T7 - MSc Pool

T-POO-700

Mobile

Bootstrap

1.2

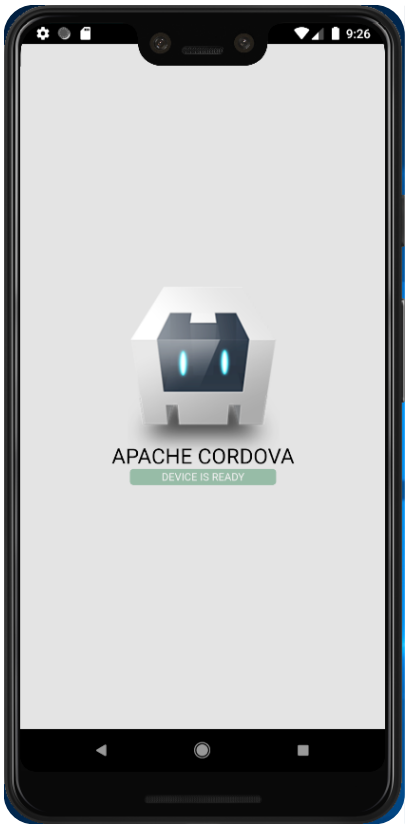
{EPITECH.}

Cordova is a cross-platform mobile application framework using `CSS3`, `HTML5` and `JavaScript`.
It creates **hybrid** applications, which are neither native nor purely based on the web stack.
You will experience here its simplicity, and will understand why it is one of the most used mobile cross-platform frameworks today.

Step 1 - Installing the environment and first application

Install and create the project from the official Cordova documentation

Follow the documentation for configuration for Android to the Setting up an Emulator section, to get an Android emulator.



Emulation does not work on the Windows Ubuntu Bash. If you are working on Windows, install the Windows version of npm and use **powershell** for all your commands.

Make sure to add the environment variables with the paths corresponding to **YOUR** installation.

For your emulator, set:

- Android API 28
- `Cold boot` as boot option

At this point, you should be able to launch your application on the emulator with a simple command.

```
Terminal
~/T-POO-700> cordova run android
```

If you can not start your emulator try to:
- start the emulator
- launch a native application (like the text messaging app for instance)
- quit the previous application
- launch the `run` command

To reuse the work you have already done, let's add `Vue.js` to the project, since Cordova allows it.
You will need the following dependencies:

```
Terminal
~/T-POO-700> npm install -g view view-cli webpack webpack-cli && npm install -D
view view-template-compile axios
```

You can now boot your project `Vue.js` in your Cordova application:

```
Terminal
~/T-POO-700> init view webpack [project-name]
```

To complete your installation, answer the questions that prompt.
Then, delete the content of the `www/` folder .
Because `Vue.js` is a web-based framework, the *index.html* file located at the root of your project contains only the minimum configuration necessary for the web browser.
You need to finetune it:

1. add the necessary configuration for a mobile usage, adding tags with the attributes you will need to associate Cordova with `Vue.js` ;
2. since we use the file generated by `Vue.js`, you must also add the script *cordova.js* to your *index.html* file ;
3. configure `Vue.js` to generate its files in the folder where Cordova will search the application pages, modifying the *build* rule in the *config/index.js* :
4. add the target platform of Cordova.

Compile, launch and test your application!

```
Terminal
~/T-POO-700> cordova platform add android --save
```

```
Terminal
~/T-POO-700> npm run build
[...]
~/T-POO-700> cordova build android
[...]
~/T-POO-700> cordova run android
```

As for the basic Cordova application, launch your emulator before the `cordova run android` command

Now that your project is complete study the tree of your application and identify the key elements:

- router
- components
- ...)

STEP 2 - FRONT CREATION

Let's now develop our own features, adapting the *Authentication* feature of the todo list application to a mobile use.
The queries to your API will also be made with **axios**, so you can reuse what you did previously.

You have not done it previously?? It's never too late to get the *Authentication* bootstrap done!

We can split the front into three separate parts: authentication views, common views and managers views.

Use the same routes as for Bootstrap "Authentication" (you will need the same settings).

Don't forget that for a mobile app to be enjoyable to use, it's important to keep in mind:
- the size of the screen
- the amount of information on the screen (a 6-button navbar is unconceivable)
- accessibility of the click (the top is harder to reach than the bottom of the screen for example)

Your interface must be functional, consistent, appealing and ergonomic.

AUTHENTICATION

As for your website, you will need to create two views for your user to authenticate:

- a login page ;
- a registration page (accessible from the login page).

API-dependent authentication will also need to store `the XSRF token`, `id`, and `role` of your user here, so you'll need to use local storage in the same way that you have already done it.

For the ergonomics of your application, do not hesitate to take inspiration from existing applications that you are used to using.

Do not forget to hash the password before sending it to the server!

COMMON VIEWS

Three views are shared by all users:

- the main page that contains the list of tasks called *tasksList* ;
- a detail page of a task, called *taskDetail* ;
- a profile management page called *profile* corresponding to the profile of the connected user.

Before starting to create these pages, let's handle the way we access them.
On a mobile there are several ways to make menus to navigate between different pages.
Here you must create a `menu sidebar` or `burger menu`. This menu will only contain three links for an employee (the *taskDetail* page being a sub-page, it will not be directly accessible):

- the main page on which the user will be automatically redirected after connection ;
- the *profile* page ;
- a link to disconnect, redirecting to the login page.

Main page It displays a list of all existing tasks, with at least their title, description and status.

Clicking on one of these tasks must redirect the user to the *taskDetail* subpage.
Additional information must appear on this page, such as the users assigned to the task.
The detail of a task must also allow the user to interact with it.

You will need to add a button for the user to assign/withdraw from a task, if and only if he has the skill associated with the task.
A second button should allow the user to change the status of the task.

For the sake of ergonomics, each of your sub-pages should contain a back button to return to the previous page

Profile Page The user must be able to change his/her profile information on this page.

Manager views

Managers being also employees, they will have access to the same pages, plus some additional elements.

The managers must be able to create a new task on the *tasksList* page .
On the *taskDetail* page, managers must be able to assign and remove a user to the job, delete a job, and edit a job.

They will also have access to two additional pages and one sub-page:

- a page that lists users called *usersList* ;
- a sub-page that details a user called *userDetail* ;
- a page listing skills called *skillsList*.

List of skills On this page, in addition to the skills list, you must implement a `Create a skill` button. Since creating a skill requires only one parameter, it's an opportunity to use the native dialogs on the OS. Clicking on this button will bring up a window of dialog with a prompt with a field "name of the skill".

Step 3 - Offline mode

One of the main problems of mobile development is the use of the application when the user is not connected to the internet.

Data storage

To allow the user to view the pages of the application in offline mode, you need to store some data on your server in the local storage of the user's phone so that you can display it when you can not make API requests.
There are several ways to store phone information; you must here set up a NoSQL database through LokiJS.

Depending on the role of your user and the different features available, you will not need to store all the data.
Employees must have access to:

- the *list of tasks* with the assigned users ;
- their *management of profile*.

Managers have the same accesses to which are added:

- the *management of skills* page ;
- the *User Management* page.

It's up to you to choose the structure of your database, avoiding data duplication and saving phone memory space.

Each time you open your application with an internet connection, the local data must be updated, but priority is given to the server.
You need to first make your API request and then update your local data through the API response or your next GET request

Interactions with users

Your data is accessible offline by the users but they can not interact with it.
To overcome this problem, give them the possibility to make changes in the local storage.

The last point is now to send these changes to the server when connecting to the Internet.

To cope with this, we will need to keep in mind the different requests that should have been sent if the user had an internet connection.
To store the queries, use a FIFO list (First In, First Out) to avoid any conflicts.

The local storage only accepts strings, you'll need JavaScript's `JSON.stringify` function.

Then, when the user connects to the internet, display the “spinner dialog” and launch the execution of the various queries.
Once all queries are complete, the “spinner dialog” should disappear.

Let’s take this opportunity to use the plugin cordova-plugin-progress-indicator.

In case of error during the sending of the requests, cancel the execution and resume the data stored on the server.

What would happen if changes are made to the website before the update of the mobile actions?