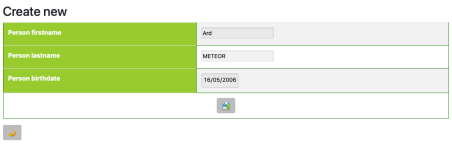


Fig. 292 : New Person fill form

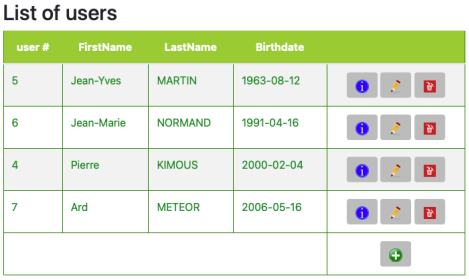


Fig. 293 : New Person is added

7.14.8.2.6 Delete user

Have a look to the delete function in the PersonController.

Route to delete a person is the same as the one we use for show, except it is in POST mode.
So, this means we have to method POST to send data in the twig file (person/index.html.twig).
For that, we can use attribute **formmethod** in the button to change the way request is sent. This will change the method, only if we select this button.

```
<button name="delete" class="btn" formmethod="post" formaction=...
```

Next, have a look to delete function in PersonController. It needs a valid CSRF token.
We will not manage these tokens in this practical work.
So, comment the line that check CSRF validity, and the end of the block.
You should keep as operational the 2 lines that remove person and flush it to database, and the line with redirectToRoute that display the result.

7.14.8.2.7 Try it.

Ok, now let's try it.
Ensure your server is launched and try the functions.

DO NOT TRY TO DELETE A USER WHO ALREADY BORROWED BOOKS
Because of foreign keys, deleting such users will result in an integrity error and will be refused.
To delete such users you first have to delete the borrowed books.

Questions

What is a twig file ?
How do you call a route from a twig file ?
How do you send a parameter to a route in a twig file ?
How do you manage forms ?
Why is the '_form' twig file common to edit and new ?
How can you call a route from a form ? How does ORM Doctine to save an object in the database ?

7.14.8.3 Creating a second set of pages : the Book pages

Ok. Now, let's do the same with books.

You might have already use the command to create CRUD for Books.
We have to apply the same kind of modifications as for the Person files.
[Fig. 294], [Fig. 295], [Fig. 296] and [Fig. 297] show the results.

Fig. 294 : List Books

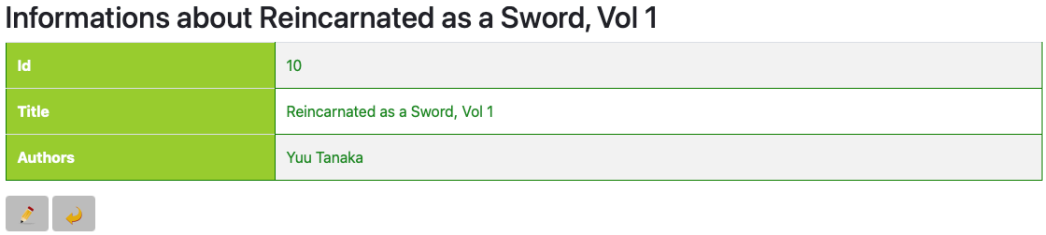


Fig. 295 : Book informations

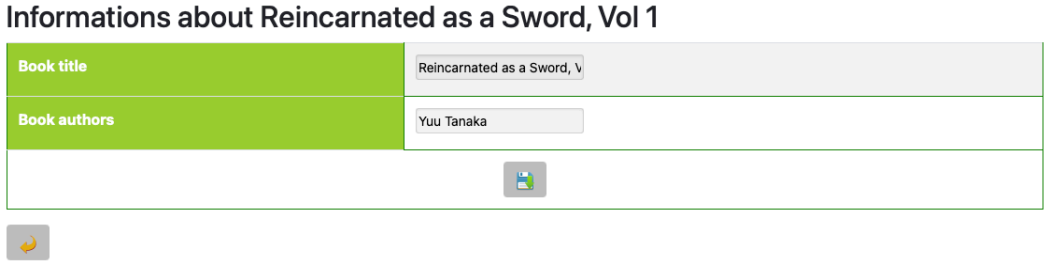


Fig. 296 : Edit Book

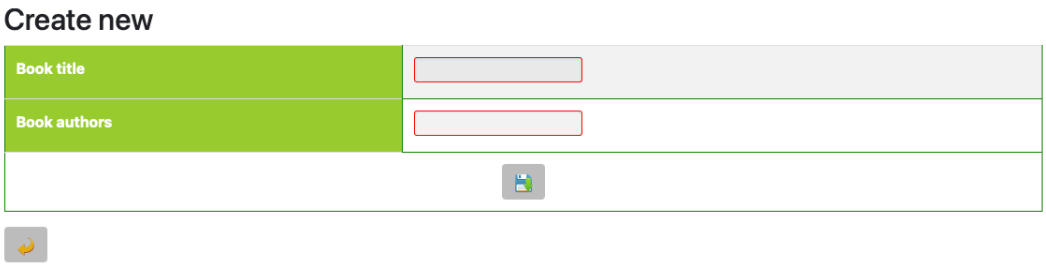


Fig. 297 : New Book

Did you also modify the delete method in BookController?

You should also try them to be sure you did not forgot something.

Do not try to remove books, for the same reason as Person.

7.14.8.4 Borrowing books

We want to know which book was borrowed by which user.
And of course user may borrow books, return them, ...

You might have use the symfony command to create the CRUD files for Borrow. If not, do it now.
That should have created your controller, and some templates files.

Now, let's display the **user's borrowed books in the page person/show**. What do we need to do?

- in PersonController, when calling person/show page, we should get the user's borrowed books from the BorrowRepository.
- in the person/show template page, add a call to a borrow termplate that displays books.
- Add methods in BorrowRepository to get user's borrowed books.
- Add tools in the borrowed template to return a book and borrow a new one.

7.14.8.4.1 Add methods to BorrowRepository

First we have to add a method in the BorrowRepository class to get a users's borrow list. In the comments in BorrowRepository, there is an example of what to do to implement a new method. We create our request using the example.

[Fig. 298] shows an example of the implementation you could use. Keep in mind that the attributes we manipulate are the Entities ones, not the database ones. So, when you create your request, take care to the names you use.

```php
/**
 * @return Borrow[] Returns an array of Borrow objects
 */
public function findByPerson(Person $person): array
{
    return $this->createQueryBuilder('b')
        ->andWhere('b.person = :person')
        ->setParameter('person', $person)
        ->orderBy('b.borrowDate', 'DESC')
        ->getQuery()
        ->getResult()
    ;
}
```

Fig. 298 : Get user borrowed books request in the repository

We create a request. Field person is equal to a parameter " :person". Parameter person's value is $person. We order results by borrowing date. The 2 last instructions launch query and get results. The function returns a user's borrowed books list (an array).

Remember to use Person top of the file or it will not be recognized as a parameter.

We will implement other methods when required.

7.14.8.4.2 Implement Controller and Templates

Let's start templates with "show person". We want to display user's borrowed books.

You might already have created CRUD elements for borrow, you should have a BorrowController, and a directory "borrow" in the twig templates, with twig files to edit, show, ...

When we display an user (understand command show on an user), we want to display the user's borrowed books. So, we need to feed the person/show twig file with the borrowed books. Then in this person/show twig file we have to display the borrow list.

First, the PersonController, the route "show". We only have to add the parameters we need when we call the template : user's borrowed books, and books. [Fig. 299] shows an example of the implementation you could use.

```php
return $this->render('person/show.html.twig', [
    'person' => $person,
    'borrows' => $borrowRepository->findByPerson($person),
    'books' => $bookRepository->findAll(),
]);
```

Fig. 299 : Add parameters when we call show

Of course, ensure borrowRepository and bookRepository are in your function parameters and imported (use) top of the file.

Now, let's display the borrow list from the show twig file.
Maybe we can add this after the table that displays the person informations.

The easiest way to do this is to include the borrowed book list. That means we have to include a twig file located in borrow.
Do you remember the way we include twig files to edit a person or a book ? For our need, We have to create a specific file in borrow.

[Fig. 300] shows how you can call the borrow list.

```
            </div>

            {{ include('borrow/_list.html.twig',
                {'button_label': 'Borrow',
                 'button_image': '/img/plus.png',
                 'button_alt': 'Borrow',
                 })
            }}

        </div>
    </div>
{% endblock %}
```

Fig. 300 : Call borrow/ _list template from person/show template

Then, we have to create the borrow/ _list.html.twig file in the templates to list borrowed books.

You can use the borrow/index.twig.html file as an example to build it.
We need the bootstrap elements and a tfoot element to borrow a book.

Some considerations :

- for each borrowed book, we add a button to return it.
  We give the borrow ID to manage it.
- In the tfoot element, we use a select tag to display the books list, and we display a button to add the selected book as a new borrowed one.
  We send the person ID as a parameter. We will have to get the book ID from the select tag.

[Fig. 301] shows how we built ours.

```
<div class="row">
    <div class="col-md-12">
        <div class="table-responsive">
        <table class="table table-striped">
            <tbody>
                <tr>
                    <th>Id</th>
                    <th>Book</th>
                    <th>Borrow Date</th>
                    <th>Return</th>
                </tr>
            </thead>
            <tbody>
            {% for borrow in borrows %}
                <tr>
                    <td>{{ borrow.borrowId }}</td>
                    <td>{{ borrow.book.bookTitle }}</td>
                    <td>{{ borrow.borrowDate ? borrow.borrowDate|date('Y-m-d') : '' }}</td>
                    <td class="text-center">
                    {% if borrow.borrowReturn %}
                        {{ borrow.borrowReturn|date('Y-m-d') }}
                    {% else %}
                        <form method="post">
                        <button name="return" class="btn" formaction="{{ path('app_borrow_return', {'borrowId': borrow.borrowId}) }}">
                            <img src="/img/return.png" alt="edit" class="icon" />
                        </button>
                        </form>
                    {% endif %}
                    </td>
                </tr>
            {% else %}
                <tr>
                    <td colspan="4">no records found</td>
                </tr>
            {% endfor %}
            </tbody>
            <tfoot>
                <tr>
                    <td colspan="5" class="text-center">
                        <form method="post">
                        <select name="bookId">
                        {% for book in books %}
                            <option value="{{ book.bookId }}">{{ book.bookTitle }}</option>
                        {% endfor %}
                        </select>
                        <button class="btn" formaction="{{ path('app_borrow_add', {'personId': person.personId}) }}">
                            <img src="{{ button_image }}" alt="{{ button_alt }}" class="icon" />
                        </button>
                        </form>
                    </td>
                </tr>
            </tfoot>
        </table>
        </div>
    </div>
</div>
```

Fig. 301 : borrow list

Ok, now the actions to launch in the borrowController. We have 2 actions to take into account :

- when we want to return a book
- when we want to borrow a new book

First, returning a book. We define a button in a form (method is POST) for each not returned book. The button has a formaction to a path (we used app_borrow_return) and the borrowed ID as a parameter.

Have a look to [Fig. 302] that shows our button's definition.

```
<form method="post">
<button name="return" class="btn" formaction="{{ path('app_borrow_return', {'borrowId': borrow.borrowId}) }}">
    <img src="/img/return.png" alt="edit" class="icon" />
</button>
</form>
```

Fig. 302 : Return book button script

Our buttons require to be managed in borrowController.

We need to implement a route with name "app_borrow_return". That means :

- we change the "return" field in the borrow object to the current date. We save it through the repository.
- we go back to the show template.
- we need some parameters for the function :
  - A Borrow parameter, computed by Symfony with the borrowId request parameter
  - The repositories

Same kind of requirements to add a new borrowed book. have a look tho the tfoot part of [Fig. 301].

- we have the person ID, that symfony will translate as a Person.
- we have the book ID in the request.
- we have to create a new Borrow object and save it to the database.
- we go back to the show template.

We have to get the book from the request parameters. The request parameter has a request field that contains data sent with the request. You can acces it by :

- $request->request->get(element name) for a GET request
- $request->request->post(element name) for a POST request

That means we need the request as a parameter to get the book ID from the GET part.
Next, we have to get the book from the bookRepository. We use find for that.
The instructions will look like these :

```
$bookId = $request->request->get("bookId");
$book = $bookRepository->find($bookId);
```

Next, we build a new Borrow element, set its parameters and save it to the database with the borrowRepository.

```
$borrow = new Borrow();
$borrow->setBook($book);
$borrow->setPerson($person);
$borrow->setBorrowDate(new \DateTime());
$borrowRepository->save($borrow, true);
```

[Fig. 303] shows an example of the scripts you can write.

```php
private function backToShow(Person $person, BookRepository $bookRepository, BorrowRepository $borrowRepository): Response
{
    if ($person != null) {
        // Back to show
        return $this->render('person/show.html.twig', [
            'person' => $person,
            'borrows' => $borrowRepository->findByPerson($person),
            'books' => $bookRepository->findAll(),
        ]);
    }

    return $this->render('person/index.html.twig', [
        'people' => $personRepository->findAll(),
    ]);
}

#[Route('/{borrowId}/return', name: 'app_borrow_return', methods: ['POST'])]
public function returnBorrow(Request $request, Borrow $borrow, BookRepository $bookRepository, BorrowRepository $borrowRepository): Response
{
    if ($borrow != null) {
        $borrow->setBorrowReturn(new \DateTime());
        $borrowRepository->add($borrow, true);  // Return date is NOW
    }

    return $this->backToShow($borrow->getPerson(), $bookRepository, $borrowRepository);
}

#[Route('/{personId}/add', name: 'app_borrow_add', methods: ['POST'])]
public function add(Request $request, Person $person, BookRepository $bookRepository, BorrowRepository $borrowRepository): Response
{
    $bookId = $request->request->get("bookId");
    $book = $bookRepository->find($bookId);

    if (($person != null) && ($book != null)) {
        // Create new Borrowed book
        $borrow = new Borrow();
        $borrow->setBook($book);
        $borrow->setPerson($person);
        $borrow->setBorrowDate(new \DateTime());

        $borrowRepository->add($borrow, true);
    }

    return $this->backToShow($person, $bookRepository, $borrowRepository);
}
```

Fig. 303 : Return and Add scripts in borrowController

Maybe you can check it works.

7.14.8.5 Navigating

When application is launched, it could be a good idea to navigate between pages.
Maybe we could use what you did in [Fig.27] and [Fig.28] and set them in the main template, in base.html.twig. You only have to replace the href link by the appropriate route name, {{ path('app_person_index') }}) for example.
[Fig.304] shows an example of page you should display.

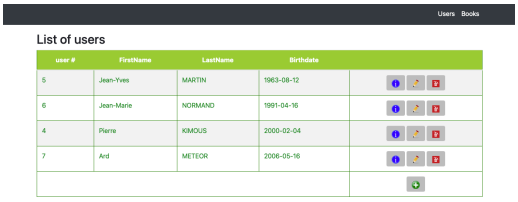


Fig. 304 : navigation

Switch between the pages to check it works.

7.14.8.6 Login page and user management

Login page is in src/templates/security/login.html.twig

As it extends the base.html.twig file, it includes the navigation bar. Maybe you can copy the base file and replace the content of the login file, and remove the navigation part, of course. The login.html.twig should be a full HTML file that does not extend base.html.twig and where there is no navigation bar. You can also customize the page as you want.

Next, we should start with the login page. That means, we need a route to "/" that is redirected to the login page. Quite simple, we only have to create a method for route "/" that redirects to "app_login" in the SecurityController.

[Fig. 305] shows the corresponding method.

```php
#[Route(path: '/', name: 'app_index')]
public function index(): Response
{
    return $this->redirectToRoute("app_login");
}
```

Fig. 305 : redirect to login page

OK, but who can connect ? We need registered users in the database. That means we have to create users with their password in table user. Registered password are encrypted. We have to be able to generate them. We need a specific controller for that.

```
php bin/console make :controller Registration
```

We need a UserPasswordHasherInterface to hash the password and a UserRepository to create user. We create user, we hash the password, we save the user in the database and we redirect to the login page.

[Fig. 306] shows our controller function.

```php
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class RegistrationController extends AbstractController
{
    #[Route('/registration', name: 'app_registration')]
    public function index(UserPasswordHasherInterface $passwordHasher, UserRepository $userRepository): Response
    {
        // Create user 1 as admin
        $userId = 1;
        $user = $userRepository->find($userId);
        if ($user == null){
            $user = new User();
            $user->setUsername("admin");
            $plaintextPassword = "admin";

            // hash the password (based on the security.yaml config for the $user class)
            $hashedPassword = $passwordHasher->hashPassword(
                $user,
                $plaintextPassword
            );
            $user->setPassword($hashedPassword);

            $roles = array('ROLE_ADMIN');
            $user->setRoles($roles);
            $userRepository->add($user, true);
        }

        return $this->redirectToRoute("app_login");
    }
}
```

Fig. 306 : registration creates default user admin

Ok, try it. http ://localhost :8000/registration
You should have the login page.

Have a look to the database and check there is an user with id 1, username is admin, password is… something hashed.
So let's try "admin", "admin" (or the password you chose").
… and … there is an error, like in [Fig. 307].

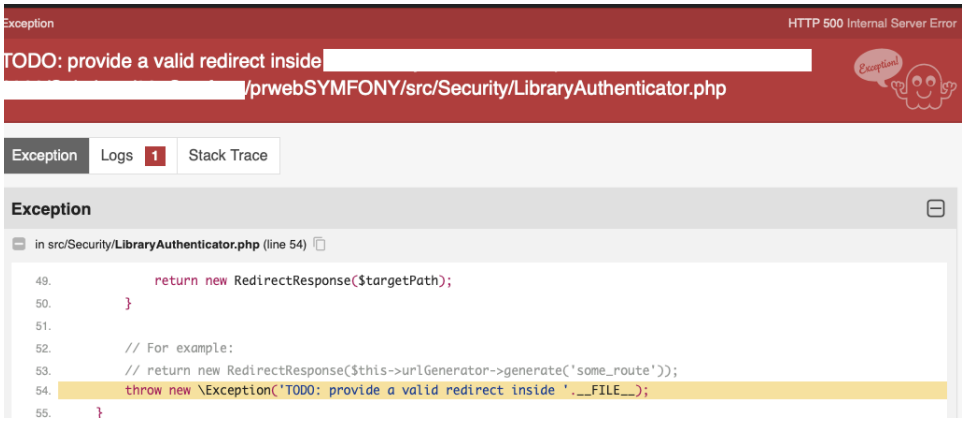


Fig. 307 : Connexion error

Ok. Let's have a look to LibraryAuthenticator.php
And change the function to the one in [Fig. 308].

```php
    public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $
    {
        if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
            return new RedirectResponse($targetPath);
        }

        // For example:
        // return new RedirectResponse($this->urlGenerator->generate('some_route'));
        return new RedirectResponse($this->urlGenerator->generate('app_person_index'));
//          throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
    }
```

Fig. 308 : Connexion error fixed

Reconnect to http ://localhost :8000
Connect with “admin” / “admin”.
... And you should have the page with the list of persons.

7.14.8.7 Logout page

As we login, we should also be able to logout.

Change the nav bar to include a logout button that route to app_logout.
Technically, the route to app_logout should never been called in SecurityController because it should be interpreted by symfony. Information is managed in the security yaml file in directory config/packages.

Questions

Why do we need a RegistrationController ?
What is the SecurityController used for ?
Where du you configure the page url routing when you log in ?

7.14.9 Summary

Here is a summary of the main instructions we used for the project

```
# Create project
symfony new prwebSYMFONY --version="6.3.*" –webapp
cd prwebSYMFONY

# Add tools
composer require doctrine/annotations
composer require symfony/security-bundle
composer install

# Configure database connection
php bin/console about
## modify .env
## ensure database is loaded and runs

# Add entities
php bin/console make :user
php bin/console doctrine :mapping :import "App\Entity" annotation --path=src/Entity
## add repository link in entities
php bin/console make :entity --regenerate

# Save migration
php bin/console make :migration
php bin/console doctrine :migrations :diff
php bin/console doctrine :migrations :migrate

# Add fixtures
composer require --dev doctrine/doctrine-fixtures-bundle
## change AppFixtures.php
php bin/console doctrine :fixtures :load

# Create controllers and views
php bin/console make :crud Person
php bin/console make :crud Book
php bin/console make :crud Borrow

php bin/console make :auth
php bin/console make :controller Registration
```