Tutoriel : Django-API, Redux-ToolKit Query et React Native

Ce tutoriel illustre les principales interactions entre une base de données et une application.

La base de données est une base SQLite créée et reliée à l'application via le framework Django REST. L'application est écrite avec React Native.

La mise en place de l'API est fortement inspirée (pour ne pas dire copiée-collée) de l'excellent livre **Django for API** de William S. Vincent. Ouvrage que je recommande vivement !

Pré-requis

Les langages informatiques utilisés sont **python** et **React** (JavaScript). Il me semble nécessaire d'avoir quelques bases de programmation avec ces deux langages pour suivre correctement ce tutoriel.

Afin de reproduire les exemples, il est impératif d'installer **python** et **npm** (ou **yarn**) et de pouvoir les utiliser en ligne de commande :

```
$ python --help

$ npm --help
```

API et base de données

C'est parti ! Commençons par créer une petite API et une base de données grâce au framework Django REST.

Mise en place de l'environnement

La préparation de l'environnement se fait comme suit :

- création d'un dossier
- mise en place d'un environnement virtuel
- installation de django
- création d'un projet **django_project**
- démarrage du serveur

```
$ mkdir testAPI

$ cd testAPI

$ virtualenv env

$ source env/bin/activate
```

```
$ pip install django

$ django-admin startproject tutorial_project .

$ python manage.py runserver
```

N'appliquons pas immédiatement les mises à jour car nous allons configurer notre propre base de données utilisateurs.

Configuration de la table des utilisateurs

Créons une petite application chargé de configurer notre profil utilisateur :

```
$ python manage.py startapp accounts
```

Bien maintenant, nous ajoutons cette nouvelle application dans le fichier `tutorial_project/settings.py` :

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # Local
    "accounts.apps.AccountsConfig",  # new
]
```

Il est temps de créer un modèle d'utilisateur. Modifons le fichier `accounts/models.py` :

```
# accounts/models.py
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    name = models.CharField(null=True, blank=True, max_length=100)
    age = models.IntegerField(null=True, blank=True)
```

Ensuite Mettons à jour `AUTH_USER_MODEL` dans le fichier `settings.py`. De base, `AUTH_USER_MODEL` pointe vers `auth.User` mais nous voulons qu'il soit égal à `accounts.CustomUser` :

```
# django_project/settings.py
AUTH_USER_MODEL  = "accounts.CustomUser"  # new
```

Maintenant, nous pouvons appliquer les migrations :

```
$ python manage.py makemigrations

$ python manage.py migrate

$ python manage.py createsuperuser # admin admin123
```

Super ! Une base de données SQLite a été créée avec plusieurs tables dont la table *accounts_customuser*.

L'une des force de Django est son interface d'administration des données (disponible à l'adresse http://127.0.0.1:8000/admin/). Pour le moment, l'administration de la base de données *customuser* n'est pas disponible. Pour y rémédier, ajoutons un formulaire dans le nouveau fichier `accounts/forms.py` :

```
# accounts/forms.py
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from .models import CustomUser

class CustomUserCreationForm(UserCreationForm):
    class Meta(UserCreationForm):
        model = CustomUser
        fields = UserCreationForm.Meta.fields + ("name", "age")

class CustomUserChangeForm(UserChangeForm):
    class Meta:
        model = CustomUser
        fields = UserChangeForm.Meta.fields
```

Il ne reste plus qu'à rajouter la table *customuser* dans la page d'administration en éditant le fichier `accounts/admin.py`:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from .forms import CustomUserCreationForm, CustomUserChangeForm
from .models import CustomUser

class CustomUserAdmin(UserAdmin):
    add_form = CustomUserCreationForm
    form = CustomUserChangeForm
    model = CustomUser
    list_display = [
        "email",
        "username",
        "name",
        "age",
        "is_staff",
```

```
    ]
    fieldsets = UserAdmin.fieldsets + ((None, {"fields": ("name",
"age")}),)
    add_fieldsets = UserAdmin.add_fieldsets + ((None, {"fields": ("name",
"age")}),)

admin.site.register(CustomUser, CustomUserAdmin)
```

CQFD ! La table *customuser* est maintenant affichée dans notre page d'administration Django.

Configuration de la table "books"

Il est temps de créer une application spécifique aux livres (*books*) :

```
$ python manage.py startapp books
```

et on l'ajoute dans le fichier `settings.py` :

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # Local
    "accounts.apps.AccountsConfig",
    "books.apps.BooksConfig",  # new
]
```

Ajoutons une nouvelle table `Book` dans notre base de données qui aura comme champ :

- title : une chaine de caractère
- author : l'auteur du poste qui est lié à la table `accounts.CustomUser`
- created_at : une date qui sera par défaut la date de création
- updated_at : une date qui sera par défaut la date de modification

Le fichier `books/models.py` devient :

```
# books/models.py
from django.conf import settings
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=50)
    author = models.ForeignKey(settings.AUTH_USER_MODEL,
```

```
  on_delete=models.CASCADE)
      created_at = models.DateTimeField(auto_now_add=True)
      updated_at = models.DateTimeField(auto_now=True)

      def __str__(self):
          return self.title
```

`author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)` implique que les livres seront liés aux utilisateurs grâce au champ `author`.

Appliquons les migrations :

```
$ python manage.py makemigrations

$ python manage.py migrate
```

On ajoute les libres dans l'administration de notre api (`books/admin.py`):

```
from django.contrib import admin
from .models import Book

admin.site.register(Book)
```

http://127.0.0.1:8000/admin/ affiche désormais les livres : CQFD ! Construisons quelques instances de "Book" pour jouer via l'API du navigateur (http://127.0.0.1:8000/admin/books/book/add/)!

Django REST Framework

Pour le moment, nous n'avons utilisé que Django. Il est temps de faire intervernir Django REST Framework afin de faciliter :

- le routage via **`urls.py`**
- la transformation des données en JSON via les **`serializers.py`**
- la logique des endpoints via les **`views.py`**

Installons Django REST Framework :

```
$ pip install djangorestframework
```

et ajoutons-le à la liste des applications (`settings.py`):

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
```

```
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # 3rd-party apps
    "rest_framework",
    # Local
    "accounts.apps.AccountsConfig",
    "books.apps.BooksConfig",
]

REST_FRAMEWORK = {  # new
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.AllowAny",
    ],
}
```

URLs

Commençons par mettre à jour le fichier `django_project/urls.py` qui contient la structure des urls du projet. Nous lui indiquons que les urls commençant par `api/v1` pointent vers les livres :

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include # new

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/v1/", include("books.urls")),  # new
]
```

Une bonne pratique est de créer des versions de notre API. Maintenant, créons un fichier `books/urls.py` pour y classer les URLs relatives aux livres et utilisateurs :

```
# books/urls.py
from django.urls import path
from rest_framework.routers import SimpleRouter

from .views import BookViewSet, UserViewSet

router = SimpleRouter()
router.register("users", UserViewSet, basename="users")
router.register("books", BookViewSet, basename="books")

urlpatterns = router.urls
```

Pour le moment, notre API ne fonctionne pas car les vues et serializers n'ont pas encore été écrits.

Serializers

Non seulement, les serializers transforment les données en JSON mais ils offrent aussi la possibilité d'inclure ou d'exclure certains champs. Inscrivons toutes ces informations dans `books/serializers.py` :

```
# books/serializers.py
from django.contrib.auth import get_user_model # new
from rest_framework import serializers
from .models import Book


class BookSerializer(serializers.ModelSerializer):
    class Meta:
        fields = (
            "id",
            "author",
            "title",
            "created_at",
        )
        model = Book

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = get_user_model()
        fields = ("id", "username",)
```

Il nous reste à écrire les vues.

Views

Les vues définissent le comportement des endpoints. Il existe un nombre non négligeable de classes de vues différentes. Pour faciliter et accélérer l'écriture de l'API, nous utiliserons les **Viewsets**. Les Viewsets implémentent automatiquement les opérations de lecture, modification, création et suppression de données sans avoir besoin de les écrire manuellement.

Le fichier `books/views.py` devient :

```
# posts/views.py
from django.contrib.auth import get_user_model
from rest_framework import viewsets

from .models import Book
from .serializers import BookSerializer, UserSerializer

class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

class UserViewSet(viewsets.ModelViewSet):
```

```
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer
```

Super ! testons si tout fonctionne en nous rendant à l'adresse http://127.0.0.1:8000/api/v1/users/ pour afficher la liste des utilisateurs et http://127.0.0.1:8000/api/v1/books/ pour afficher la liste des livres 😃

CORS

La dernière chose à faire avant de développer l'application React Native est de mettre en place les autorisations pour les requêtes extérieures. Installons le package `django-cors-headers` :

```
$ pip install django-cors-headers
```

Modifions `django_project/settings.py` :

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # 3rd-party apps
    "rest_framework",
    "corsheaders",  # new
    # Local
    "accounts.apps.AccountsConfig",
    "books.apps.BooksConfig",
]

REST_FRAMEWORK = {
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.AllowAny",
    ],
}

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "corsheaders.middleware.CorsMiddleware",  # new
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]

# new
ALLOWED_HOSTS = ['*']
```

Cette fois-ci, testons notre API avec curl :

```
$ curl -X GET http://127.0.0.1:8000/api/v1/books/
```

Excellent pour le moment, arrêtons-nous la pour le développement de l'API. Il est temps de passer à React-Native.

```
$ cd ..

$ deactivate
```

React Native et Redux-ToolKit

React Native est un framework basée sur React permettant de créer des applications IOS et Android. **Redux-ToolKit** est une librairie JavaScript de gestion de l'état d'une application web.

React Native sera utilisé pour créer l'interface de l'application tandis que Redux-ToolKit (et plus particulièrement, Redux-ToolKit Query) servira à effectuer des requêtes avec Django REST Framework et mettre les données reçues en cache.

Installation via Expo

Expo est une plate-forme facilitant la création d'application IOS, Android et web. Commençons donc par créer un nouveau projet React Native avec Expo et téléchargeons la librairie Redux-ToolKit:

```
$ npx create-expo-app testReactNative

$ cd testReactNative

$ npm install @reduxjs/toolkit
```

Installation de React Native Debugger

Avant de lancer l'application, nous allons télécharger React Native Debugger (RND). RND est un debugger complet prenant en compte React Native mais aussi Redux. Le guide d'installation est disponible ici (https://github.com/jhen0409/react-native-debugger/blob/master/docs/getting-started.md) et ici (https://github.com/jhen0409/react-native-debugger)

Sous Mac, la commande est `brew install --cask react-native-debugger`. L'application est directement installée dans le dossier **Applications**.

Android Studio

Ajouter un texte d'explications pour l'installation d'Android Studio

Ouverture d'Android > "Device Manager" > start a device

XCode

Pour visualiser l'application sur un appareil Apple (iphone, par exemple), il est nécessaire d'avoir un Mac avec XCode installé et prêt à l'emploi.

Ouverture de l'application

Lançons l'application avec la commande `npm start` et choisissons **Android** (si c'est le cas).

Sur notre faux téléphone ouvert par Android Studio, l'écran affiche : "Open up App.js to start working on your app!". L'application est bien connectée au faux téléphone.

Ensuite, autorisons le "Debug Remote JS" dans le "Toggle Menu" (le menu s'affiche en tapand "m" en ligne de commande). Le debugger devrait s'ouvrir dans le navigateur avec comme URL http://localhost:19000/debugger-ui/. Le port est 19000. Fermons la page et lançons la commande :

```
open "rndebugger://set-debugger-loc?host=localhost&port=19000"
```

RND s'ouvre et affiche l'état de notre projet dans la console 😃 Si cela ne fonctionne pas directement, il faut éventuellement arrêter et relancer dans le "Debug Remote JS" dans le "Toggle Menu".

Connexion avec l'API : requête GET

Les requêtes GET récupèrent des données de l'API pour les afficher à l'écran. Avant de commencer à coder, modifons un peu l'architecture des dossiers.

Au sein du dossier **testReactNative**, créons deux dossiers : **src/features** et **src/reducers** :

```
$ mkdir src

$ mkdir src/features

$ mkdir src/reducers
```

Dans le dossier **features**, ajoutons deux sous-dossiers :

```
$ mkdir src/features/api

$ mkdir src/features/book
```

L'architecture basique du projet ressemble donc à :

- /testAPI
    - manage.py
    - /tutorial_project
    - /books
    - /accounts
- /testReactNative
    - App.js
    - /src
        - /features
            - /api
            - /book
        - /reducers

Le dossier `src/features/api` contient les fichiers avec le code Redux. Créons un nouveau fichier `src/features/api/bookSlice.js` et éditons-le :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'http://10.0.2.2:8000/api/v1/' }),
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
    }),
  }),
})

export const { useGetListOfBooksQuery } = bookApi
```

`createApi` définit un ensemble de endpoints décrivant comment récupérer des données à partir d'un backend :

- `reducerPath` le nom de l'endroit où sera monté nos données dans le store (dans notre cas, `bookApi`)
- `baseQuery` est l'URL de base de chaque requête (i.e. `'http://10.0.2.2:8000/api/v1/'`)
- `endpoints` est une liste d'`endpoints`. Dans notre exemple, `books/` sera ajoutée à l'URL de base pour envoyer des requêtes à `'http://10.0.2.2:8000/api/v1/books/'`.

Redux-ToolKit Query construit automatiquement des hooks utiles à l'utilisation des requêtes. Ici, la récupération des données se fera grâce à `useGetListOfBooksQuery` et sera utilisé dans le nouveau fichier `src/features/book/BookList.js` :

```
import React from 'react';
import { Text, View, Image, StyleSheet } from 'react-native';
import { useGetListOfBooksQuery } from '../api/bookSlice'

export const BookList = () => {

    const { data, isLoading, isSuccess, isError, error } =
useGetListOfBooksQuery()

    let content

    if (isLoading) {
        content = <Text> Loading </Text>
    } else if (isSuccess) {
        content = <Text> Query works ! </Text>
    } else if (isError) {
        content = <Text> Query doesn't work !</Text>
    }

    return (
        <View>
            { content }
        </View>
    )
}
```

Pour le moment, l'application affiche :

- **Loading** pendant le téléchargement des données
- **Query works !** si la requête GET est un succès
- **Query doesn't work !** si la requête échoue

Seulement pour effectuer des requêtes, nous devons éditer le *store* de Redux avec le *reducer* créé dans `src/features/api/bookSlice.js`. Editons donc le fichier `src/reducers/store.js` :

```
import { configureStore } from '@reduxjs/toolkit';
import { bookApi } from '../features/api/bookSlice';

export const store = configureStore({
  reducer: {
    [bookApi.reducerPath]: bookApi.reducer
  },
  middleware: getDefaultMiddleware =>
    getDefaultMiddleware().concat(bookApi.middleware)
});
```

Excellent ! Enfin, importons notre composant dans le fichier `App.js` et enveloppons notre application avec le store :

```
import { StatusBar } from 'expo-status-bar';
import { StyleSheet, View, Text } from 'react-native';
import { BookList } from "./src/features/book/BookList"
import { Provider } from 'react-redux';
import { store } from './src/reducers/store';

export default function App() {
  return (
    <Provider store={store}>
      <View style={styles.container}>
        <BookList />
        <StatusBar style="auto" />
      </View>
    </Provider>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

(En cas d'erreur, il est possible que la solution soit d'installer à nouveau react-redux `npm install --save react-redux`).

Super ! **Query works !** s'affiche sur l'écran du téléphone émulé : la requête GET est un succès !

Connexion avec l'API : requête POST

Les requêtes POST envoient des données au serveur afin de modifier la base de données. Nous allons créer un petit bouton qui enverra des requêtes POST à chaque fois que le bouton sera pressé. Les données envoyées seront les mêmes à chaque requête. Plus tard, nous ajouterons un formulaire afin de customiser les données envoyées au serveur.

Modifons le fichier `src/features/api/bookSlice.js` :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'http://10.0.2.2:8000/api/v1/' }),
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
    }),
    addNewBook: builder.mutation({
```

```
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
        body: initialBook
      })
    })
  }),
})

export const { useGetListOfBooksQuery, useAddNewBookMutation } = bookApi
```

Le nouveau hook `useAddNewBookMutation` est ensuite utilisé dans un nouveau fichier `src/features/book/BookPost.js` :

```
import React, { useState } from 'react';
import { Text, View, Button, StyleSheet } from 'react-native';
import { useAddNewBookMutation } from '../api/bookSlice'

const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 22,
        marginTop: 30,
    },
    item: {
        padding: 10,
        fontSize: 18,
        height: 44,
        textAlign: 'center',
    },
});

export const BookPost = () => {

    const [title, setTitle] = useState('Le Machine learning avec Python
!')
    const [authorId, setAuthorId] = useState(1)
    const [addNewBook, { isLoading }] = useAddNewBookMutation()

    const canSave = [authorId, title].every(Boolean) && !isLoading

    const onSaveBookClicked = async () => {
        if (canSave) {
            try {
                await addNewBook({ title, author: authorId }).unwrap()
                setTitle('')
                setAuthorId('')
            } catch (err) {
                console.error('Failed to save the post: ', err)
            }
        }
```

```
    }

    return (
        <View style={styles.container}>
            <Button
                onPress={onSaveBookClicked}
                title="Make a POST Request"
                color="#6495ed"
                accessibilityLabel="Make a POST Request by clicking this
button"
            />
        </View>
    )
}
```

Ce composant servira à afficher un bouton qui enverra le titre d'un nouveau livre (ici, "Le Machine learning avec Python !").

Enfin ajoutons le nouveau bouton dans `App.js`:

```
import { StatusBar } from 'expo-status-bar';
import { StyleSheet, View, Text } from 'react-native';
import { BookList } from "./src/features/book/BookList"
import { BookPost } from './src/features/book/BookPost';

import { Provider } from 'react-redux';
import { store } from './src/reducers/store';

export default function App() {
  return (
    <Provider store={store}>
      <View style={styles.container}>
        <BookList />
        <BookPost />
        <StatusBar style="auto" />
      </View>
    </Provider>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Et voilà 😃 A chaque fois que le bouton sera pressé, une nouvelle requête POST sera envoyée avec comme auteur "1" qui est l'identifiant de l'administrateur et comme titre "Le Machine learning avec Python !". La liste est toujours disponible à l'adresse http://127.0.0.1:8000/api/v1/books/.

Mettre à jour les données en cache

Améliorons `src/features/book/BookList.js` afin d'afficher la liste des livres :

```
import React from 'react';
import { Text, View, Image, StyleSheet, FlatList } from 'react-native';
import { useGetListOfBooksQuery } from '../api/bookSlice'

const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 22,
        marginTop: 30,
    },
    item: {
        padding: 10,
        fontSize: 18,
        height: 44,
        textAlign: 'center',
    },
});

export const BookList = () => {

    const { data, isLoading, isSuccess, isError, error } =
useGetListOfBooksQuery()

    let content

    if (isLoading) {
        content = <Text> Loading ... </Text>
    } else if (isSuccess) {
        content = <FlatList data={data} renderItem={({ item }) => <Text
style={styles.item}>Titre {item.id} : {item.title}</Text>} />
    } else if (isError) {
        content = <Text> Query doesn't work !</Text>
    }

    return (
        <View style={styles.container}>
            {content}
        </View>
    )
}
```

La liste des livres s'affichent désormais sur l'écran. Par-contre, elle ne se met pas à jour à chaque fois que le bouton de la requête POST est pressé. Mettons en place cette amélioration en ajoutant `tagTypes:`

`['Book']` à `src/features/api/bookSlice.js` :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'http://10.0.2.2:8000/api/v1/' }),
  tagTypes: ['Book'], //new
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
      providesTags: ['Book'] //new
    }),
    addNewBook: builder.mutation({
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
        body: initialBook
      }),
      invalidatesTags: ['Book'] //new
    })
  })
})

export const { useGetListOfBooksQuery, useAddNewBookMutation } = bookApi
```

Les **tags** sont très utiles pour synchroniser la base de données avec l'application. Maintenant dès qu'un livre est ajouté à la base de données, la liste des livres affichée par l'application est automatiquement mise à jour.

Connexion avec l'API : requête DELETE

Pour le moment, nous avons intégré des requêtes GET et POST. La requête suivant est DELETE. Le slice Redux devient :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'http://10.0.2.2:8000/api/v1/' }),
  tagTypes: ['Book'],
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
      providesTags: ['Book']
    }),
    addNewBook: builder.mutation({
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
```

```
        body: initialBook
      }),
      invalidatesTags: ['Book']
    }),
    deleteBook: builder.mutation({
      query: (id) => ({
        url: `/books/${id}/`,
        method: 'DELETE',
      }),
      invalidatesTags: ['Book'],
    }),
  })
})

export const { useGetListOfBooksQuery, useAddNewBookMutation,
useDeleteBookMutation } = bookApi
```

Le hook `useDeleteBookMutation` sert à supprimer un livre de la base de données et le tag `invalidatesTags: ['Book']` met à nouveau à jour la liste des livres en cas de suppression d'une entrée.

Ajoutons le bouton de suppression en-dessous de chaque titre de livre :

```
import React from 'react';
import { Text, View, Button, StyleSheet, FlatList } from 'react-native';
import { useGetListOfBooksQuery, useDeleteBookMutation } from
'../api/bookSlice'

const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 22,
        marginTop: 30,
    },
    item: {
        padding: 10,
        marginTop: 30,
        fontSize: 18,
        height: 44,
        textAlign: 'center',
    },
});

export const BookList = () => {

    const { data, isLoading, isSuccess, isError, error } =
useGetListOfBooksQuery()
    const [deleteBook, response] = useDeleteBookMutation()

    let content
```

```
    if (isLoading) {
        content = <Text> Loading ... </Text>
    } else if (isSuccess) {
        content = <FlatList data={data} renderItem={({ item }) => <View>
            <Text style={styles.item}>Titre {item.id} : {item.title}
</Text>
            <Button onPress={() => deleteBook(item.id)} title="Delete
Book" color="#6495ed"/> //new
        </View>} />
    } else if (isError) {
        content = <Text> Query doesn't work !</Text>
    }

    return (
        <View style={styles.container}>
            {content}
        </View>
    )
}
```

Excellent !

Création de formulaire : Formik

Pour le moment, la requête POST ajoute systématiquement le même livre dans la base de données (*Le Machine learning avec Python* avec comme auteur *admin*). Il serait intéressant de pouvoir rentrer un titre et le nom de l'auteur avant de soumettre la requête POST. La librairie Formik (https://formik.org/) est la librairie la plus utilisée pour la création de formulaire avec React Native. Téléchargeons-la :

```
$ npm install formik
```

En plus de formik, installons la librairie Yup qui servira à valider les données rentrées par l'utilisateur :

```
$ npm install yup
```

Bien, maintenant modifions le fichier `BookPost.js` :

```
import React, { useState } from 'react';
import { Text, View, Button, StyleSheet, TextInput, Label } from 'react-
native';
import { Formik } from 'formik';
import * as Yup from 'yup';

import { useAddNewBookMutation } from '../api/bookSlice'

const styles = StyleSheet.create({
    container: {
```

```
        flex: 1,
        padding: 10,
        marginTop: 20,
    },
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 12,
        textAlign: 'center',
        fontSize: 18,
    },
    inputLablel: {
        paddingTop: 10,
        fontSize: 18,
        height: 44,
        fontWeight: "bold",
    }
});


export const BookPost = () => {

    const [addNewBook, { isLoading }] = useAddNewBookMutation()

    const onSaveBookClicked = async (values) => {

        const canSave = [values.author, values.book].every(Boolean) &&
!isLoading

        if (canSave) {
            try {
                await addNewBook({ title: values.book, author:
values.author }).unwrap()
            } catch (err) {
                console.error('Failed to save the post: ', err)
            }
        }
    }

    const MyReactNativeForm = props => (
        <Formik
            initialValues={{
                book: "Le Machine learning avec Python !",
                author: 1
            }}
            onSubmit={values => onSaveBookClicked(values)}
            validationSchema={Yup.object({
                book: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required'),
                author: Yup
                    .number("Must be more than 0")
```

```
                    .integer("Must be more than 0")
                    .required('Required'),
            })}
        >
            {({ handleChange,
                handleBlur,
                handleSubmit,
                values,
                errors,
                touched,
                isValid, }) => (
                <View>
                    <Text style={styles.inputLablel}>Book :</Text>
                    <TextInput
                        name="book"
                        placeholder='Add a new book'
                        onChangeText={handleChange('book')}
                        onBlur={handleBlur('book')}
                        style={styles.inputStyle}
                    />
                    {touched.book && errors.book &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.book}</Text>
                    }
                    <Text style={styles.inputLablel}>Author :</Text>
                    <TextInput
                        name="author"
                        placeholder='1'
                        onChangeText={handleChange('author')}
                        onBlur={handleBlur('author')}
                        value={values.author}
                        style={styles.inputStyle}
                    />
                    {touched.author && errors.author &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.author}</Text>
                    }
                    <Button onPress={handleSubmit} title="Submit"
color="#6495ed" />
                </View>
            )}
        </Formik>
    );

    return (
        <View style={styles.container}>
            <MyReactNativeForm />
        </View>
    )
}
```

Nous avons défini "Le Machine learning avec Python !" et "1" pour valeurs initiales pour respectivement le titre et l'auteur du livre (`initialValues`). Ensuite, la librairie Yup nous a permis de définir les valeurs acceptées par le formulaire (`validationSchema`) : `title` est un `string` et `author` est un `integer`.

Un formulaire est créé et les erreurs de remplissage d'informations sont affichées (`{touched.book && errors.book && <Text style={{ fontSize: 16, color: '#FF0D10' }}>{errors.book} </Text>}`). Si les données rentrées sont valides, un nouveau livre sera ajouté à la base de données. Sinon le bouton n'effectuera aucune action.

Connexion avec l'API : requête PUT

Après la lecture, la suppression et l'ajout, il ne reste plus qu'à mettre en place la modification des données. La modification se fait grâce aux requêtes POST. Commençons par ajouter un nouvel endpoint dans le fichier `bookSlice.js` :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'http://10.0.2.2:8000/api/v1/' }),
  tagTypes: ['Book'],
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
      providesTags: ['Book']
    }),
    addNewBook: builder.mutation({
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
        body: initialBook
      }),
      invalidatesTags: ['Book']
    }),
    deleteBook: builder.mutation({
      query: (id) => ({
        url: `/books/${id}/`,
        method: 'DELETE',
      }),
      invalidatesTags: ['Book'],
    }),
    updateBook: builder.mutation({ // new
      query(data) {
        const { id, ...body } = data
        return {
          url: `books/${id}/`,
          method: 'PUT',
          body,
        }
      },
      invalidatesTags: ['Book'],
```

```
    }),
  })
})

export const { useGetListOfBooksQuery, useAddNewBookMutation,
useDeleteBookMutation, useUpdateBookMutation } = bookApi
```

Le hook `useUpdateBookMutation` servira à updater les données dans le fichier `BookList.js` :

```
import React from 'react';
import { Text, View, Button, StyleSheet, FlatList, TextInput } from
'react-native';
import { Formik } from 'formik';
import * as Yup from 'yup';

import { useGetListOfBooksQuery, useDeleteBookMutation,
useUpdateBookMutation } from '../api/bookSlice'

const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 22,
        marginTop: 30,
    },
    item: {
        padding: 10,
        marginTop: 30,
        fontSize: 18,
        height: 44,
        textAlign: 'center',
    },
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 6,
        marginTop: 12,
        textAlign: 'center',
        fontSize: 18,
    },
});

export const BookList = () => {

    const { data, isLoading, isSuccess, isError, error } =
useGetListOfBooksQuery()
    const [deleteBook, response] = useDeleteBookMutation()
    const [updateBook, { isLoading: isUpdating }] =
useUpdateBookMutation()

    const onUpdateBookClicked = async (modifiedValues, initialValues) => {
```

```
        const canSave = [initialValues.props.author, modifiedValues.book,
initialValues.props.id].every(Boolean)

        if (canSave) {
            try {
                await updateBook({ id: initialValues.props.id, title:
modifiedValues.book, author: initialValues.props.author }).unwrap()
            } catch (err) {
                console.error('Failed to save the post: ', err)
            }
        }
    }

    const UpdateForm = props => (
        <Formik
            initialValues={{
                book: props.title,
            }}
            onSubmit={values => onUpdateBookClicked(values, props)}
            validationSchema={Yup.object({
                book: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required'),
            })}
        >
            {({ handleChange,
                handleBlur,
                handleSubmit,
                values,
                errors,
                touched,
                isValid, }) => (
                <View>
                    <TextInput
                        name="book"
                        placeholder='Modify book title'
                        onChangeText={handleChange('book')}
                        onBlur={handleBlur('book')}
                        style={styles.inputStyle}
                    />
                    {touched.book && errors.book &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.book}</Text>
                    }
                    <Button onPress={handleSubmit} title="Modify Book"
color="#6495ed" />
                </View>
            )}
        </Formik>
    );

    let content
```

```
    if (isLoading) {
        content = <Text> Loading ... </Text>
    } else if (isSuccess) {
        content = <FlatList data={data} renderItem={({ item }) => <View>
            <Text style={styles.item}>Titre {item.id} : {item.title}
</Text>
            <Button onPress={() => deleteBook(item.id)} title="Delete
Book" color="#6495ed"/>
            <UpdateForm props={item}/>
        </View>} />
    } else if (isError) {
        content = <Text> Query doesn't work !</Text>
    }

    return (
        <View style={styles.container}>
            {content}
        </View>
    )

}
```

Bien 😃 Le petit formulaire Formik ne contenant qu'un champ "titre" permet de modifier le titre des livres dans la base de données ! Nous pourrions améliorer la gestion des erreurs mais nous verrons ça après l'authentification car cette partie nécessite d'être attentif aux réponses du serveur. Pour le moment, nous avons déjà une application capable d'effectuer des requêtes GET, POST, DELETE et PUT 😃

Navigation

La navigation sur une application est un peu différente de la navigation web. Les balises `link` ou `href` ne sont plus utilisées. A la place, nous allons utiliser la librairie **React Navigation**. Le très bon tutoriel officiel de la librairie est disponible à l'adresse https://reactnavigation.org/docs/getting-started. Nous allons nous en inspirer pour parfaire notre exemple.

Commençons par installer les dépendances nécessaires :

```
$ npx expo install react-native-screens react-native-safe-area-context

$ npm install @react-navigation/native-stack

$ npm install @react-navigation/bottom-tabs@^5.x
```

Ensuite, créons un dossier pour les écrans et deux écrans *HomeScreen.js* et *BookListScreen.js* :

```
$ mkdir src/screens

$ touch src/screens/HomeScreen.js
```

```
$ touch src/screens/BookListScreen.js
```

Le fichier `BookListScreen.js` contiendra le code actuel d'affichage et de modification des listes tandis que `HomeScreen.js` affichera un petit message d'accueil. Par la suite, nous ajouterons des écrans de connexion et authentification.

Ajoutons le code de `src/features/BookList.js` dans `BookListScreen.js` :

```
import { BookList } from '../features/book/BookList';
import { BookPost } from '../features/book/BookPost';
import { SafeAreaView } from 'react-native-safe-area-context';

const BookListScreen = ({ navigation }) => {
  return (
    <SafeAreaView style={{ flex: 1, alignItems: 'center', justifyContent:
'center' }}>
      <BookList />
      <BookPost />
    </SafeAreaView>
  )
}

export default BookListScreen;
```

Bien et ajoutons un petit message d'accueil dans `HomeScreen.js` :

```
import { Text, Button, StyleSheet } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';

const styles = StyleSheet.create({
    container: {
        flex: 1,
        alignItems: 'center',
        justifyContent: 'center',
        padding: 10,
    },
    textStyle: {
        fontSize: 36,
        fontWeight: "bold",
        marginBottom: 10
    }
});

const HomeScreen = ({ navigation }) => {
    return (
        <SafeAreaView style={styles.container}>
            <Text style={styles.textStyle}>Home Screen</Text>
            <Button
```

```
                title="Go to Book List"
                onPress={() => navigation.navigate('Books')}
                color="#6495ed"
            />
        </SafeAreaView>
    );
}

export default HomeScreen;
```

Enfin enveloppons l'application dans notre objet de navigation (`App.js`) :

```
import React from 'react';
import { View, Button, Text } from "react-native"
import { Provider } from 'react-redux';
import { store } from './src/reducers/store';
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-
stack';
import { SafeAreaProvider } from 'react-native-safe-area-context';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import BookListScreen from './src/screens/BookListScreen';
import HomeScreen from './src/screens/HomeScreen'
import Ionicons from '@expo/vector-icons/Ionicons'

const Tab = createBottomTabNavigator();

export default function App() {
  return (
    <Provider store={store}>
      <SafeAreaProvider>
        <NavigationContainer>
          <Tab.Navigator
            screenOptions={({ route }) => ({
              tabBarIcon: ({ focused, color, size }) => {
                let iconName;
                if (route.name === 'Home') {
                  iconName = focused
                    ? 'home'
                    : 'home-outline';
                } else if (route.name === 'Books') {
                  iconName = focused
                    ? 'book'
                    : 'book-outline';
                }
                return <Ionicons name={iconName} size={size} color={color}
/>;
              }
            })}
          >
            <Tab.Screen name="Home" component={HomeScreen} options={{
title: 'Home' }} />
```

```
            <Tab.Screen name="Books" component={BookListScreen} options={{
title: 'Books' }} />
          </Tab.Navigator>
        </NavigationContainer>
      </SafeAreaProvider>
    </Provider>
  );
}
```

Nous avons un menu de navigation en bas de notre écran 😉 C'est déjà pas mal ! Seulement en prévision de la mise en place de l'authentification, ajoutons deux écrans : 1) pour l'inscription d'un nouvel utilisateur (*Sign-In*) 2) pour la connexion d'un utilisateur existant (*Sign-Up*).

Commençons par créer `src/screens/SignInScreen.js` et `src/screens/SignUpScreen.js` ainsi qu'un écran de bienvenue `src/screens/WelcomeScreen.js` qui accueillera les visiteurs non-authentifiés.

```
$ touch src/screens/SignInScreen.js

$ touch src/screens/SignUpScreen.js

$ touch src/screens/WelcomeScreen.js
```

Le code de `SignInScreen.js` sera écrit dans `src/authentification/SignIn.js` et le code de `SignUpScreen.js` dans `src/authentification/SignUp.js` :

```
$ mkdir src/features/authentification

$ touch src/features/authentification/SignIn.js

$ touch src/features/authentification/SignUp.js

$ touch src/features/authentification/Welcome.js
```

Pour le moment, ajoutons simplement une troisième page à notre menu **Tab** qui pointera vers la page `Welcome.js`. Cette page permettra de choisir entre **SignIn** et **Sign-Up**. La page `SignIn.js` renverra vers la page `SignUp.js` qui elle-même renverra vers la page `Home`.

Bien commençons par éditer `src/features/authentification/SignIn.js` :

```
import React from 'react';
import { Text, View, Button, StyleSheet, TextInput } from 'react-native';
import { Formik } from 'formik';
import * as Yup from 'yup';
import { useNavigation } from '@react-navigation/native';
```

```
const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 10,
        marginTop: 20,
    },
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 12,
        textAlign: 'center',
        fontSize: 18,
    },
    inputLablel: {
        paddingTop: 10,
        fontSize: 18,
        height: 44,
        fontWeight: "bold",
    }
});


export const SignIn = () => {

    const navigation = useNavigation();

    const SignInForm = props => (
        <Formik
            initialValues={{
                username: "Pierre",
                email: "pierre@email.com",
                password: "password"
            }}
            onSubmit={() => navigation.navigate('Sign-Up')}
            validationSchema={Yup.object({
                username: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required'),
                email: Yup
                    .string()
                    .email("email is not valid")
                    .required('Required'),
                password: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required')
            })}
        >
            {({ handleChange,
                handleBlur,
                handleSubmit,
                values,
```

```
                errors,
                touched,
                isValid, }) => (
                <View>
                    <Text style={styles.inputLablel}>Username :</Text>
                    <TextInput
                        name="username"
                        placeholder='username'
                        onChangeText={handleChange('username')}
                        onBlur={handleBlur('username')}
                        value={values.username}
                        style={styles.inputStyle}
                    />
                    {touched.username && errors.username &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.username}</Text>
                    }
                    <Text style={styles.inputLablel}>Email :</Text>
                    <TextInput
                        name="email"
                        placeholder='email'
                        onChangeText={handleChange('email')}
                        onBlur={handleBlur('email')}
                        value={values.email}
                        style={styles.inputStyle}
                    />
                    {touched.email && errors.email &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.email}</Text>
                    }
                    <Text style={styles.inputLablel}>Password :</Text>
                    <TextInput
                        name="password"
                        placeholder='password'
                        onChangeText={handleChange('password')}
                        onBlur={handleBlur('password')}
                        value={values.password}
                        style={styles.inputStyle}
                    />
                    {touched.password && errors.password &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.password}</Text>
                    }
                    <View style={{ paddingTop: 20 }}>
                    <Button onPress={handleSubmit} title="Sign In"
color="#6495ed"/>
                    </View>
                </View>
            )}
        </Formik>
    );

    return (
        <View style={styles.container}>
```

```
            <SignInForm />
        </View>
    )
}
```

Ensuite `src/features/authentification/SignUp.js` :

```
import React from 'react';
import { Text, View, Button, StyleSheet, TextInput } from 'react-native';
import { Formik } from 'formik';
import * as Yup from 'yup';
import { useNavigation } from '@react-navigation/native';

const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 10,
        marginTop: 20,
    },
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 12,
        textAlign: 'center',
        fontSize: 18,
    },
    inputLablel: {
        paddingTop: 10,
        fontSize: 18,
        height: 44,
        fontWeight: "bold",
    }
});


export const SignUp = () => {

    const navigation = useNavigation();

    const SignUpForm = props => (
        <Formik
            initialValues={{
                username: "Pierre",
                email: "pierre@email.com",
                password: "password"
            }}
            onSubmit={() => navigation.navigate('Home')}
            validationSchema={Yup.object({
                username: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
```

```
                    .required('Required'),
                password: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required')
            })}
        >
            {({ handleChange,
                handleBlur,
                handleSubmit,
                values,
                errors,
                touched,
                isValid, }) => (
                <View>
                    <Text style={styles.inputLablel}>Username :</Text>
                    <TextInput
                        name="username"
                        placeholder='username'
                        onChangeText={handleChange('username')}
                        onBlur={handleBlur('username')}
                        value={values.username}
                        style={styles.inputStyle}
                    />
                    {touched.username && errors.username &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.username}</Text>
                    }
                    <Text style={styles.inputLablel}>Password :</Text>
                    <TextInput
                        name="password"
                        placeholder='password'
                        onChangeText={handleChange('password')}
                        onBlur={handleBlur('password')}
                        value={values.password}
                        style={styles.inputStyle}
                    />
                    {touched.password && errors.password &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.password}</Text>
                    }
                    <View style={{ paddingTop: 20 }}>
                    <Button onPress={handleSubmit} title="Sign Up"
color="#6495ed"/>
                    </View>
                </View>
            )}
        </Formik>
    );

    return (
        <View style={styles.container}>
            <SignUpForm />
        </View>
```

```
    )
}
```

Comme on peut le voir, une fois que l'on a fait un formulaire, les autres suivent très facilement. La page `src/features/authentification/SignUp.js` contient deux boutons qui renvoient vers le **SignIn** ou le **SignOut** :

```
import { Text, Button, StyleSheet, View } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';

const styles = StyleSheet.create({
    container: {
        flex: 1,
        alignItems: 'center',
        justifyContent: 'center',
        padding: 10,
    },
    textStyle: {
        fontSize: 36,
        fontWeight: "bold",
        marginBottom: 10
    }
});

const Welcome = ({ navigation }) => {
    return (
        <SafeAreaView style={styles.container}>
            <Text style={styles.textStyle}>Welcome Screen</Text>
            <View style={{ flexDirection: 'row' }}>
                <View style={{ padding: 5 }}>
                    <Button
                        title="Go to Sign-Up"
                        onPress={() => navigation.navigate('Sign-Up')}
                        color="#6495ed"
                    />
                </View>
                <View style={{ padding: 5 }}>
                    <Button
                        title="Go to Sign-In"
                        onPress={() => navigation.navigate('Sign-In')}
                        color="#6495ed"
                    />
                </View>
            </View>
        </SafeAreaView>
    );
}

export default Welcome;
```

Parfait ! Il ne reste plus qu'à importer ces trois functions dans leur écran respectif. Commençons par `src/screens/SignInScreen.js` :

```
import { SignIn } from '../features/authentification/SignIn';
import { SafeAreaView } from 'react-native-safe-area-context';

const SignInScreen = () => {
  return (
    <SafeAreaView style={{ flex: 1, alignItems: 'center', justifyContent:
'center' }}>
      <SignIn />
    </SafeAreaView>
  )
}

export default SignInScreen;
```

`src/screens/SignUpScreen.js` est très similaire :

```
import { SignUp } from '../features/authentification/SignUp';
import { SafeAreaView } from 'react-native-safe-area-context';

const SignUpScreen = () => {
  return (
    <SafeAreaView style={{ flex: 1, alignItems: 'center', justifyContent:
'center' }}>
      <SignUp />
    </SafeAreaView>
  )
}

export default SignUpScreen;
```

et enfin `src/screens/WelcomeScreen.js` :

```
import { Text, Button, StyleSheet } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';
import Welcome from '../features/authentification/Welcome'

const styles = StyleSheet.create({
    container: {
        flex: 1,
        alignItems: 'center',
        justifyContent: 'center',
        padding: 10,
    }
});
```

```
const WelcomeScreen = ({ navigation }) => {
    return (
        <SafeAreaView style={styles.container}>
            <Welcome navigation={navigation} />
        </SafeAreaView>
    );
}

export default WelcomeScreen;
```

Super 😃 Un peu long mais pas très compliqué à mettre en oeuvre. Finalement, il ne reste plus qu'à ajouter ces écrans dans un `Stack.Navigator`. Ce `Stack.Navigator` gérera la navigation entre les composants `Welcome.js`, `SignIn.js` et `SignUp.js` et sera à l'intérieur du navigateur principal **Tab**. Pour cela, créons une fonction `WelcomeNavigation` dans `App.js` :

```
import React from 'react';
import { Provider } from 'react-redux';
import { store } from './src/reducers/store';
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-
stack';
import { SafeAreaProvider } from 'react-native-safe-area-context';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import BookListScreen from './src/screens/BookListScreen';
import HomeScreen from './src/screens/HomeScreen'
import SignInScreen from './src/screens/SignInScreen';
import SignUpScreen from './src/screens/SignUpScreen';
import WelcomeScreen from './src/screens/WelcomeScreen'
import Ionicons from '@expo/vector-icons/Ionicons'

const Tab = createBottomTabNavigator();
const Stack = createNativeStackNavigator();

function WelcomeNavigation() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Welcome" component={WelcomeScreen} />
      <Stack.Screen name="Sign-In" component={SignInScreen} options={{
title: 'Sign In' }} />
      <Stack.Screen name="Sign-Up" component={SignUpScreen} options={{
title: 'Sign Up' }} />
    </Stack.Navigator>
  );
}

export default function App() {
  return (
    <Provider store={store}>
      <SafeAreaProvider>
        <NavigationContainer>
          <Tab.Navigator
```

```
            screenOptions={({ route }) => ({
              tabBarIcon: ({ focused, color, size }) => {
                let iconName;
                if (route.name === 'Home') {
                  iconName = focused
                    ? 'home'
                    : 'home-outline';
                } else if (route.name === 'Books') {
                  iconName = focused
                    ? 'book'
                    : 'book-outline';
                } else if (route.name === 'Welcome Nav') {
                  iconName = focused
                    ? 'log-in'
                    : 'log-in-outline';
                }
                return <Ionicons name={iconName} size={size} color={color}
/>;
              }
            })}
          >
            <Tab.Screen name="Welcome Nav" component={WelcomeNavigation}
options={{ title: 'Welcome', headerShown: false }} />
            <Tab.Screen name="Home" component={HomeScreen} options={{
title: 'Home' }} />
            <Tab.Screen name="Books" component={BookListScreen} options={{
title: 'Books' }} />
          </Tab.Navigator>
        </NavigationContainer>
      </SafeAreaProvider>
    </Provider>
  );
}
```

CQFD ! Nous avons maintenant deux navigateurs : un "stack" pour la navigation "Welcome" et un "tab" pour la navigation générale.

Modal

Avant de mettre en place l'authentification, modifions un tout petit peu la liste des livres afin de la rendre plus jolie. Plutôt que d'afficher les formulaires de modification et de création d'un livre, nous allons utiliser les boutons qui ouvriront des petites fenêtres pop-up (les *modals*) contenant les formulaires.

La documentation officielle de React Native sur les modals est disponible à https://reactnative.dev/docs/modal.

Bien commençons par créer un nouveau dossier `src/features/form` qui contiendra les formulaires. Il est plus propre d'écrire les formulaires dans des fichiers dédiés. `src/features/form/CreateForm.js` ressemble à :

```
import React from 'react';
import { Text, View, Button, StyleSheet, FlatList, TextInput } from
'react-native';
import { Formik } from 'formik';
import * as Yup from 'yup';

import { useAddNewBookMutation } from '../api/bookSlice'

const styles = StyleSheet.create({
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 12,
        textAlign: 'center',
        fontSize: 18,
    },
    inputLablel: {
        paddingTop: 10,
        fontSize: 18,
        height: 44,
        fontWeight: "bold",
    },
});

function CreateForm() {

    const [addNewBook, { isLoading }] = useAddNewBookMutation()

    const onSaveBookClicked = async (values) => {

        const canSave = [values.author, values.book].every(Boolean) &&
!isLoading

        if (canSave) {
            try {
                await addNewBook({ 'title': values.book, 'author':
values.author }).unwrap()
            } catch (err) {
                <Text style={{ fontSize: 16, color: '#FF0D10' }}>{err}
</Text>

            }
        }
    }

    return (
        <Formik
            initialValues={{
                book: "Le Machine learning avec Python !",
                author: 1
            }}
            onSubmit={values => onSaveBookClicked(values)}
```

```
            validationSchema={Yup.object({
                book: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required'),
                author: Yup
                    .number("Must be more than 0")
                    .integer("Must be more than 0")
                    .required('Required'),
            })}
        >
            {({ handleChange,
                handleBlur,
                handleSubmit,
                values,
                errors,
                touched,
                isValid, }) => (
                <View>
                    <Text style={styles.inputLablel}>Book :</Text>
                    <TextInput
                        name="book"
                        placeholder='Add a new book'
                        onChangeText={handleChange('book')}
                        onBlur={handleBlur('book')}
                        style={styles.inputStyle}
                    />
                    {touched.book && errors.book &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.book}</Text>
                    }
                    <Text style={styles.inputLablel}>Author :</Text>
                    <TextInput
                        name="author"
                        placeholder='1'
                        onChangeText={handleChange('author')}
                        onBlur={handleBlur('author')}
                        value={values.author}
                        style={styles.inputStyle}
                    />
                    {touched.author && errors.author &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.author}</Text>
                    }
                    <Button onPress={handleSubmit} title="Add Book"
color="#6495ed" />
                </View>
            )}
        </Formik>
    );
}

export default CreateForm;
```

`src/features/form/UpdateForm.js` contient :

```
import React from 'react';
import { Text, View, Button, StyleSheet, FlatList, TextInput } from
'react-native';
import { Formik } from 'formik';
import * as Yup from 'yup';

import { useUpdateBookMutation } from '../api/bookSlice'

const styles = StyleSheet.create({
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 12,
        marginTop: 12,
        textAlign: 'center',
        fontSize: 18,
    },
});

function UpdateForm( {props} ) {

    const [updateBook, { isLoading: isUpdating }] =
useUpdateBookMutation()

    const onUpdateBookClicked = async (modifiedValues, props) => {

        const canSave = [props.author, modifiedValues.book,
props.id].every(Boolean)

        if (canSave) {
            try {
                await updateBook({ id: props.id, title:
modifiedValues.book, author: props.author }).unwrap()
            } catch (err) {
                console.error('Failed to save the post: ', err)
            }
        }
    }

    return (
    <Formik
        initialValues={{
            book: props.title,
        }}
        onSubmit={values => onUpdateBookClicked(values, props)}
        validationSchema={Yup.object({
            book: Yup
                .string()
                .min(3, 'Must be 3 characters or less')
```

```
                .required('Required'),
        })}
    >
        {({ handleChange,
            handleBlur,
            handleSubmit,
            values,
            errors,
            touched,
            isValid, }) => (
            <View>
                <TextInput
                    name="book"
                    placeholder='Modify book title'
                    onChangeText={handleChange('book')}
                    onBlur={handleBlur('book')}
                    style={styles.inputStyle}
                    value={values.book}
                />
                {touched.book && errors.book &&
                    <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.book}</Text>
                }
                <Button onPress={handleSubmit} title="Modify Book"
color="#6495ed" />
            </View>
        )}
    </Formik>
    )
}

export default UpdateForm;
```

Bien ! Maintenant créons un nouveau dossier `src/features/modal` et ajoutant `src/features/modal/CreateModal.js` ainsi que `src/features/modal/UpdateModal.js`. `src/features/modal/CreateModal.js` ressemble à :

```
import React, { useState } from 'react';
import { View, Modal, StyleSheet, Pressable, Text } from 'react-native';

import CreateForm from '../form/CreateForm';

const styles = StyleSheet.create({
    containerModal: {
        flexDirection: 'row',
        position: 'absolute',
        alignItems: 'center',
        justifyContent: 'center',
        right: 15,
        bottom: 15,
    },
```

```
    modalView: {
        margin: 20,
        alignItems: 'center',
        backgroundColor: "white",
        borderRadius: 10,
        padding: 35,
        alignItems: "center",
        shadowColor: "#000",
        shadowOffset: {
            width: 0,
            height: 2
        },
        shadowOpacity: 0.25,
        shadowRadius: 4,
        elevation: 5
    },
    button: {
        borderRadius: 20,
        padding: 10,
        elevation: 10,
        backgroundColor: "#DE271F",
    },
    buttonClose: {
        backgroundColor: "#6495ed",
        marginTop: 22,
    },
    textStyle: {
        color: "white",
        fontWeight: "bold",
        textAlign: "center",
    }
});

const CreateModal = () => {
    const [modalVisible, setModalVisible] = useState(false);
    return (
        <View style={styles.containerModal}>
            <Modal
                animationType="slide"
                transparent={true}
                visible={modalVisible}
                onRequestClose={() => { setModalVisible(!modalVisible);
}}>
                <View>
                    <View style={styles.modalView}>
                        <CreateForm />
                        <Pressable
                            style={[styles.button, styles.buttonClose]}
                            onPress={() => setModalVisible(!modalVisible)}
                        >
                            <Text style={styles.textStyle}>Close</Text>
                        </Pressable>
                    </View>
                </View>
```

```
            </Modal>
            <Pressable
                style={styles.button}
                onPress={() => setModalVisible(true)}
            >
                <Text style={styles.textStyle}>Add book</Text>
            </Pressable>
        </View>
    );
};

export default CreateModal;
```

Un modal dépend généralement d'un état local (ici `modalVisible`) qui est égal à `true` quand le modal est affiché et `false` quand il est caché. Ainsi un modal peut être vu comme un bout de code tantôt caché tantôt affiché. `src/features/modal/UpdateModal.js` contient :

```
import React, { useState } from 'react';
import { View, Button, Modal, StyleSheet } from 'react-native';

import UpdateForm from '../form/UpdateForm'

const styles = StyleSheet.create({
    containerModal: {
        alignItems: 'center',
        justifyContent: 'center',
    },
    modalView: {
        margin: 20,
        alignItems: 'center',
        backgroundColor: "white",
        borderRadius: 10,
        padding: 35,
        alignItems: "center",
        shadowColor: "#000",
        shadowOffset: {
            width: 0,
            height: 2
        },
        shadowOpacity: 0.25,
        shadowRadius: 4,
        elevation: 5
    },
    button: {
        padding: 10,
        elevation: 10,
        backgroundColor: "#6495ed",
        borderRadius: 2
    },
    buttonClose: {
        backgroundColor: "#6495ed",
```

```
        marginTop: 22,
    },

});

const UpdateModal = ({ props }) => {

    const [modalVisible, setModalVisible] = useState(false);
    return (
        <View style={styles.containerModal}>
            <Modal
                animationType="slide"
                transparent={true}
                visible={modalVisible}
                onRequestClose={() => { setModalVisible(!modalVisible);
}}>
                <View>
                    <View style={styles.modalView}>
                        <UpdateForm props={props} />
                        <View style={{marginTop:10}}>
                            <Button
                                onPress={() =>
setModalVisible(!modalVisible)}
                                title='Close'
                                color="#6495ed"
                            />
                        </View>
                    </View>
                </View>
            </Modal>
            <Button
                onPress={() => setModalVisible(true)}
                title='Update book'
                color="#6495ed"
            />
        </View>
    );
};

export default UpdateModal;
```

Et voilà 😃 Ce modal-ci est déclenché par un bouton et non plus un pressable mais le processus est le même. Il ne reste plus qu'à modifier `BookPost.js` :

```
import { View } from 'react-native';
import CreateModal from '../modal/CreateModal';

const BookPost = () => {

    return (
        <View style={{ flex: 1 }}>
```

```
            <CreateModal />
        </View>
    )
}

export default BookPost;
```

et `BookList.js` :

```
import React, { useState } from 'react';
import { Text, View, Button, StyleSheet, FlatList } from 'react-native';

import { useGetListOfBooksQuery, useDeleteBookMutation } from
'../api/bookSlice'
import UpdateModal from '../modal/UpdateModal';

const styles = StyleSheet.create({
    item: {
        padding: 10,
        marginTop: 10,
        fontSize: 18,
        height: 44,
        textAlign: 'center',
    },
    listItem: {
        margin: 10,
        padding: 10,
        backgroundColor: "#FFF",
        width: "80%",
        flex: 1,
        alignSelf: "center",
        flexDirection: "column",
        borderRadius: 5
    },
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 6,
        marginTop: 12,
        textAlign: 'center',
        fontSize: 18,
    },
});

function RenderItemList({ item }) {

    const [deleteBook, response] = useDeleteBookMutation()

    return (
        <View style={styles.listItem}>
            <Text style={styles.item}>Titre {item.id} : {item.title}
```

```
</Text>
            <View style={{ flexDirection: 'row', justifyContent: 'space-
evenly'}}>
                <Button onPress={() => deleteBook(item.id)} title="Delete
Book" color="#6495ed" />
                <UpdateModal props={item} />
            </View>
        </View>
    )
}

function GetBookList() {

    const { data, isLoading, isSuccess, isError, error } =
useGetListOfBooksQuery()

    let content

    if (isLoading) {
        content = <Text> Loading ... </Text>
    } else if (isSuccess) {
        content = <FlatList data={data} renderItem={({ item }) =>
<RenderItemList item={item} />} />
    } else if (isError) {
        content = <Text> Query doesn't work !</Text>
    }

    return (
        <View >
            {content}
        </View>
    )
}

const BookList = () => {
    return (
        <View >
            <GetBookList />
        </View>
    )
}

export default BookList;
```

Comme nous avons modifié l'export des composants `BookPost` et `BookList`, nous devons changer leur importation dans `BookListScreen.js` :

```
import BookList from '../features/book/BookList';
import BookPost from '../features/book/BookPost';
import { SafeAreaView } from 'react-native-safe-area-context';
```

```
const BookListScreen = () => {
  return (
    <SafeAreaView style={{ flex: 1 }}>
      <BookList />
      <BookPost />
    </SafeAreaView>
  )
}

export default BookListScreen;
```

Génial ! Nos composants sont plus propres ainsi que notre application 😃

Django REST Framwork : Permissions et Authentification

L'application permet d'effectuer les quatre opérations CRUD principales : GET, POST, DELETE et PUT. A l'heure actuelle, tout le monde peut effectuer n'importe quelles requêtes sans s'authentifier. Cela pose deux problèmes de sécurité :

- Tous les utilisateurs ont accès à tous les endpoints : modifions **les permissions** pour n'autoriser l'accès qu'aux utilisateurs authentifiés.
- L'authentification est très basique : les librairies **dj-rest-auth** et **django-allauth** permettent un meilleur contrôle des authentifications et de l'enregistrement des utilisateurs.

Mais d'abord, modifions un petit peu la structure de l'API pour que les endpoints relatifs aux utilisateurs soient gérés dans l'application *accounts*.

Modifications de l'application *accounts*

Modification des vues

`accounts/views.py` :

```
from django.contrib.auth import get_user_model
from rest_framework import viewsets

from .serializers import UserSerializer

class UserViewSet(viewsets.ModelViewSet):
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer
```

et `books/views.py` :

```
# posts/views.py
from rest_framework import viewsets
```

```
from .models import Book
from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

Modification des serializers

```
$ touch accounts/serializers.py
```

accounts/serializers.py :

```
from django.contrib.auth import get_user_model # new
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = get_user_model()
        fields = ("id", "username",)
```

books/serializers.py:

```
# books/serializers.py
from rest_framework import serializers
from .models import Book


class BookSerializer(serializers.ModelSerializer):
    class Meta:
        fields = (
            "id",
            "author",
            "title",
            "created_at",
        )
        model = Book
```

Modification des urls

```
$ touch accounts/urls.py
```

`accounts/urls.py` :

```
from django.urls import path
from rest_framework.routers import SimpleRouter

from .views import UserViewSet

router = SimpleRouter()
router.register("users", UserViewSet, basename="users")

urlpatterns = router.urls
```

`books/urls.py`:

```
# posts/urls.py
from django.urls import path
from rest_framework.routers import SimpleRouter

from .views import BookViewSet

router = SimpleRouter()
router.register("books", BookViewSet, basename="books")

urlpatterns = router.urls
```

`tutorial_project/urls.py` :

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/v1/", include("books.urls")),
    path("api/v1/", include("accounts.urls")),
]
```

Excellent ! Le code fonctionne comme précédemment mais il est un peu plus propre.

Permissions

Project-Level Permissions

Pour le moment, tout le monde a accès à notre API. En effet, dans `tutorial_project/settings.py`, les permissions sont accordées à tout le monde via `AllowAny` :

```
REST_FRAMEWORK = {
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.AllowAny",
    ],
}
```

Modifions `AllowAny` en `IsAuthenticated` :

```
REST_FRAMEWORK = {
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticated",  # new
    ],
}
```

Maintenant seuls les utilisateurs authentifiés ont accès à l'application. Si on va à l'adresse http://127.0.0.1:8000/api/v1/users/, l'API nous renvoie un message d'erreur 403 ! Par-contre, authentifions-nous à l'adresse http://127.0.0.1:8000/admin/ et retournons à http://127.0.0.1:8000/api/v1/users/. Maintenant l'accès est autorisé 😀

Créons un utilisateur régulier *testuser* avec pour mot de passe *AppleAreRed* afin de tester notre installation. Pour ajouter cet utilisateur, il suffit de se rendre à http://127.0.0.1:8000/admin/accounts/customuser/add/, de remplir les champs du nom d'utilisateur et du mot de passe et enfin de sauvegarder.

Deconnectons-nous afin de faire des tests.

Dans le fichier `tutorial_project/urls.py`, ajoutons la ligne `path("api-auth/", include("rest_framework.urls")), # new` :

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/v1/", include("books.urls")),
    path("api/v1/", include("accounts.urls")),
    path("api-auth/", include("rest_framework.urls")),  # new
]
```

Cette ligne permet d'ajouter un petit bouton **Log-In** et **Log-Out** sur les pages de l'API.

View-Level Permissions

Les permissions au niveau des vues permettent de restreindre l'accès aux endpoints à certains utilisateurs.

Créons un fichier `books/permissions.py` qui contiendra les permissions d'accès pour les livres :

```
from rest_framework import permissions

class IsAuthorOrReadOnly(permissions.BasePermission):
    def has_permission(self, request, view):
        # Authenticated users only can see list view
        if request.user.is_authenticated:
            return True
        return False

    def has_object_permission(self, request, view, obj):
        # Read permissions are allowed to any request so we'll always #
allow GET, HEAD, or OPTIONS requests
        if request.method in permissions.SAFE_METHODS:
            return True
            # Write permissions are only allowed to the author of a post
        return obj.author == request.user
```

Dans le fichier `books/views.py`, il est maintenant possible d'ajouter des permissions spécifiques à l'accès des livres :

```
# posts/views.py
from rest_framework import viewsets
from .permissions import IsAuthorOrReadOnly

from .models import Book
from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):
    permission_classes = (IsAuthorOrReadOnly,)
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

L'accès aux endpoints des utilisateurs est plus restreint puisque seul l'administrateur y a accès `accounts/views.py` :

```
from django.contrib.auth import get_user_model
from rest_framework import viewsets
from rest_framework.permissions import IsAdminUser # new
from .serializers import UserSerializer

class UserViewSet(viewsets.ModelViewSet):
    permission_classes = [IsAdminUser] # new
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer
```

Super ! L'utilisateur régulier *testuser* peut lire la liste des livres, ajouter/modifier/supprimer des livres dans sa propre liste de livres mais il ne peut que lire les livres créés par d'autres utilisateurs.

Il persiste quelques défaults. Par-exemple, *testuser* peut créer des livres à la place de l'administrateur. Nous modifierons ce comportement par la suite.

Authentification

C'est parti : implémentons une authentification par token !

`tutorial_project/settings.py` :

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # 3rd-party apps
    "rest_framework",
    "corsheaders",
    "rest_framework.authtoken",  # new
    # Local
    "accounts.apps.AccountsConfig",
    "books.apps.BooksConfig",
]

REST_FRAMEWORK = {
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticated",
        ],
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework.authentication.SessionAuthentication",
        "rest_framework.authentication.TokenAuthentication",  # new
    ],
}
```

Pas besoin d'installer la librairie via pip car elle fait partie des librairies par défaut de Django. Par contre, nous devons quand même prévenir Django qu'elle sera utilisée via la ligne `"rest_framework.authtoken", # new`. Profitons-en pour effectuer une migration comme à chaque modification du fichier `settings.py` :

```
$ ./manage.py makemigrations
No changes detected

$ ./manage.py migrate
Operations to perform:
  Apply all migrations: accounts, admin, auth, authtoken, books,
contenttypes, sessions
Running migrations:
  Applying authtoken.0001_initial... OK
```

```
  Applying authtoken.0002_auto_20160226_1747... OK
  Applying authtoken.0003_tokenproxy... OK
```

Super 😉 Si nous nous rendons à la page http://127.0.0.1:8000/admin/authtoken/tokenproxy/, une liste de tokens devrait être affichée ! Pour le moment, cette liste est vide malgré nos deux utilisateurs (admin et testuser). C'est normal, un token de connexion sera généré après la première requête vers l'API.

Endpoints

Nous allons avoir besoin d'endpoints afin d'effectuer des requêtes de connexion, déconnexion, mettre à jour un mot de passe, ... Ces endpoints seront créés en combinant deux librairies **dj-rest-auth** et **django-allauth**. Django-allauth est LA librairie d'authentification de Django. Elle souffre cependant d'une lacune majeure : les endpoints ne sont pas disponibles pour des requêtes faites depuis une origine différente de Django. Comme nous allons utiliser React Native pour effectuer les requêtes, nous devons ajouter la librairie dj-rest-auth qui permet d'ajouter des endpoints accessibles depuis une SPA 😃

dj-rest-auth

dj-rest-auth nous sera utile pour ajouter les endpoints de log-in, log-out et de réinitialisation du mot de passe. Commençons par installer la librairie :

```
$ pip install dj-rest-auth
```

Ensuite, ajoutons la librairie dans `settings.py` :

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # 3rd-party apps
    "rest_framework",
    "corsheaders",
    "rest_framework.authtoken",
    "dj_rest_auth",  # new
    # Local
    "accounts.apps.AccountsConfig",
    "books.apps.BooksConfig",
]
```

Enfin, modifions `tutorial_project/urls.py` :

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/v1/", include("books.urls")),
    path("api/v1/", include("accounts.urls")),
    path("api-auth/", include("rest_framework.urls")),
    path("api/v1/dj-rest-auth/", include("dj_rest_auth.urls")),  # new
]
```

Et voilà 😃 C'est aussi simple que ça ! Nous avons maintenant de nouveaux endpoints :

- http://127.0.0.1:8000/api/v1/dj-rest-auth/login/ : log-in de l'utilisateur
- http://127.0.0.1:8000/api/v1/dj-rest-auth/logout/ : log-out de l'utilisateur
- http://127.0.0.1:8000/api/v1/dj-rest-auth/password/reset/ : réinitialiser le mot de passe de l'utilisateur

django-allauth

Il ne manque qu'un endpoint pour l'inscription d'un nouvel utilisateur. Nous allons utiliser django-allauth qui permet l'inscription classique via email mais aussi via réseaux sociaux comme FaceBook ou Twitter.

```
$ pip install django-allauth
```

Modifions `settings.py` :

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # 3rd-party apps
    "rest_framework",
    "corsheaders",
    "rest_framework.authtoken",
    "allauth",  # new
    "allauth.account",  # new
    "allauth.socialaccount",  # new
    "dj_rest_auth",
    "dj_rest_auth.registration",  # new
    # Local
    "accounts.apps.AccountsConfig",
    "books.apps.BooksConfig",
]
```

ainsi que

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
                "django.template.context_processors.request",  # new
            ],
        },
    },
]

EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"  # new

SITE_ID = 1  # new
```

Par défaut, django-allauth utilise la variable `EMAIL_BACKEND` pour la confirmation de l'adresse email d'un nouvel utilisateur. Plutôt que d'envoyer un email de confirmation, nous allons afficher un message de confirmation dans la console.

Effectuons les migrations :

```
$ python manage.py migrate
```

et ajoutons une nouvelle URL dans `urls.py` :

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/v1/", include("books.urls")),
    path("api/v1/", include("accounts.urls")),
    path("api-auth/", include("rest_framework.urls")),
    path("api/v1/dj-rest-auth/", include("dj_rest_auth.urls")),
    path("api/v1/dj-rest-auth/registration/",
include("dj_rest_auth.registration.urls")),   # new
]
```

Excellent 😃 Créons un nouvel utilisateur pour être certain que tout fonctionne ! L'URL http://127.0.0.1:8000/api/v1/dj-rest-auth/registration/ permet d'enregistrer un nouvel utilisateur. Créons-en un avec comme :

- username : tigrou
- email : tigrou@email.com
- password : AppleAreRed

La réponse est du serveur est :

```
[27/Dec/2022 12:48:47] "GET /api/v1/dj-rest-auth/registration/ HTTP/1.1"
405 8209
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: [127.0.0.1:8000] Please Confirm Your E-mail Address
From: webmaster@localhost
To: tigrou@email.com
Date: Tue, 27 Dec 2022 12:51:43 -0000
Message-ID: <167214550307.15895.7235737074843607062@MacBook-Pro.local>

Hello from 127.0.0.1:8000!

You're receiving this e-mail because user tigrou has given your e-mail
address to register an account on 127.0.0.1:8000.

To confirm this is correct, go to http://127.0.0.1:8000/api/v1/dj-rest-
auth/registration/account-confirm-email/MQ:1pA9Ql:jW9uxq-
K6vKf8GX0RY_nZ4SgShiZ2aDNgL03nahI-uI/

Thank you for using 127.0.0.1:8000!
127.0.0.1:8000
-------------------------------------------------------------------------
-----
[27/Dec/2022 12:51:43] "POST /api/v1/dj-rest-auth/registration/ HTTP/1.1"
201 7501
```

et la réponse de l'API :

```
HTTP 201 Created
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "key": "444643f6a7921956a458567f8ec733120f640402"
}
```

Afin de voir le token, il suffit de se rendre à l'adresse http://127.0.0.1:8000/admin/ comme administrateur. Tadam !! http://127.0.0.1:8000/admin/authtoken/tokenproxy/ contient un token pour *tigrou*. Ce token devra être stocké par le front-end et renvoyé à chaque requête vers l'API 😃

Nous pouvons néanmoins utiliser *curl* pour visualiser l'apparence de notre requête. Précédemment la requête était :

```
$ curl -X GET http://127.0.0.1:8000/api/v1/books/
{"detail":"Authentication credentials were not provided."}%
```

```
$ curl -H "Authorization: Token 444643f6a7921956a458567f8ec733120f640402"
-X GET http://127.0.0.1:8000/api/v1/books/

[{"id":110,"author":1,"title":"Blabla","created_at":"2022-12-
26T15:24:02.915728Z"},
{"id":111,"author":1,"title":"Blabla","created_at":"2022-12-
26T15:24:15.337840Z"},
{"id":112,"author":2,"title":"Okay","created_at":"2022-12-
26T15:24:27.083463Z"},
{"id":113,"author":1,"title":"Check","created_at":"2022-12-
26T15:27:15.306440Z"},{"id":114,"author":3,"title":"Nouveau
Titre","created_at":"2022-12-27T13:21:14.149753Z"}]%
```

Parfait 😄 Passons maintenant à React Native !

React Native : Permissions et Authentification

Commençons par un petit test : actuellement, la liste des livres n'est plus affichée. C'est normal : les endpoints ne sont pas disponibles à moins d'être authentifié. Ajoutons donc le token de l'utilisateur tigrou aux **headers** des requêtes. Dans le fichier `ssrc/features/api/bookSlice.js`, ajoutons le `prepareHeaders` :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({
    baseUrl: 'http://localhost:8000/api/v1/', // 192.168.1.20 et
10.0.2.2:8000
    prepareHeaders: (headers, { getState }) => {

      //const token = getState().auth.token
      const token = '444643f6a7921956a458567f8ec733120f640402'

      if (token) {
        headers.set('authorization', `Token ${token}`)
      } else {
```

```
        headers.set('authorization', `Token
444643f6a7921956a458567f8ec733120f640402`)
      }

      return headers
    },
   }),
  tagTypes: ['Book'],
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
      providesTags: ['Book']
    }),
    addNewBook: builder.mutation({
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
        body: initialBook
      }),
      invalidatesTags: ['Book']
    }),
    deleteBook: builder.mutation({
      query: (id) => ({
        url: `/books/${id}/`,
        method: 'DELETE',
      }),
      invalidatesTags: ['Book'],
    }),
    updateBook: builder.mutation({
      query(data) {
        const { id, ...body } = data
        return {
          url: `books/${id}/`,
          method: 'PUT',
          body,
        }
      },
      invalidatesTags: ['Book'],
    }),
  })
})

export const { useGetListOfBooksQuery, useAddNewBookMutation,
useDeleteBookMutation, useUpdateBookMutation } = bookApi
```

Parfait, la liste des livres s'affichent à nouveau. Nous pouvons également créer et modifier les livres appartenant à *tigrou* mais plus aux autres utilisateurs.

Expo-secure-store

Seulement, écrire en clair le token d'un utilisateur n'est pas recommandé au niveau de la sécurité. A la place, nous pouvons utiliser **expo-secure-store**. Le secure-store d'Expo encrypte et stocke des paires de

valeurs (clé - valeur) sur la machine de l'utilisateur. Pour l'installer, lançons la commande :

```
$ npx expo install expo-secure-store
```

Bien, nous allons modifier le bouton "Sign-Up" pour qu'il enregistre le token dans le secure-store. Pour le moment, le token sera passé en clair au secure-store. Un peu plus tard, il sera directement enregistré depuis la réponse de l'API. Modifions le fichier `SignUp.js` en y ajoutant la fonction `save` :

```
import React from 'react';
import { Text, View, Button, StyleSheet, TextInput } from 'react-native';
import { Formik } from 'formik';
import * as Yup from 'yup';
import { useNavigation } from '@react-navigation/native';
import * as SecureStore from 'expo-secure-store';

const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 10,
        marginTop: 20,
    },
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 12,
        textAlign: 'center',
        fontSize: 18,
    },
    inputLablel: {
        paddingTop: 10,
        fontSize: 18,
        height: 44,
        fontWeight: "bold",
    }
});


function SignUp() {

    const navigation = useNavigation();

    async function save(key, value, navigation) {
        await SecureStore.setItemAsync(key, value);
        navigation.navigate('Home')
    }

    const SignUpForm = props => (
        <Formik
            initialValues={{
```

```
                username: "Pierre",
                email: "pierre@email.com",
                password: "password"
            }}
            onSubmit={() => save('token',
'444643f6a7921956a458567f8ec733120f640402', navigation)}
            validationSchema={Yup.object({
                username: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required'),
                password: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required')
            })}
        >
            {({ handleChange,
                handleBlur,
                handleSubmit,
                values,
                errors,
                touched,
                isValid, }) => (
                <View>
                    <Text style={styles.inputLablel}>Username :</Text>
                    <TextInput
                        name="username"
                        placeholder='username'
                        onChangeText={handleChange('username')}
                        onBlur={handleBlur('username')}
                        value={values.username}
                        style={styles.inputStyle}
                    />
                    {touched.username && errors.username &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.username}</Text>
                    }
                    <Text style={styles.inputLablel}>Password :</Text>
                    <TextInput
                        name="password"
                        placeholder='password'
                        onChangeText={handleChange('password')}
                        onBlur={handleBlur('password')}
                        value={values.password}
                        style={styles.inputStyle}
                    />
                    {touched.password && errors.password &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.password}</Text>
                    }
                    <View style={{ paddingTop: 20 }}>
                    <Button onPress={handleSubmit} title="Sign Up"
color="#6495ed"/>
```

```
                    </View>
                </View>
            )}
        </Formik>
    );

    return (
        <View style={styles.container}>
            <SignUpForm />
        </View>
    )
}

export default SignUp;
```

La fonction `save` ajoute le token dans le secure store à la clé "token". Bien maintenant, modifions `bookSlice.js` pour que l'en-tête soit complétée avec le token extrait du secure-store d'Expo :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import * as SecureStore from 'expo-secure-store';

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({
    baseUrl: 'http://localhost:8000/api/v1/', // 192.168.1.20 et
10.0.2.2:8000
    prepareHeaders: async (headers) => {
      const token = await SecureStore.getItemAsync('token');
      if (token) {
        headers.set('authorization', `Token ${token}`)
      } else {
        console.log("mince, petite erreur !")
      }
      return headers
    },
   }),
  tagTypes: ['Book'],
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
      providesTags: ['Book']
    }),
    addNewBook: builder.mutation({
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
        body: initialBook
      }),
      invalidatesTags: ['Book']
    }),
    deleteBook: builder.mutation({
```

```
      query: (id) => ({
        url: `/books/${id}/`,
        method: 'DELETE',
      }),
      invalidatesTags: ['Book'],
    }),
    updateBook: builder.mutation({
      query(data) {
        const { id, ...body } = data
        return {
          url: `books/${id}/`,
          method: 'PUT',
          body,
        }
      },
      invalidatesTags: ['Book'],
    }),
  })
})

export const { useGetListOfBooksQuery, useAddNewBookMutation,
useDeleteBookMutation, useUpdateBookMutation } = bookApi
```

Sign-Up, Sign-In et Log-Out

Excellent ! Le token est stocké dans secure-store 😃 Attaquons-nous au problème du token écrit en claire. L'url http://127.0.0.1:8000/api/v1/dj-rest-auth/login/ renvoie le token de l'utilisateur lorsque ce dernier se connecte via mot de passe. Ajoutons cette url à la liste des endpoints de `bookSlice.js` :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import * as SecureStore from 'expo-secure-store';

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({
    baseUrl: 'http://localhost:8000/api/v1/', // 192.168.1.20 et
10.0.2.2:8000
    prepareHeaders: async (headers) => {
      const token = await SecureStore.getItemAsync('token');
      if (token) {
        headers.set('authorization', `Token ${token}`)
      } else {
        alert("mince, petite erreur !")
      }
      return headers
    },
   }),
  tagTypes: ['Book'],
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
```

```
      providesTags: ['Book']
    }),
    addNewBook: builder.mutation({
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
        body: initialBook
      }),
      invalidatesTags: ['Book']
    }),
    deleteBook: builder.mutation({
      query: (id) => ({
        url: `/books/${id}/`,
        method: 'DELETE',
      }),
      invalidatesTags: ['Book'],
    }),
    updateBook: builder.mutation({
      query(data) {
        const { id, ...body } = data
        return {
          url: `books/${id}/`,
          method: 'PUT',
          body,
        }
      },
      invalidatesTags: ['Book'], // ne recharger que le livre modifié
    }),
    logIn: builder.mutation ({
      query(creditentials) {
        return {
          url: `dj-rest-auth/login/`,
          method: 'POST',
          body: creditentials,
        }
      },
      transformResponse: async (response, meta, arg) => {
        await SecureStore.setItemAsync('token', response.key);
      },
    }),
  })
})

export const { useGetListOfBooksQuery, useAddNewBookMutation,
  useDeleteBookMutation, useUpdateBookMutation,
  useLogInMutation } = bookApi
```

Ensuite dans le fichier `SignUp.js` :

```
import React from 'react';
import { Text, View, Button, StyleSheet, TextInput } from 'react-native';
import { Formik } from 'formik';
```

```
import * as Yup from 'yup';
import { useNavigation } from '@react-navigation/native';
import { useLogInMutation } from '../api/bookSlice'


const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 10,
        marginTop: 20,
    },
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 12,
        textAlign: 'center',
        fontSize: 18,
    },
    inputLablel: {
        paddingTop: 10,
        fontSize: 18,
        height: 44,
        fontWeight: "bold",
    }
});


function SignUp() {

    const navigation = useNavigation();
    const [logIn, { isLoading }] = useLogInMutation() // ajouter error

    function save(values, navigation) {
        logIn({'username': 'tigrou', 'password': values.password})
        .unwrap()
        .then(() => {
            console.log('fulfilled')
            navigation.navigate('Home')
        })
        .catch((error) => {
            console.log('oh nooooo !!! rejected', error.status,
error.data, error.message)})
    }

    const SignUpForm = props => (
        <Formik
            initialValues={{
                username: "Pierre",
                email: "pierre@email.com",
                password: "password"
            }}
            onSubmit={values => save(values, navigation)}
            validationSchema={Yup.object({
```

```
                username: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required'),
                password: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required')
            })}
        >
            {({ handleChange,
                handleBlur,
                handleSubmit,
                values,
                errors,
                touched,
                isValid, }) => (
                <View>
                    <Text style={styles.inputLablel}>Username :</Text>
                    <TextInput
                        name="username"
                        placeholder='username'
                        onChangeText={handleChange('username')}
                        onBlur={handleBlur('username')}
                        value={values.username}
                        style={styles.inputStyle}
                    />
                    {touched.username && errors.username &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.username}</Text>
                    }
                    <Text style={styles.inputLablel}>Password :</Text>
                    <TextInput
                        name="password"
                        placeholder='password'
                        onChangeText={handleChange('password')}
                        onBlur={handleBlur('password')}
                        value={values.password}
                        style={styles.inputStyle}
                    />
                    {touched.password && errors.password &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.password}</Text>
                    }
                    <View style={{ paddingTop: 20 }}>
                    <Button onPress={handleSubmit} title="Sign Up"
color="#6495ed"/>
                    </View>
                </View>
            )}
        </Formik>
    );

    return (
```

```
        <View style={styles.container}>
            <SignUpForm />
        </View>
    )
}

export default SignUp;
```

et aussi `SignUpScreen.js` :

```
import SignUp from '../features/authentification/SignUp';
import { SafeAreaView } from 'react-native-safe-area-context';

const SignUpScreen = () => {
  return (
    <SafeAreaView style={{ flex: 1, alignItems: 'center', justifyContent:
'center' }}>
      <SignUp />
    </SafeAreaView>
  )
}

export default SignUpScreen;
```

Pour que tout fonctionne, il faut enlever une ligne `"rest_framework.authentication.SessionAuthentication"` dans `settings.py` :

```
REST_FRAMEWORK = {
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticated",
        ],
    "DEFAULT_AUTHENTICATION_CLASSES": [
        #"rest_framework.authentication.SessionAuthentication",
        "rest_framework.authentication.TokenAuthentication",  # new
    ],
}
```

Excellent ! Maintenant si on remplit le mot de passe avec **AppleAreRed**, le token sera stocké dans le secure store de Expo 😃

Nous pouvons faire pareil avec l'enregistrement de nouvel utilisateur. C'est parti, modifions `SignIn.js` :

```
import React from 'react';
import { Text, View, Button, StyleSheet, TextInput } from 'react-native';
import { Formik } from 'formik';
import * as Yup from 'yup';
import { useNavigation } from '@react-navigation/native';
```

```
import { useRegistrationMutation } from '../api/bookSlice'


const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 10,
        marginTop: 20,
    },
    inputStyle: {
        borderWidth: 1,
        borderColor: '#4e4e4e',
        padding: 12,
        marginBottom: 12,
        textAlign: 'center',
        fontSize: 18,
    },
    inputLablel: {
        paddingTop: 10,
        fontSize: 18,
        height: 44,
        fontWeight: "bold",
    }
});


function SignIn() {

    const navigation = useNavigation();

    const [Registration, { isLoading }] = useRegistrationMutation()

    function save(values, navigation) {
        Registration({'username': values.username, 'email': values.email,
'password1': values.password, 'password2': values.password})
        .unwrap()
        .then(() => {
            console.log('fulfilled')
            navigation.navigate('Home')
        })
        .catch((error) => {
            console.log('oh nooooo !!! rejected', error.status,
error.data, error.message)})
    }

    const SignInForm = props => (
        <Formik
            initialValues={{
                username: "Pierre",
                email: "pierre@email.com",
                password: "password"
            }}
            onSubmit={values => save(values, navigation)}
            validationSchema={Yup.object({
```

```
                username: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required'),
                email: Yup
                    .string()
                    .email("email is not valid")
                    .required('Required'),
                password: Yup
                    .string()
                    .min(3, 'Must be 3 characters or less')
                    .required('Required')
            })}
        >
            {({ handleChange,
                handleBlur,
                handleSubmit,
                values,
                errors,
                touched,
                isValid, }) => (
                <View>
                    <Text style={styles.inputLablel}>Username :</Text>
                    <TextInput
                        name="username"
                        placeholder='username'
                        onChangeText={handleChange('username')}
                        onBlur={handleBlur('username')}
                        value={values.username}
                        style={styles.inputStyle}
                    />
                    {touched.username && errors.username &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.username}</Text>
                    }
                    <Text style={styles.inputLablel}>Email :</Text>
                    <TextInput
                        name="email"
                        placeholder='email'
                        onChangeText={handleChange('email')}
                        onBlur={handleBlur('email')}
                        value={values.email}
                        style={styles.inputStyle}
                    />
                    {touched.email && errors.email &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.email}</Text>
                    }
                    <Text style={styles.inputLablel}>Password :</Text>
                    <TextInput
                        name="password"
                        placeholder='password'
                        onChangeText={handleChange('password')}
                        onBlur={handleBlur('password')}
```

```
                        value={values.password}
                        style={styles.inputStyle}
                    />
                    {touched.password && errors.password &&
                        <Text style={{ fontSize: 16, color: '#FF0D10' }}>
{errors.password}</Text>
                    }
                    <View style={{ paddingTop: 20 }}>
                    <Button onPress={handleSubmit} title="Sign In"
color="#6495ed"/>
                    </View>
                </View>
            )}
        </Formik>
    );

    return (
        <View style={styles.container}>
            <SignInForm />
        </View>
    )
}

export default SignIn;
```

Ensuite, modifions `SignInScreen.js` :

```
import SignIn from '../features/authentification/SignIn';
import { SafeAreaView } from 'react-native-safe-area-context';

const SignInScreen = () => {
  return (
    <SafeAreaView style={{ flex: 1, alignItems: 'center', justifyContent:
'center' }}>
      <SignIn />
    </SafeAreaView>
  )
}

export default SignInScreen;
```

Et enfin `bookSlice.js` :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import * as SecureStore from 'expo-secure-store';

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({
```

```
    baseUrl: 'http://localhost:8000/api/v1/', // 192.168.1.20 et
10.0.2.2:8000
    prepareHeaders: async (headers) => {
      const token = await SecureStore.getItemAsync('token');
      if (token) {
        headers.set('authorization', `Token ${token}`)
      } else {
        alert("mince, petite erreur !")
      }
      return headers
    },
   }),
  tagTypes: ['Book'],
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
      providesTags: ['Book']
    }),
    addNewBook: builder.mutation({
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
        body: initialBook
      }),
      invalidatesTags: ['Book']
    }),
    deleteBook: builder.mutation({
      query: (id) => ({
        url: `/books/${id}/`,
        method: 'DELETE',
      }),
      invalidatesTags: ['Book'],
    }),
    updateBook: builder.mutation({
      query(data) {
        const { id, ...body } = data
        return {
          url: `books/${id}/`,
          method: 'PUT',
          body,
        }
      },
      invalidatesTags: ['Book'], // ne recharger que le livre modifié
    }),
    logIn: builder.mutation ({
      query(creditentials) {
        return {
          url: `dj-rest-auth/login/`,
          method: 'POST',
          body: creditentials,
        }
      },
      transformResponse: async (response, meta, arg) => {
        await SecureStore.setItemAsync('token', response.key);
```

```
      },
    }),
    registration: builder.mutation ({
      query(creditentials) {
        return {
          url: `dj-rest-auth/registration/`,
          method: 'POST',
          body: creditentials,
        }
      },
      transformResponse: async (response, meta, arg) => {
        await SecureStore.setItemAsync('token', response.key);
      },
    }),
  })
})

export const { useGetListOfBooksQuery, useAddNewBookMutation,
  useDeleteBookMutation, useUpdateBookMutation,
  useLogInMutation, useRegistrationMutation } = bookApi
```

CQFD ! Nous pouvons enregistrer n'importe quel utilisateur 😄

Voyons maintenant comment faire pour déconnecter un utilisateur. dj-rest-auth propose le endpoint *http://127.0.0.1:8000/api/v1/dj-rest-auth/logout/*. Une méthode POST vers cette URL supprime le token de l'utilisateur qui envoie la requête de la base de données. Implémentons là dans `bookSlice.js` :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import * as SecureStore from 'expo-secure-store';

export const bookApi = createApi({
  reducerPath: 'bookApi',
  baseQuery: fetchBaseQuery({
    baseUrl: 'http://localhost:8000/api/v1/', // 192.168.1.20 et
10.0.2.2:8000
    prepareHeaders: async (headers) => {
      const token = await SecureStore.getItemAsync('token');
      if (token) {
        headers.set('authorization', `Token ${token}`)
      } else {
        console.log("mince, petite erreur !")
      }
      return headers
    },
   }),
  tagTypes: ['Book'],
  endpoints: builder => ({
    getListOfBooks: builder.query({
      query: () => `books/`,
      providesTags: ['Book']
    }),
```

```
    addNewBook: builder.mutation({
      query: initialBook => ({
        url: 'books/',
        method: 'POST',
        body: initialBook
      }),
      invalidatesTags: ['Book']
    }),
    deleteBook: builder.mutation({
      query: (id) => ({
        url: `/books/${id}/`,
        method: 'DELETE',
      }),
      invalidatesTags: ['Book'],
    }),
    updateBook: builder.mutation({
      query(data) {
        const { id, ...body } = data
        return {
          url: `books/${id}/`,
          method: 'PUT',
          body,
        }
      },
      invalidatesTags: ['Book'], // ne recharger que le livre modifié
    }),
    logIn: builder.mutation ({
      query(creditentials) {
        return {
          url: `dj-rest-auth/login/`,
          method: 'POST',
          body: creditentials,
        }
      },
      transformResponse: async (response, meta, arg) => {
        await SecureStore.setItemAsync('token', response.key);
      },
    }),
    registration: builder.mutation ({
      query(creditentials) {
        return {
          url: `dj-rest-auth/registration/`,
          method: 'POST',
          body: creditentials,
        }
      },
      transformResponse: async (response, meta, arg) => {
        await SecureStore.setItemAsync('token', response.key);
      },
    }),
    logOut: builder.mutation ({
      query(creditentials) {
        return {
          url: `dj-rest-auth/logout/`,
```

```
            method: 'POST',
            body: creditentials,
          }
        },
        transformResponse: async (response, meta, arg) => {
          await SecureStore.deleteItemAsync('token');
        },
      }),
    })
  })

  export const { useGetListOfBooksQuery, useAddNewBookMutation,
    useDeleteBookMutation, useUpdateBookMutation,
    useLogInMutation, useRegistrationMutation,
    useLogOutMutation } = bookApi
```

`await SecureStore.deleteItemAsync('token');` supprime le token de l'utilisateur du secure store si la réponse du serveur est positive. Bien maintenant implémentons la méthode dans le fichier `Welcome.js` :

```
import { Text, Button, StyleSheet, View } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';
import { useLogOutMutation } from '../api/bookSlice'

const styles = StyleSheet.create({
    container: {
        flex: 1,
        flexDirection: 'column',
        alignItems: 'center',
        justifyContent: 'center',
        padding: 10,
    },
    textStyle: {
        fontSize: 36,
        fontWeight: "bold",
        marginBottom: 10
    }
});

const Welcome = ({ navigation }) => {

    const [logOut, { isLoading }] = useLogOutMutation() // ajouter error

    function loggingOut() {
        logOut()
        .unwrap()
        .then(() => {
            alert('Log Out Okay :)')
        })
        .catch((error) => {
            alert('Log Out Failed :(', error)
```

```
        })
    }

    return (
        <SafeAreaView style={styles.container}>
            <Text style={styles.textStyle}>Welcome Screen</Text>
            <View >
                <View style={{ padding: 5 }}>
                    <Button
                        title="Go to Sign-In"
                        onPress={() => navigation.navigate('Sign-In')}
                        color="#6495ed"
                    />
                </View>
                <View style={{ padding: 5 }}>
                    <Button
                        title="Go to Sign-Up"
                        onPress={() => navigation.navigate('Sign-Up')}
                        color="#6495ed"
                    />
                </View>
                <View style={{ padding: 5 }}>
                    <Button
                        title="Log-Out"
                        onPress={() => loggingOut()}
                        color="#6495ed"
                    />
                </View>
            </View>
        </SafeAreaView>
    );
}

export default Welcome;
```

CQFD 😃 Une fois que le bouton `Log-Out` est cliqué, le token est supprimé et toutes les requêtes futures échoueront.

React Navigation : restreindre l'accès

Il reste assez peu de features à mettre en place pour que l'application ressemble à une application réelle. Il pourrait être intéressant de restreindre l'accès aux écrans **Home** et **Books** si l'utilisateur n'est pas connecté. Pour ce faire, créons un nouveau reducer Redux qui ne contiendra qu'une seule variable `isSignIn` qui vaut `true` si l'utilisateur est connecté. Le reducer sera écrit dans `src/features/api/authentificationSlice.js` :

```
import { createSlice } from '@reduxjs/toolkit'

const initialState = { isSignIn: false }
```

```
const authentificationSlice = createSlice({
  name: 'authentification',
  initialState,
  reducers: {
    signedIn: (state, action) => {
        state.isSignIn = action.payload
    }
  }
})

export const { signedIn } = authentificationSlice.actions

// selecteurs
export const selectIsSignIn = (state) => state.authentification.isSignIn;

export default authentificationSlice.reducer
```

Une fois que le reducer est créé, il suffit de l'ajouter au store :

```
import { configureStore } from '@reduxjs/toolkit';
import { bookApi } from '../features/api/bookSlice';
import authentificationReducer from
'../features/api/authentificationSlice'

export const store = configureStore({
  reducer: {
    [bookApi.reducerPath]: bookApi.reducer,
    authentification: authentificationReducer,
  },
  middleware: getDefaultMiddleware =>
    getDefaultMiddleware().concat(bookApi.middleware)
});
```

Parfait ! Un nouveau reducer devrait apparaître dans le debugger. Pour l'utiliser, nous devons légèrement modifier la fonction `save` dans les fichiers `SignIn.js` et `SignUp.js`. Si la récupération des tokens de l'utilisateur est un succès, la valeur de `isSignIn` devient `true`.

```
import React from 'react';
import { Text, View, Button, StyleSheet, TextInput } from 'react-native';
import { Formik } from 'formik';
import * as Yup from 'yup';
import { useRegistrationMutation } from '../api/bookSlice';
import { useDispatch } from 'react-redux';
import { signedIn } from '../api/authentificationSlice'

...

function SignIn() {
```

```
    const dispatch = useDispatch();
    const [Registration, { isLoading }] = useRegistrationMutation()

    function save(values) {
        Registration({'username': values.username, 'email': values.email,
'password1': values.password, 'password2': values.password})
        .unwrap()
        .then(() => {
            console.log('fulfilled')
            dispatch(signedIn(true))
        })
        .catch((error) => {
            console.log('oh nooooo !!! rejected', error.status,
error.data, error.message)
        })
    }
...
```

ainsi que le fichier `HomeScreen.js` car nous allons déplacer le bouton de *log-out* sur cet écran :

```
import { Text, Button, StyleSheet, View } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';
import { useLogOutMutation } from '../features/api/bookSlice';
import { useDispatch } from 'react-redux';
import { signedIn } from '../features/api/authentificationSlice'


const styles = StyleSheet.create({
    container: {
        flex: 1,
        alignItems: 'center',
        justifyContent: 'center',
        padding: 10,
    },
    textStyle: {
        fontSize: 36,
        fontWeight: "bold",
        marginBottom: 10
    }
});

const HomeScreen = ({ navigation }) => {

    const dispatch = useDispatch();
    const [logOut, { isLoading }] = useLogOutMutation()

    function loggingOut() {
        logOut()
            .unwrap()
            .then(() => {
                dispatch(signedIn(false))
                alert('Log Out Okay :)')
```

```
            })
            .catch((error) => {
                alert('Log Out Failed :(', error)
            })
    }

    return (
        <SafeAreaView style={styles.container}>
            <Text style={styles.textStyle}>Home Screen</Text>
            <Button
                title="Go to Book List"
                onPress={() => navigation.navigate('Books')}
                color="#6495ed"
            />
            <View style={{ padding: 5 }}>
                <Button
                    title="Log-Out"
                    onPress={() => loggingOut()}
                    color="#6495ed"
                />
            </View>
        </SafeAreaView>
    );
}

export default HomeScreen;
```

Excellent ! Du coup, le fichier `Welcome.js` devient :

```
import { Text, Button, StyleSheet, View } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';

const styles = StyleSheet.create({
    container: {
        flex: 1,
        flexDirection: 'column',
        alignItems: 'center',
        justifyContent: 'center',
        padding: 10,
    },
    textStyle: {
        fontSize: 36,
        fontWeight: "bold",
        marginBottom: 10
    }
});

const Welcome = ({ navigation }) => {

    return (
        <SafeAreaView style={styles.container}>
            <Text style={styles.textStyle}>Welcome Screen</Text>
```

```
            <View >
                <View style={{ padding: 5 }}>
                    <Button
                        title="Go to Sign-In"
                        onPress={() => navigation.navigate('Sign-In')}
                        color="#6495ed"
                    />
                </View>
                <View style={{ padding: 5 }}>
                    <Button
                        title="Go to Sign-Up"
                        onPress={() => navigation.navigate('Sign-Up')}
                        color="#6495ed"
                    />
                </View>
            </View>
        </SafeAreaView>
    );
}

export default Welcome;
```

Plus besoin de mettre un bouton de déconnexion sur cet écran car l'utilisateur ne le verra pas s'il n'est pas connecté. Bien ! Maintenant, créons un fichier `src/screens/DefaultScreen.js` dans lequel nous allons recopier la logique du fichier `App.js` :

```
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-
stack';
import { SafeAreaProvider } from 'react-native-safe-area-context';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import BookListScreen from './BookListScreen';
import HomeScreen from './HomeScreen'
import SignInScreen from './SignInScreen';
import SignUpScreen from './SignUpScreen';
import WelcomeScreen from './WelcomeScreen'
import Ionicons from '@expo/vector-icons/Ionicons'
import { useSelector } from 'react-redux';
import { selectIsSignIn } from '../features/api/authentificationSlice';

const Tab = createBottomTabNavigator();
const Stack = createNativeStackNavigator();

function WelcomeNavigation() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Welcome" component={WelcomeScreen} />
      <Stack.Screen name="Sign-In" component={SignInScreen} options={{
title: 'Sign In' }} />
      <Stack.Screen name="Sign-Up" component={SignUpScreen} options={{
title: 'Sign Up' }} />
```

```
    </Stack.Navigator>
  );
}

export default function DefaultScreen() {

  const isSignedIn = useSelector(selectIsSignIn);

  return (
      <SafeAreaProvider>
        <NavigationContainer>
          <Tab.Navigator
            screenOptions={({ route }) => ({
              tabBarIcon: ({ focused, color, size }) => {
                let iconName;
                if (route.name === 'Home') {
                  iconName = focused
                    ? 'home'
                    : 'home-outline';
                } else if (route.name === 'Books') {
                  iconName = focused
                    ? 'book'
                    : 'book-outline';
                } else if (route.name === 'Welcome Nav') {
                  iconName = focused
                    ? 'log-in'
                    : 'log-in-outline';
                }
                return <Ionicons name={iconName} size={size} color={color}
/>;
              }
            })}
          >
            {isSignedIn? (
              <>
                <Tab.Screen name="Home" component={HomeScreen} options={{
title: 'Home' }} />
                <Tab.Screen name="Books" component={BookListScreen}
options={{ title: 'Books' }} />
              </>
            ) : (
              <Tab.Screen name="Welcome Nav" component={WelcomeNavigation}
options={{ title: 'Welcome', headerShown: false }} />
            )}
          </Tab.Navigator>
        </NavigationContainer>
      </SafeAreaProvider>
  );
}
```

La seule différence est la présence de `isSignedIn` qui autorise l'accès aux écrans `Home` et `Books` si la valeur de `isSignedIn` est `true`. Enfin, il ne reste plus qu'à simplifier `App.js` :

```
import React from 'react';
import { Provider } from 'react-redux';
import { store } from './src/reducers/store';
import DefaultScreen from './src/screens/DefaultScreen';


export default function App() {

  return (
    <Provider store={store}>
      <DefaultScreen />
    </Provider>
  );
}
```

CQFD 😄 L'application fonctionne à merveille ! Les dernières manipulations sont un peu laborieuses car nous devons modifier beaucoup de fichiers mais aucune notion n'est difficile. Il suffit de créer un state global dans Redux qui contient un booléen permettant d'accéder ou non à une boucle `if` avec un opérateur ternaire.

Et voilà ! Je pense que nous pouvons nous arrêter là pour le moment. C'est du bon boulot 😃

Améliorations

Une liste des améliorations éventuelles est disponible ci-dessous :

- **React Native Paper** pour améliorer l'aspect de l'application
- Récupérer l'**ID** des utilisateurs et l'injecter dans les requêtes POST de création / modification de livres
- Ajouter plus de contrôle aux permissions d'accès sur Django Rest Framework
- ...