



---

## Web technology

---

**Laurent BOSSART  
Danial JOUKAR**

22 Août 2022

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                  | <b>2</b>  |
| <b>2</b> | <b>Cahier des charges</b>                            | <b>2</b>  |
| <b>3</b> | <b>Choix techniques</b>                              | <b>3</b>  |
| 3.1      | Back end . . . . .                                   | 3         |
| 3.2      | Front end . . . . .                                  | 3         |
| <b>4</b> | <b>Front end</b>                                     | <b>4</b>  |
| 4.1      | Hiérarchie des fichiers . . . . .                    | 4         |
| 4.2      | Fonctionnement de React . . . . .                    | 4         |
| 4.3      | Illustration des pages . . . . .                     | 9         |
| <b>5</b> | <b>Back end</b>                                      | <b>15</b> |
| 5.1      | Model . . . . .                                      | 15        |
| 5.2      | Serializer . . . . .                                 | 16        |
| 5.3      | Views . . . . .                                      | 17        |
| 5.3.1    | UserCreate . . . . .                                 | 17        |
| 5.3.2    | UserLogin . . . . .                                  | 17        |
| 5.3.3    | UserLogout . . . . .                                 | 17        |
| 5.3.4    | DownloadCleo . . . . .                               | 17        |
| 5.3.5    | RunCleo . . . . .                                    | 18        |
| 5.4      | Mise en ligne . . . . .                              | 18        |
| <b>6</b> | <b>Manuel d'installation pour utilisation locale</b> | <b>18</b> |
| <b>7</b> | <b>Principales difficultés</b>                       | <b>19</b> |
| <b>8</b> | <b>Conclusion</b>                                    | <b>19</b> |

## 1 Introduction

Ce projet consiste à développer une application Web responsive. Plus spécifiquement, un site permettant l'accès aux applications médicales d'aide au diagnostic des maladies de scoliose/ostéoporose.

Les objectifs du projets sont :

- Développer une plateforme web et évolutive avec Django
- Créer/gérer une base de données possédant divers informations sur les utilisateurs
- Gérer les inscriptions, connexions et droits d'admin
- Héberger/déployer les applications médicales avec un affichage des résultats en fonctions des paramètres choisis par l'utilisateur
- Anonymiser et chiffrer les données médicales

Le choix sur le back end est imposé, il s'agit du *framework Django*. En ce qui concerne le *front end*, nous avions libre arbitre. Nous avons décidé d'utiliser *ReactJS* pour sa simplicité et sa grande documentation. De plus, il existe beaucoup d'applications web utilisant *Django* et *Reactjs* ensemble, ce qui témoigne de leur bon fonctionnement.

## 2 Cahier des charges

Pour ce projet, la cahier des charges à été définis avec les éléments suivants :

1. Lister et analyser le contenu du site existant.
2. Effectuer la maquette du site.
3. Analyse modèle de données MCD.
4. Gestion des droits d'admin, connexions et inscriptions à la plate-forme.
5. Intégration des applications sur le site.
6. Anonymisation des données.
7. Gestion portefeuilles d'utilisateurs.
8. Réalisation du visuel du site.
9. Déployer le site sur un serveur distant.

Au terme de ce travail, nous pouvons compter que six des neufs étapes ont été réalisées, et qu'une des étapes a été réalisée partiellement. Effectivement, les étapes 1, 2, 3, 4, 8, 9 ont été réalisées, et l'étape 5 a été réalisée à moitié pour cause de problèmes techniques avec la seconde application (CLES).

L'étape 8 n'a pas été réalisée pour cause de la contrainte temporelle, bien que l'implémentation actuelle du site permettrait son ajout sans trop de difficultés. Pour ce qui est de l'étape 6, nous n'avons pas réussi à déterminer comment la mettre en application.

## 3 Choix techniques

### 3.1 Back end

Pour ce travail, il nous étant imposé d'utiliser *Django*<sup>1</sup> pour le côté *back end*. Bien que ce soit imposé, il est intéressant de remarquer que *Django* est utilisé pour le *back end* d'un certain nombre de sites populaires<sup>2</sup> tels que *YouTube*, *Instagram*, et *Spotify*. Cependant, nous avons décidé d'utiliser le *framework REST*<sup>3</sup> de *Django* afin d'implémenter un serveur API. La motivation de ce choix et de n'utiliser le *back end* que pour transmettre les données au *front end*, et pour des tâches telles que vérifier si un utilisateur est connecté.

### 3.2 Front end

Au niveau du développement du *front-end*, nous avons décidé d'utiliser *reactjs* car celui-ci possède plusieurs avantages. Tout d'abord, ce *framework* possède une grande documentation. Ensuite, celui-ci facilite le développement car on peut avoir le rendu tout en codant en même temps. De plus, cet outil est assez intuitif. Le code est assez simple à prendre en main. Une autre raison d'utiliser ce *framework* est le fait qu'il existe beaucoup de projets utilisant *reactjs* et *Django* ensemble, il est donc simple d'avoir des exemples pour en tirer de l'inspiration.

---

<sup>1</sup><https://www.djangoproject.com/>

<sup>2</sup><https://djangostars.com/blog/10-popular-sites-made-on-django/>

<sup>3</sup><https://www.django-rest-framework.org/>

## 4 Front end

### 4.1 Hiérarchie des fichiers

Sur la figure 1 est illustrée la hiérarchie des fichiers. Le dossier source contient un dossier par page. Ces dossiers contiennent généralement un fichier index.js et un fichier .css. Le fichier index.js contient du code react et du HTML permettant de définir la structure de la page. Le fichier css permet définir le style de la page. Le dossier NavBar est un peu différent car celui-ci utilise la librairie styled-components pour faire le style à place de css. Le fichier App.js permet de définir la structure de l'application web en incluant toutes les pages.

```
frontend/
├ src/
|   ├ Components/
|   |   ├ AboutPage/
|   |   |   ├ index.js
|   |   |   ├ AboutPage.css
|   |   ├ CleoPage/
|   |   |   ├ index.js
|   |   |   ├ CleoPage.css
|   |   ├ HomePage/
|   |   |   ├ index.js
|   |   |   ├ HomePage.css
|   |   ├ ContactPage/
|   |   |   ├ index.js
|   |   |   ├ ContactPage.css
|   |   ├ LoginPage/
|   |   |   ├ index.js
|   |   |   ├ LoginPage.css
|   |   ├ NavBar/
|   |   |   ├ index.js
|   |   |   ├ NavBarElements.js
|   |   ├ ProductsPage/
|   |   |   ├ index.js
|   |   |   ├ ProductPage.css
|   |   ├ RegisterPage/
|   |   |   ├ index.js
|   |   |   ├ RegisterPage.css
|   |   ├ ServicesApplicationsPage/
|   |   |   ├ index.js
|   |   |   ├ ServicesApplicationsPage.css
|   └ App.js
```

Figure 1: Hiérarchie des fichiers

### 4.2 Fonctionnement de React

Pour définir une page avec React, il faut définir la structure et le style. Par exemple, pour la page d'accueil, le code définissant la structure de celle-ci est illustrée dans

la figure 3. Le champ **const HomePage** permet de créer une variable contenant le code HTML de cette page. Cette variable sera utilisée par le fichier App.js. Celui-ci importe toute les pages tel illustrée dans la figure 2. Ce fichier lie les urls avec les différentes pages. Par exemple, le chemin **http://localhost:3000/Products** est lié à la variable **ProductsPage** qui contient la page.

```

1  import './App.css';
2  import React, { useState } from 'react';
3  import NavBar from './Components/NavBar';
4  import {BrowserRouter as Router, Routes, Route} from 'react-router-dom'
5  import HomePage from './Components/HomePage';
6  import ProductsPage from './Components/ProductsPage';
7  import ServicesApplicationsPage from './Components/ServicesApplicationsPage';
8  import LoginPage from './Components/LoginPage';
9  import RegisterPage from './Components/RegisterPage';
10 import { setAuthToken } from './Contexts/setAuthToken';
11 import CleoPage from './Components/CleoPage';
12 import ContactPage from './Components/ContactPage';
13 import AboutPage from './Components/AboutPage';
14 import ErrorPage from './Components/ErrorPage';
15 const token = localStorage.getItem("token");
16 if (token) {
17   setAuthToken(token);
18 }
19 function App() {
20   return (
21     <Router>
22       <NavBar/>
23       <Routes>
24         <Route path='/' element={<HomePage/>}/>
25         <Route path='/Products' element={<ProductsPage/>}/>
26         <Route path='/Contact/' element={<ContactPage/>}/>
27         <Route path='/ServicesApplications/' element={<ServicesApplicationsPage/>}/>
28         <Route path='/Login/' element={<LoginPage/>}/>
29         <Route path='/Register/' element={<RegisterPage/>}/>
30         <Route path='/About/' element={<AboutPage/>}/>
31         <Route path='/Error/' element={<ErrorPage/>}/>
32         <Route path='/ServicesApplications/cleo/' element={<CleoPage/>}/>
33       </Routes>
34     </Router>
35   );
36 }
37
38 export default App;
39

```

Figure 2: App.js

Dans ce code html, certaines balises possèdent des champs **className**. Ceux-ci permettent d'accéder à leurs balises à partir du fichier css. La figure 4 illustre le fichier permettant d'ajouter du style à la page d'accueil.

```

import React from 'react'
import './HomePageElements.css'
const HomePage = () => {
  return (
    <div className='homepage'>
      <div className='hometextfield'>
        <p>
          Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
        </p>
      </div>
    </div>
  );
}

export default HomePage;

```

Figure 3: Structure de la page d'accueil

```

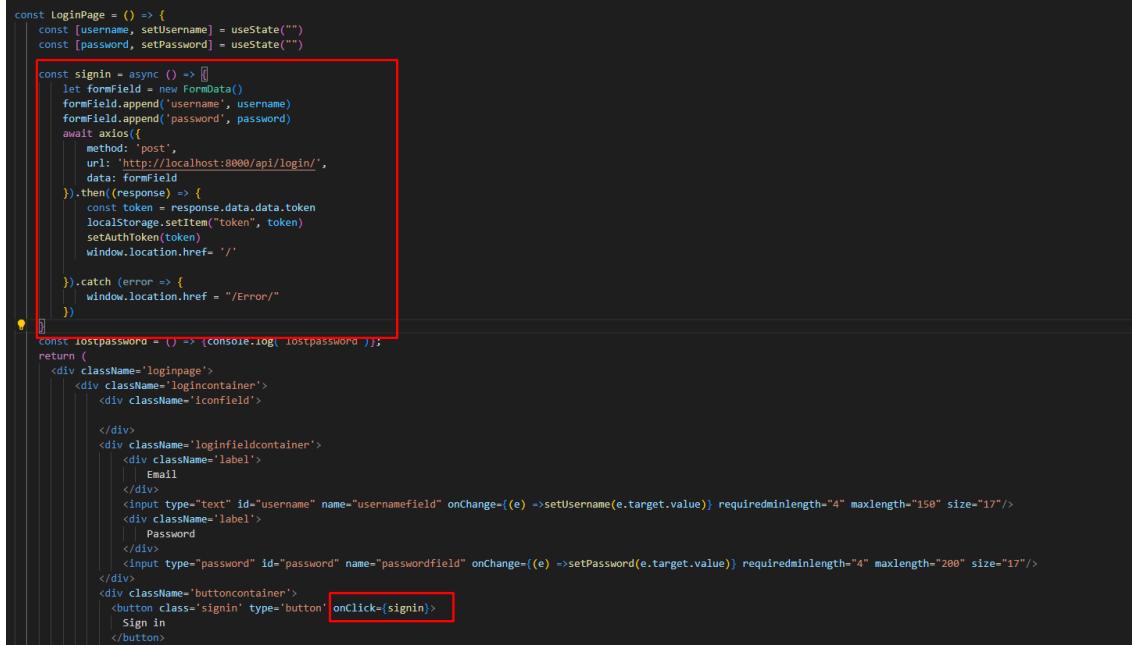
1 .homepage {
2   background:url('../Assets/Computer-background.jpg') no-repeat center center fixed;
3   height: 100vh;
4   width: 100%;
5   background-position: center;
6   background-repeat: no-repeat;
7   background-size: cover;
8   object-fit: cover;
9 }
10 .hometextfield {
11   position: absolute;
12   margin: auto;
13   top:50%;
14   left:50%;
15   transform: translate(-50%, -50%);
16   padding: 1% 3% 1%;
17   font-family: Arial, sans-serif;
18   color: #fff;
19   background: rgba(0, 0, 0, .25);
20   font-weight: bold;
21 }
22
23
24 html, body{
25   margin:0;
26   padding:0;
27 }
28

```

Figure 4: Style de la page d'accueil

Pour l'authentification et la création de compte, il est nécessaire d'envoyer une requête HTML de type 'POST' à Django. Pour ce faire, nous utilisons la librairie Axios de React. Sur la figure 5 est illustrée la méthode qui envoie une requête permettant de se connecter à son compte lorsque l'utilisateur appuie sur le bouton **signin**. D'abord une variable **FormData** est instanciée dans laquelle nous venons ajouter le nom de l'utilisateur ainsi que le mot de passe. Ensuite, nous envoyons la requête à l'aide de la fonction **axios()**. Celle-ci prend le type de méthode, l'url et les données comme paramètres. Si la requête se passe bien, nous recevons une

réponse contenant le token de l'utilisateur. Celui-ci est stocké dans la mémoire du navigateur, ensuite nous re-dirigeons l'utilisateur vers la page d'accueil. Sinon, nous re-dirigeons l'utilisateur vers une page d'erreur.



```

const LoginPage = () => {
  const [username, setUsername] = useState("")
  const [password, setPassword] = useState("")

  const signin = async () => [
    let formfield = new FormData()
    formfield.append('username', username)
    formfield.append('password', password)
    await axios({
      method: 'post',
      url: 'http://localhost:8000/api/login/',
      data: formfield
    }).then((response) => {
      const token = response.data.data.token
      localStorage.setItem("token", token)
      setAuthToken(token)
      window.location.href= '/'
    }).catch ((error) => {
      window.location.href = "/Error/"
    })
  ]

  const lostpassword = () => {console.log(lostpassword)}
  return (
    <div className="loginpage">
      <div className="logincontainer">
        <div className="iconfield">

          </div>
          <div className="loginfieldcontainer">
            <div className="label">
              | Email
            </div>
            <input type="text" id="username" name="usernamefield" onChange={(e) =>setUsername(e.target.value)} requiredminlength="4" maxlength="150" size="17"/>
            <div className="label">
              | Password
            </div>
            <input type="password" id="password" name="passwordfield" onChange={(e) =>setPassword(e.target.value)} requiredminlength="4" maxlength="200" size="17"/>
          </div>
          <div className="buttoncontainer">
            <button className="signin" type="button" onClick={signin}>
              Sign in
            </button>
          </div>
        </div>
      </div>
    </div>
  )
}

```

Figure 5: Requête à partir de React

Afin de naviguer entre les différents onglets, nous avons implémenté une barre de navigation. Pour ce faire, nous utilisons une balise **NavLink** qui se base sur la balise **Link** de la librairie **react-router-dom**. Sur la figure 6 est illustrée l'importation la balise **Link**. Ensuite, dans la figure 7, nous pouvons voir la création de la barre de navigation contenant les différents onglets. Les onglets sont créés à l'aide de la balise **NavLink** importée depuis le fichier **NavBarElement.js**.

```

1 import styled from 'styled-components'
2 import { NavLink as Link } from 'react-router-dom'
3
4
5 export const Nav = styled.nav` 
6   position:fixed;
7   background: #144a61;
8   height: 60px;
9   width: 100%;
10  display: flex;
11  padding: 0rem calc((100vw - 700px) / 2);
12  z-index: 10;
13
14 export const NavLink = styled(Link)` 
15  color: #fff;
16  display: flex;
17  align-items: center;
18  text-decoration: none;
19  padding: 0 1rem;
20  height: 100%;
21  cursor: pointer;
22  &.active {
23    background: #1d6888;
24  }
25

```

Figure 6: Fichier NavBarElement.js

```

function hasJWT() {
  let flag = false;

  //check user has JWT token
  localStorage.getItem("token") ? flag=true : flag=false

  return flag
}
return (
  <>
    <Nav>
      <NavLink to="/" activeStyle>
        Home
      </NavLink>
      <NavLink to="/About" activeStyle>
        About us
      </NavLink>
      <NavLink to="/Products" activeStyle>
        Products
      </NavLink>
      <NavLink to="/ServicesApplications" activeStyle>
        Services & Applications
      </NavLink>
      <NavLink to="/Contact" activeStyle>
        Contact
      </NavLink>
      {!hasJWT() ?
        <NavLink to="/Login" activeStyle>
          Login
        </NavLink>
        : null}
      {!hasJWT() ?
        <NavLink to="/Register" activeStyle>
          Register
        </NavLink>
        : null}
      {hasJWT() ?
        <button class='logout' type='button' onClick={logout}>
          Logout
        </button>
        : null}
    </Nav>
  </>
);

```

Figure 7: Fichier index.js pour le Navbar

La fonction **hasJWT()** permet de déterminer si l'utilisateur est connecté ou pas

à l'aide du token. Si celui-ci est connecté, nous n'affichons plus l'onglet **Register**, l'onglet **Login** et nous affichons le bouton **Log out**. Sinon, nous gardons les onglets **Register** et **Login**. Lorsque l'utilisateur se déconnecte de sa session, la fonction **logout()**(voir figure 8) est lancée. Celle-ci demande à Django de retirer son token de la base de donnée à l'aide de la fonction **axios**.

```
const logout = () => {
  console.log('loggingoff')
  const token = window.localStorage.getItem("token")
  axios.post("http://127.0.0.1:8000/api/logout/", {headers: {"Authorization": "Token ${token}"}});
  window.localStorage.removeItem("token")
  window.location.href="/Login"
};
```

Figure 8: Fonction permettant la déconnexion

Pour télécharger un fichier depuis le serveur, il suffit d'effectuer une requête axios en indiquant l'url ciblée et le token si nécessaire. Ensuite, le serveur répond avec le fichier désiré. React télécharge le fichier en utilisant la fonction **Download** de la librairie **downloadjs**. Un exemple de cette requête est illustrée dans la figure 9

```
const downloadcleo = () => {
  const token = window.localStorage.getItem("token")
  axios.post("http://127.0.0.1:8000/api/download/cleo/", formData, {
    responseType: 'arraybuffer',
    headers: {"Authorization": "Token " + token},
  })
  .then(response => {
    const content = response.headers['content-type'];
    download(response.data, "CLEO_V5.zip", content);
    console.log(response)
  }).catch (error => {
    window.location.href = "/Error/"
  });
};
```

Figure 9: Requête de téléchargement de fichier

### 4.3 Illustration des pages

Dans cette section, les différentes pages sont illustrées afin d'avoir une visualisation du site.

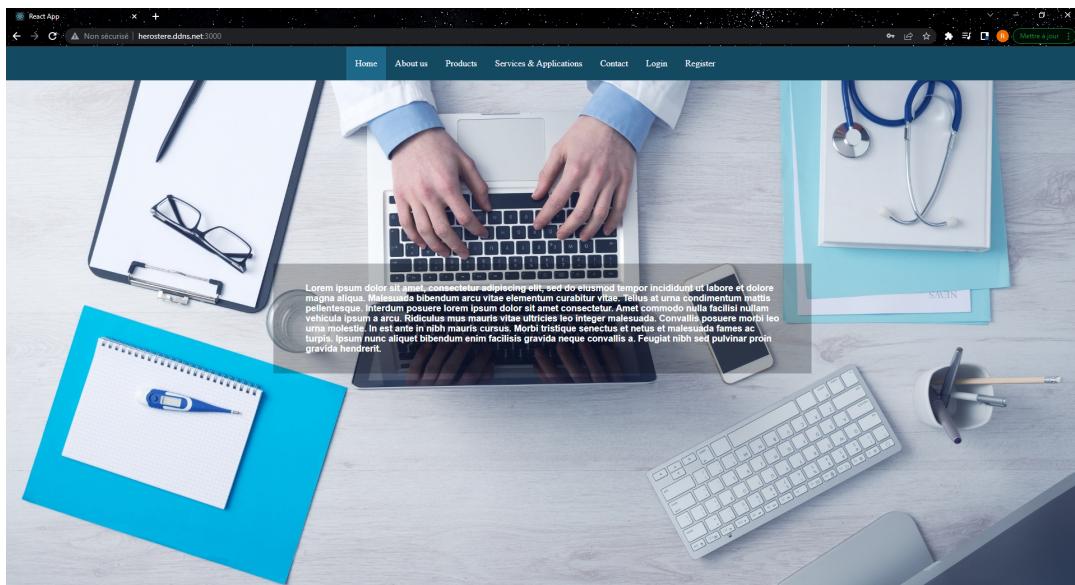


Figure 10: Home

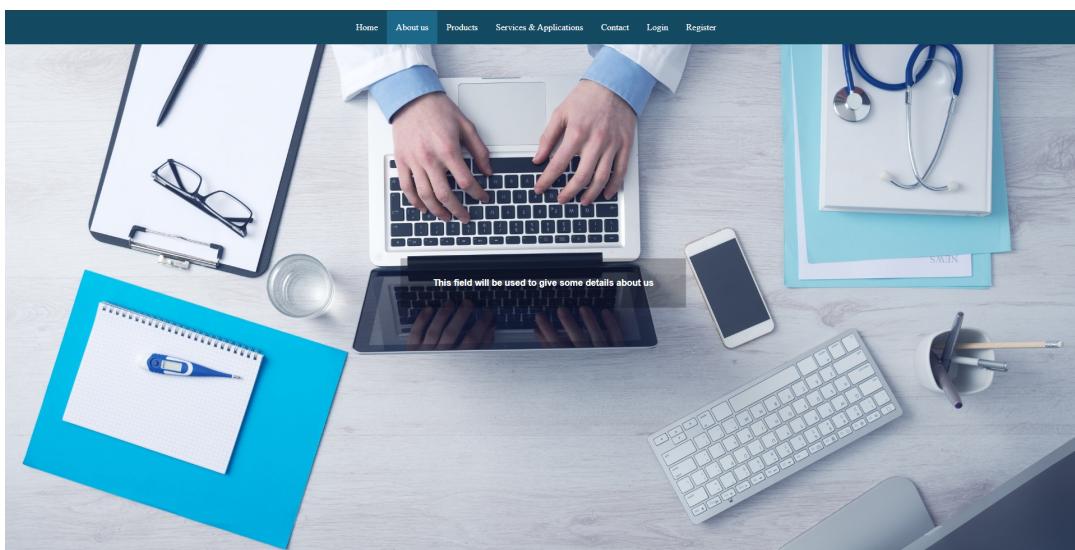


Figure 11: About us

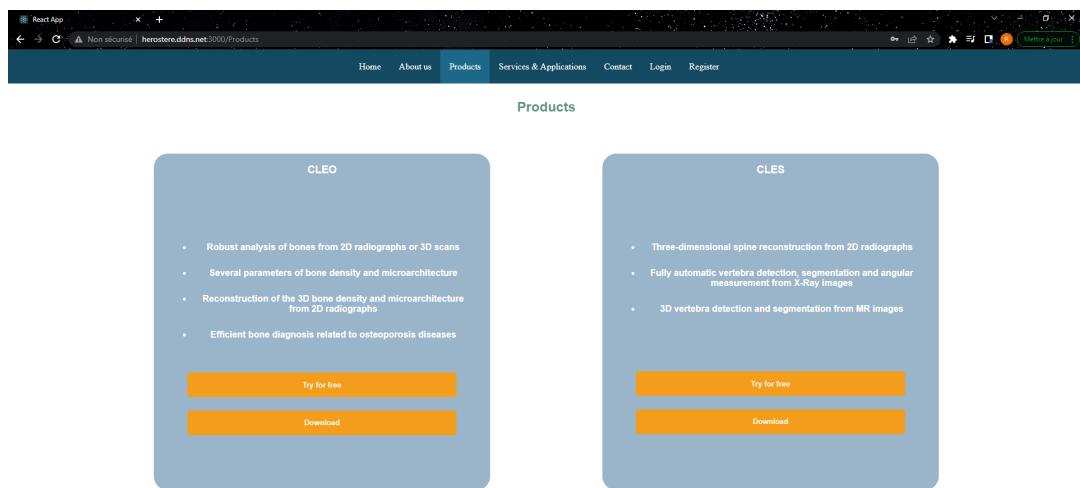


Figure 12: Products

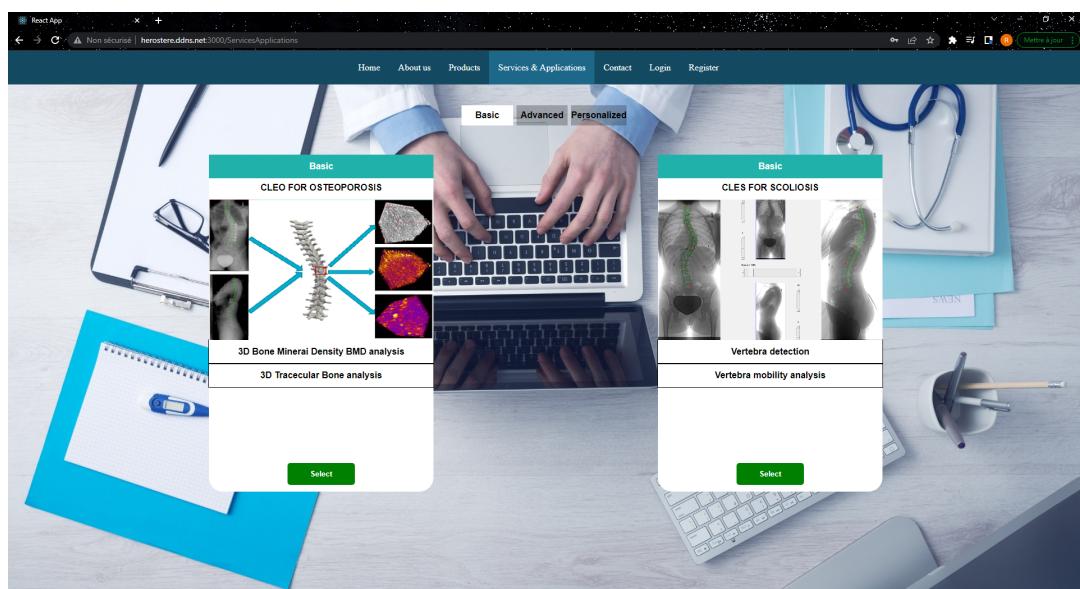


Figure 13: Services &amp; Applications Basic

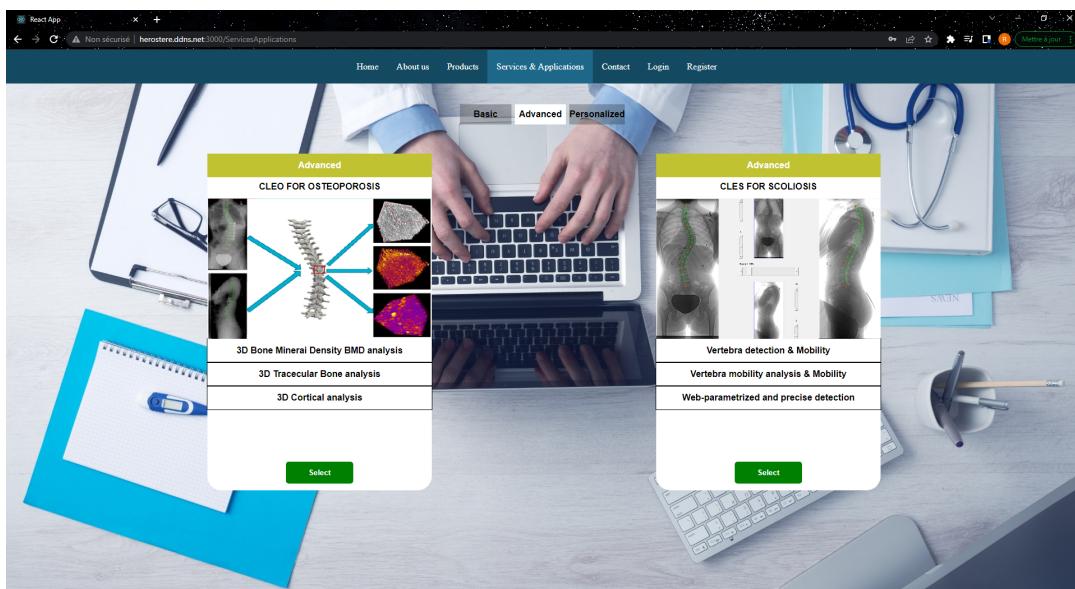


Figure 14: Services &amp; Applications Advanced

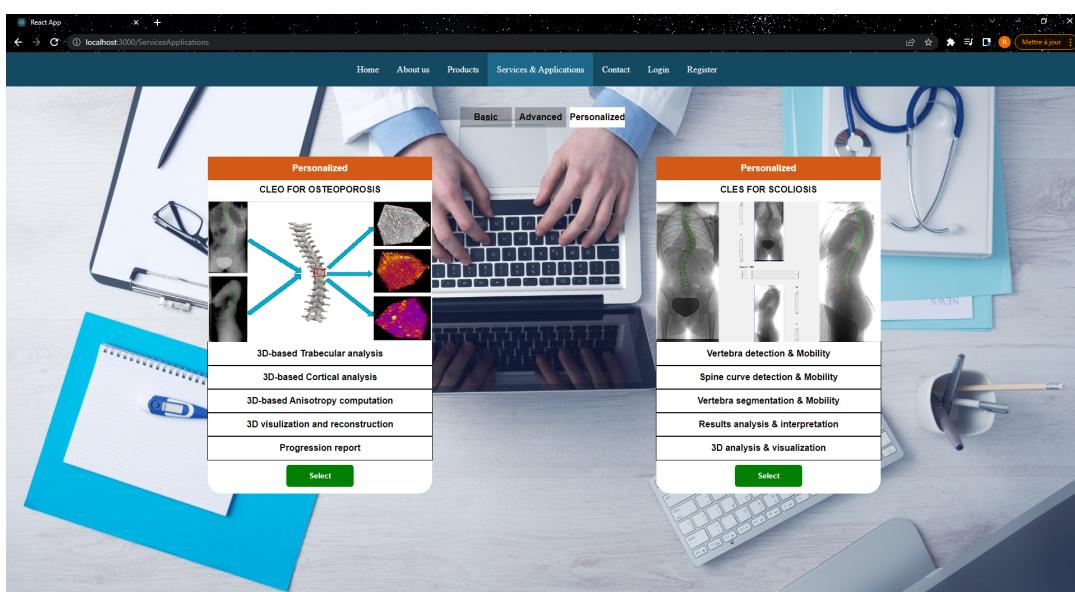


Figure 15: Services &amp; Applications Personalized

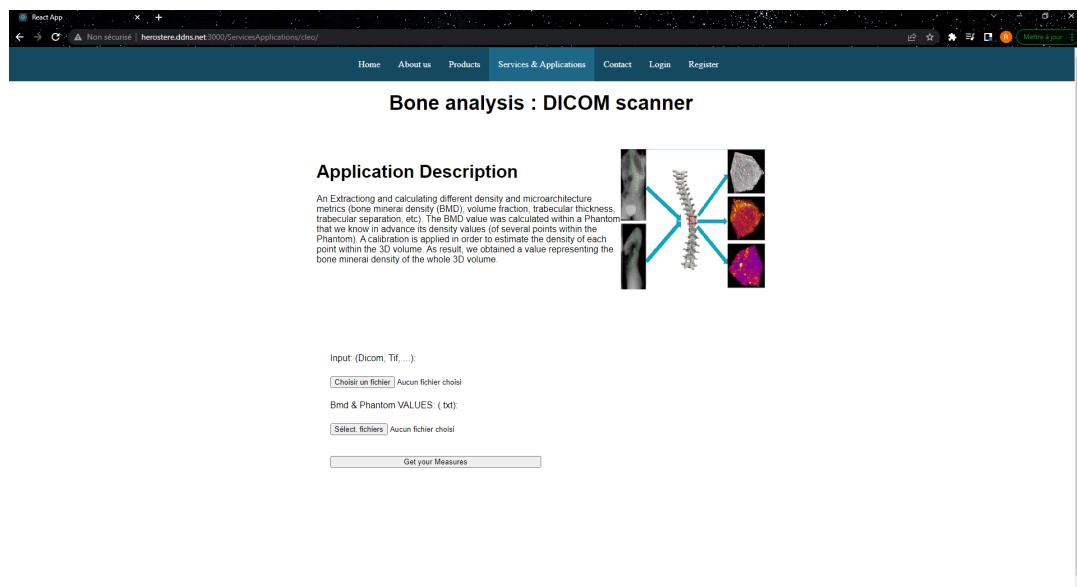


Figure 16: Cleo

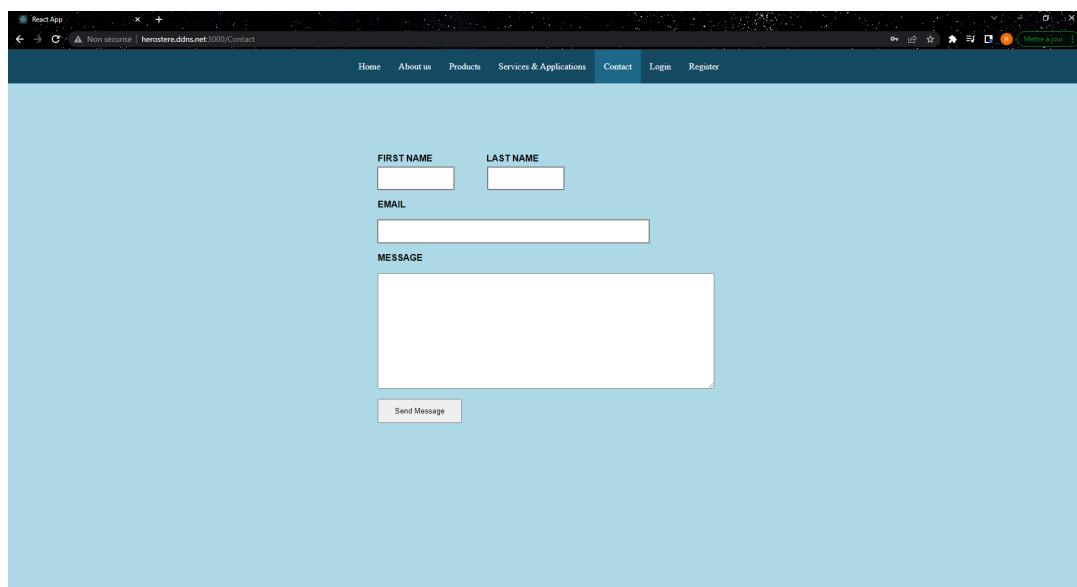


Figure 17: Contact

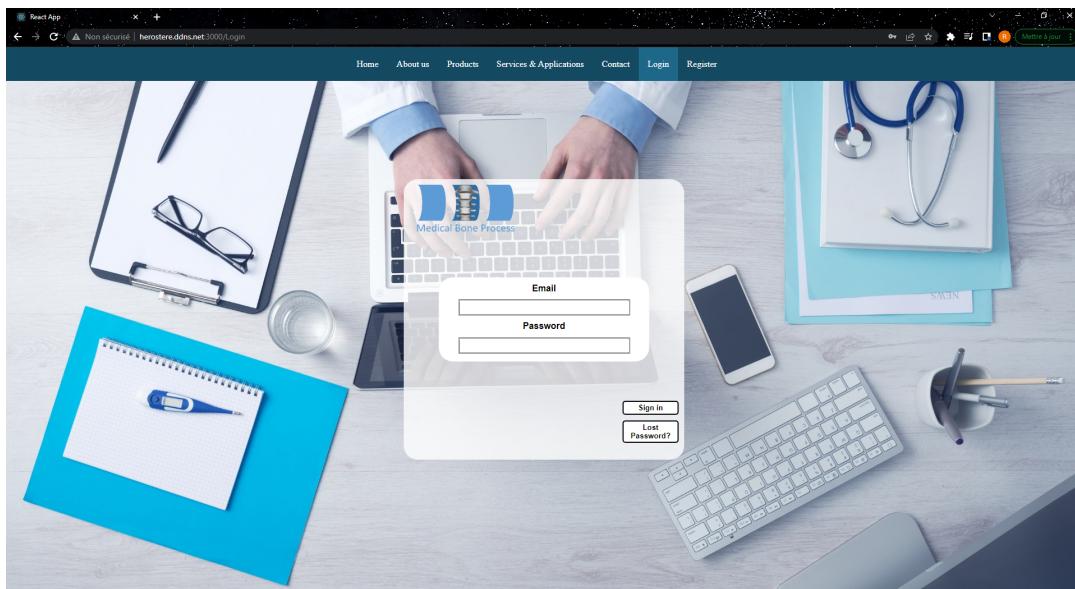


Figure 18: Login

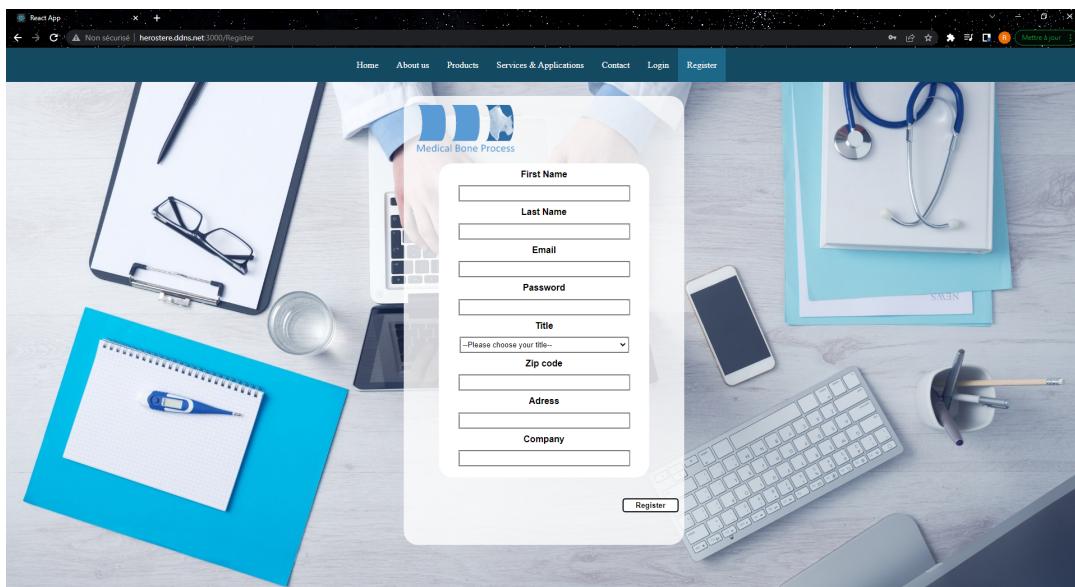


Figure 19: Register

## 5 Back end

Le *back end* de l'application est développé avec *Django* en utilisant le *framework REST*. Ce *framework* permet d'utiliser *Django* comme une *Web API*. De ce fait, nous avons créé un certain nombre de *endpoints* permettant à la partie *front-end* du projet de communiquer avec la partie *back end*.

Les différents *endpoints* utilisés sont listés et décris ci-dessous :

- `api/create/`  
Permet à un utilisateur de créer un nouveau compte.
- `api/login/`  
Permet à un utilisateur de se connecter.
- `api/logout/`  
Permet à un utilisateur de se déconnecter.
- `api/cleo/`  
Permet d'exécuter l'application CLEO.
- `api/download/cleo/`  
Permet de télécharger l'application CLEO.

### 5.1 Model

Une partie important du *back end* est le modèle utilisé dans la base de données. Afin de pouvoir personnaliser les champs attribués à un utilisateur, il a été nécessaire de créer une modèle de base de données pour les utilisateurs. Ce modèle comporte les champs suivants :

- `id` : un nombre unique attribué à chaque utilisateur permettant de les identifier.
- `first_name` : le prénom de l'utilisateur.
- `last_name` : le nom de famille de l'utilisateur.
- `email` : l'adresse email de l'utilisateur. Elle sera utilisée pour que l'utilisateur puisse se connecter.
- `is_staff` : indique que l'utilisateur fait partie de l'équipe de développement.

- `is_active` : permet d'indiquer si le compte est actif. Si le compte est inactif, l'utilisateur ne pourra plus se connecter.
- `money` : représente le nombre de crédit que possède un utilisateur. Par défaut la valeur assignée est de 10. Un utilisateur ne peut pas choisir sa valeur de "base". Les 10 premiers crédits sont gratuits, mais les suivants seront payants. Les crédits seront utilisés pour vérifier si un utilisateur peut utiliser une application.
- `cleo_license` : indique si un utilisateur possède la licence pour télécharger l'application CLEO.
- `title` : représente le sexe de la personne (homme/femme). Ce champ est facultatif.
- `zip_code` : le code postal de l'utilisateur.
- `address` : l'adresse physique de l'utilisateur.
- `company` : la société dans laquelle travaille l'utilisateur. Ce champ est facultatif.

De plus, la fonction `get_cleo_license` implémentée dans le modèle pour pouvoir facilement vérifier si un utilisateur possède la licence pour télécharger l'application CLEO.

## 5.2 Serializer

Une autre partie importante du *back end* est le *serializer*. On retrouve dans la documentation<sup>4</sup> que le *serializer* permet de convertir des types de données complexes, e.g. les *querysets* et les instances de modèles, en des types de données natifs de Python qui peuvent être affiché sous forme de JSON, XML ou autre.

Il a donc été nécessaire d'implémenter un *serializer* pour les utilisateurs. Celui-ci a permis de valider si l'adresse mail indiquée par un utilisateur est véritablement une adresse mail, vérifier si le mot de passe répond aux critères de sécurité imposés par le serveur, e.g. avoir un minimum de huit caractères, et indiquer les valeurs de base pour le nombre de crédits et pour la possession de la licence de CLEO. Cette façon de faire devrait permettre d'éviter qu'un utilisateur informé puisse envoyer une autre valeur que 10 pour le nombre de crédits lorsque la requête pour la création de son compte est envoyée au serveur.

---

<sup>4</sup><https://www.djangoproject.org/api-guide/serializers/>

## 5.3 Views

Enfin, nous retrouvons les différentes vues utilisées par l'API. Une description du fonctionnement de chacune des ces vues est donnée dans cette sous-section.

### 5.3.1 UserCreate

Cette vue est utilisée pour la création d'un compte d'utilisateur. Lorsqu'un utilisateur appuie sur le bouton “*register*” lors de la création de son compte, une requête *POST* est envoyée au *endpoint* chargé de la création de compte de l'API. Si la requête est correcte, et que tous les champs sont conforme, l'utilisateur est ajouté à la base de données et peut maintenant se connecter.

Les mots de passes sont “hashés” avant d'être sauvegardés dans la base de donnée. De cette façon, une fuite de données ne pourra pas les compromettre. Cependant, une amélioration est requise, car les mots de passes sont envoyés en clair au serveur. L'instauration d'un certificat SSL pourrait résoudre ce problème.

### 5.3.2 UserLogin

Cette vue permet de vérifier si les *credentials* fournis par l'utilisateur sont correcte. Pour se faire, lorsque l'utilisateur appuie sur le bouton “*login*”, une requête *POST* est envoyée au serveur. Cette requête contient le mot de passe ainsi que l'adresse mail de l'utilisateur. Ces données sont vérifiées et, si elles sont correctes, un *token* de connexion est envoyé à l'utilisateur par le biais de la réponse. Ce *token* sera utilisé pour vérifier qu'un utilisateur est bien connecté lors de différentes interactions.

Les *tokens*, comme les mots de passe, sont “hashés” avant d'être sauvegardés. Toutes fois, le même problème que celui des mots de passe est présent, car les *tokens* sont envoyés en clair au serveur. L'instauration d'un certificat SSL pourrait résoudre ce problème.

### 5.3.3 UserLogout

Cette vue permet de déconnecter un utilisateur. Pour se faire, une requête *POST* contenant le *token* de connexion de l'utilisateur dans son *header* est envoyée au serveur. Si ce *token* est valide, il est supprimé de la base de données et l'utilisateur est déconnecté du site.

### 5.3.4 DownloadCleo

Pour pouvoir vérifier si un utilisateur à le droit de télécharger l'application CLEO, et pour pouvoir la lui envoyer le cas échéant, il a été nécessaire d'implémenter une

vue. Cette vue reçoit le *token* de connexion de l'utilisateur dans le *header* d'une requête *POST*. Si le *token* est valide, le serveur vérifie si l'utilisateur associé au *token* possède la licence de l'application CLEO. Si c'est le cas, l'API renvoie une réponse contenant une archive *zip* qui contient le fichier *jar* de l'application. Sinon, une réponse indiquant une erreur est renvoyée.

Nous avons rencontré une certaine difficulté à envoyer le fichier *jar* "tel quel". C'est la raison pour laquelle nous avons décidé d'envoyer le fichier dans une archive *zip*.

#### 5.3.5 RunCleo

Enfin, cette vue permet à un utilisateur d'exécuter l'application CLEO avec des fichiers fournis en entrée. Pour se faire, comme pour les vues précédentes, le *token* de connexion de l'utilisateur est envoyé dans le *header* d'une requête *POST*, et les fichiers sont fournis dans le *body* de cette requête. Si les fichiers reçus correspondent bien aux fichiers attendus, un dossier est créé sur le serveur pour y sauvegarder les données fournies par l'utilisateur, et l'application CLEO est lancée. Une fois les résultats obtenus, ils sont placés dans le dossier de l'utilisateur, compressés dans une archive *zip* (de façon à pouvoir renvoyer les deux fichiers obtenus en sorties de l'application d'une seule fois) et l'archive est renvoyée à l'utilisateur.

Le même problème que rencontré dans la Section 5.3.2 est présent pour les *tokens*. De plus, nous avons imaginé qu'il serait intéressant de créer des sous-dossiers avec un nom unique, e.g. un numéro qui s'incrémente, pour garder un historique des utilisations faites par l'utilisateur. De cette façon, il serait possible de lister l'historique des utilisation sur le site web. Nos essais pour mettre en plus une telle approche n'ont pas aboutis à cause de difficultés techniques.

### 5.4 Mise en ligne

Nous avons rendu le site accessible en ligne via l'adresse `http://herostere.ddns.net:3000`. Depuis cette adresse, il est possible de parcourir tout le site. De plus, l'interface administrateur est aussi accessible via l'adresse `http://herostere.ddns.net:3000/admin/`.

## 6 Manuel d'installation pour utilisation locale

Pour pouvoir tester le site en version locale, il est nécessaire d'installer le dépendances Python contenues dans le fichier *requirements*. Aussi, il faut modifier certains fichiers de la partie *front end*, à savoir :

- `frontend/src/CleoPage/index.js`

- frontend/src/NavBar/index.js
- frontend/src/LoginPage/index.js
- frontend/src/ProductsPage/index.js
- frontend/src/RegisterPage/index.js

Dans ces fichiers, il faut modifier toutes références à l'adresse `http://herostere.ddns.net:8000/...` par `http://localhost:8000/...`.

Finalement, il faudra installer les modules importés dans les différents fichiers “*index.js*”.

Ensuite, le serveur web peut être démarré depuis le dossier “*Web-Technologies-Project\frontend*” avec la commande : `npm start`  
et le serveur de l'API peut être démarré avec la commande : `python Web-Technologies-Project\backend\manage.py runserver localhost:8000`

## 7 Principales difficultés

Lors de l'implémentation de ce travail, nous avons rencontré trois principales difficultés. La première fut de ne pas pouvoir consulter l'ancienne version du site autant que nous le souhaitions pour en essayer toutes les fonctionnalités en profondeur. Nous avons eu l'opportunité d'essayer le site sur une machine ne nous appartenant pas, mais il aurait été nécessaire que nous puissions le consulter à tout moment, pour vérifier le fonctionnement de certaines options.

Ensuite, nous avons eu des difficultés à mettre en oeuvre la communication entre le *front end* et le *back end* lors de l'authentification et le téléchargement de fichier. Cela représentait un certain défi technique, compte tenu que nous étions complètement novice dans la matière. Finalement, nous sommes parvenus à atteindre notre objectif, et le *front end* fonctionne correctement avec le *back end*.

Finalement, nous avons vainement tenté d'intégrer l'application CLES à notre site. L'application se trouvait dans un environnement *Docker* que nous n'avons pas réussi à démarrer, malgré de nombreux essaies et tentatives d'ajustement.

## 8 Conclusion

Ce projet fut l'opportunité de se rendre compte des mécanismes de développement *front end* et *back end*. Effectivement, nous nous sommes répartis le travail de façon à ce qu'une personne ne se préoccupe principalement que d'une des deux “facettes”

de l'implémentation. Il était intéressant de faire fonctionner ces deux mécanismes ensemble. De plus, nous avons appris de nouvelles technologies, à savoir *ReactJS* et le *framework REST* de *Django*.

Certaines améliorations restent cependant à apporter. Nous pensons notamment à l'utilisation d'un certificat SSL pour sécuriser les connexions entre le serveur et le client, ainsi qu'à l'utilisation des crédits par l'utilisateur.

Finalement, il serait intéressant d'utiliser les mécanismes mis en place pour l'utilisation de l'application CLEO pour intégrer l'application CLES. Effectivement, tous les ingrédients pour intégrer cette application sont à notre disposition, à condition de réussir à faire fonctionner l'application CLES.