Marquer comme terminé

title: React Native - Séance 4

author: Cours de François Rioult francois.rioult@unicaen.fr

Navigation

Le but de cette séance est d'apprendre à concevoir un menu pour enchaîner des écrans. Traditionnellement sur mobile, une barre de navigation est disponible en bas de l'écran. L'appui sur les items de menus déclenche l'affichage de l'écran correspondant. On peut également trouver des icônes plutôt que des items textuels. Le concept est celui de `TabNavigator`.

Mise en oeuvre

Pour regrouper tous les éléments de navigation au sein d'un même composant, on crée un composant dans un dossier dédié\ : `Navigation/Navigation.js`.

```
import React from 'react'
import { View, Text } from 'react-native'
import { NavigationContainer } from '@react-navigation/native'
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs'

import TodoListsScreen from '../Screen/TodoListsScreen'
import HomeScreen from '../Screen/HomeScreen'
import SignInScreen from '../Screen/SignInScreen'
import SignOutScreen from '../Screen/SignOutScreen'

const Tab = createBottomTabNavigator()

export default function Navigation () {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        <Tab.Screen name='Home' component={HomeScreen} />
        <Tab.Screen name='TodoLists' component={TodoListsScreen} />
        <Tab.Screen name='SignOut' component={SignOutScreen} />
      </Tab.Navigator>
    </NavigationContainer>
  )
}
```

Le composant principal défini dans `App.js` fera simplement appel à la navigation\ :

```
export default function App () {
  return <Navigation />
}
```

La navigation est constituée:

- d'un container `NavigationContainer`
- d'un type de navigateur, ici `Tab.Navigator`, `Tab` ayant été initialisé comme un `createBottomTabNavigator()`
- d'écrans avec un label qui sera affiché et un composant qui sera appelé comme écran.

Lorsqu'on arrive sur un écran par le biais de la navigation, on dispose dans les *props* d'un champ `navigation` et d'un champ `route`. `navigation` permet de naviguer sur un autre écran comme suit\ :

```
export default function SignOutScreen ({ navigation, route }) {
  return <Button title='Sign me out' onPress={() => navigation.navigate('Home')} />
}
```

La `route` contient éventuellement des paramètres au moment de l'appel de l'écran\ :

?

```
// à l'appel de l'écran
<Button onPress={() => props.navigation.navigate('Details', {id: props.id})}>

// à l'arrivée sur l'écran
<Text>{ route.params.id }</Text>
```

Persistence de variables pendant la navigation

Il est utile de partager des variables entre les écrans de navigation. C'est la notion de *contexte*, qui permet de définir des variables *globales*, globales dans le sens où elles seront accessible à chaque partie de l'arborescence des composants de l'application. Ce peut être le nom de l'utilisateur authentifié (ou un jeton), le thème ou la préférence de langue.

Les contexte sont à différencier des *props*. Les *props* sont transmises d'un composant *parent* à l'*enfant*. Le contexte peut quant à lui être partagé par des composants qui ne sont pas issus d'un parent commun.

Les contextes seront définis dans un fichier spécifique: `Contexte/Context.js`. On pourra définir un contexte particulier pour chaque variable à gérer\ :

```
import React from 'react';

export const TokenContext = React.createContext();

export const UsernameContext = React.createContext();
```

Ci-dessus nous avons défini deux contextes: l'un pour le jeton d'authentification auprès de l'API, l'autre pour le nom de l'utilisateur.

Les contextes sont accompagnés de deux fonctionnalités\ :

1. le `Provider`, qui donne la valeur au contexte
2. le `Consumer`, qui permet de lire la valeur du contexte

En général, le contexte sera géré de la même façon que l'on gère un *état*\ : on y stocke une paire $(valeur, setValeur)$. C'est une base pratique que de récupérer cette paire à l'aide de `useState`.

Voici le composant principal dans `App.js`, pourvu d'un contexte et son `Provider`\ :

```
import { TokenContext, UsernameContext } from './Context/Context'

export default function App () {
  const [token, setToken] = useState(null)
  const [username, setUsername] = useState(null)

  console.log('token', token)
  return (
    <UsernameContext.Provider value={[username, setUsername]}>
      <TokenContext.Provider value={[token, setToken]}>
        <Navigation />
      </TokenContext.Provider>
    </UsernameContext.Provider>
  )
}
```

Dans un écran de la navigation, on consommera le contexte comme suit\ :

```
export default function HomeScreen () {
  return (
    <UsernameContext.Consumer>
      {([username, setUsername]) => {
        return (
          <>
            <Text>Welcome !</Text>
            <Text>You are logged as {username}</Text>
          </>
        )
      }}
    </UsernameContext.Consumer>
  )
}
```

Si besoin, il faudra emboîter plusieurs consommateurs\ :

?

```
export default function SignInScreen ({ navigation }) {
  return (
    <TokenContext.Consumer>
      {([token, setToken]) => (
        <UsernameContext.Consumer>
          {([username, setUsername]) => {
      ...
          }}
        </UsernameContext.Consumer>
      )}
    </TokenContext.Consumer>
  )
}
```

Dans les composants fonction, on pourra préférer utiliser le *hook* `useContext`. Cela permet d'alléger le code\ :

```
export default function HomeScreen () {
  const [username, setUsername] = useContext(UsernameContext)
  return (
    <>
      <Text>Welcome !</Text>
      <Text>You are logged as {username}</Text>
    </>
  )
}
```

Navigation depuis un composant

Si le composant est défini comme un écran de la navigation, il est appelé avec l'objet de navigation dans les *props*\ :

```
export default function SignInScreen ({ navigation }) {
...
// si besoin :
   navigation.navigate(...)
```

Si c'est un composant qui n'est pas un écran de navigation, il ne dispose pas de la navigaton dans ses *props*. Il faudra donc éventuellement que l'écran ancètre transmette la navigation dans les props.

Notez cependant que forcer la navigation est souvent superflu\ :

- si on modifie l'état d'une variable qui modifie la navigation, par exemple on récupère un jeton sur l'écran SignIn, ce qui bascule la navigation vers l'ensemble de pages accessibles aux utilisateurs enregistrés
- si on utilise les liens comme indiqué ci-dessous.

Imbrication de navigation

Il est tout à fait possible d'impriquer plusieurs navigation, par exemple\ :

```

function NavigationTodo () {
  return (
      <Stack.Navigator initialRouteName='List'>
        <Stack.Screen name='List' component={TaskListsScreen} />
        <Stack.Screen name='Details' component={TaskListDetailsScreen} />
      </Stack.Navigator>
  )
}

export default function App(){
  return (
    <Tab.Navigator>
      <Tab.Screen name='Home' component={HomeScreen} />
      <Tab.Screen name='TodoLists' component={NavigationTodo} />
      <Tab.Screen name='SignOut' component={SignOutScreen} />
    </Tab.Navigator>
  )
}
```

Gestion des liens

En React Native, on peut définir des liens vers des écrans et leur passer des paramètres. Des détails ici

?

```
import { Link } from '@react-navigation/native';

// ...

function Home() {
  return (
    <Link to={{ screen: 'Profile', params: { id: 'jane' } }}>
      Go to Jane's profile
    </Link>
  );
}
```

Structuration finale du code

On évitera de mélanger les composants écran avec les autres, on mettra à part la navigation et la définition du contexte\ :

```
+-- App.js
+-- components
|   \-- SignIn.js
+-- Context
|   \-- Context.js
+-- Navigation
|   \-- Navigation.js
+-- Screen
|   +-- HomeScreen.js
|   +-- SignInScreen.js
|   +-- SignOutScreen.js
|   +-- TodoLists.js
|   \-- TodoListsScreen.js
```

Liens

- https://reactnavigation.org/
- https://www.bigbinary.com/learn-react-native/react-native-using-nested-navigators
- https://medium.com/swlh/react-native-nested-stack-navigation-bad56ea33820
- https://heartbeat.comet.ml/nesting-tab-and-stack-navigators-in-react-native-and-expo-apps-cc118a141e70
- https://javascript.plainenglish.io/react-native-navigation-8ccea9e14523

Token d'authentification

Les applications modernes utilisent des API dites CRUD, qui permettent d'exposer un système d'information pour y envoyer, par exemple par HTTP, des requêtes pour créer (Create), lire (Read), modifier (Update) ou détruire (Delete) des contenus. Ces API peuvent être développées dans de nombreux langages, par exemple PHP ou Node.js pour répondre à une requête HTTP, et servent d'interface avec un *point d'accès* (*endpoint*) à la base de données.

La technologie `GraphQL`, parmi d'autres, fournit une spécification pour définir le schéma de donné et effectuer des requêtes.

Lors de la réalisation d'une API CRUD, il n'est pas toujours possible de compter sur la disponibilité d'un *coockie* d'authentification. Il faut alors se tourner vers une autre technologie : les *jetons* d'authentification.

Ces jetons sont fournis par le serveur, qui va hacher un ensemble d'informations à l'aide d'une clé secrète. L'utilisateur récupère ce jeton lors de son inscription ou de sa connexion, et doit le transmettre lors de chaque opération pour s'authentifier. Lorsque le serveur reçoit un jeton, il lui est facile de le dé-hacher pour vérifier les informations sur l'utilisateur.

Les jetons `JWT`

Nous utilisons ici une technologie particulière de jeton : `JWT` (JSON web token). Lors de la connexion (signIn), on récupère le jeton :

```
mutation {
  signIn(username: "admin", password: "rootroot")
}
```

```
{
  "data": {
    "signIn":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxZmNlODk2Yy0zNmJiLTQyZjQtYWFiMi0xNGZiMjMxZjkwMzEiLCJyb2xlcyI6WyJhZG1pbiJdLCJqdGkiOiIyODVhNjZiYS1mOGE5LTQwYTUtOWU3ZC1iMjg5ZmE4YWVjYmQiLCJpYXQiOjE2MzQ4MzA2Mjd9.3O3gDRu5FUqOKclwsSpEtWoyP0PBm3t2YbkKZ6oN5j8"
  }
}
```

Pour utiliser le jeton, il faut ajouter un entête HTTP à la requête `GraphQL` :

```
{
  "authorization": "Bearer eyJ0eXAi...j8"
}
```

?

Noter que le jeton est en trois parties, séparées par des points. Les deux premières consistent en un encodage `base64` des données et ne sont pas cryptées (il ne faut donc pas y mettre de données sensibles), la dernière est une signature par la clé, qui certifie que le jeton n'a pu être fourni que par le serveur. On peut effectuer le décodage le token à l'aide de https://jwt.io/ (secret: dFt8QaYykR6PauvxcyKVXKauxvQuWQTc)

- header (type de token + algorithme de hachage)

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

- payload (la donnée véhiculée)
  - sub (l'id du souscripteur)
  - roles
  - jti (un id tiré au hasard)
  - iat (la date à partir de laquelle le jeton est valide)
  - exp (la date jusqu'à laquelle le jeton est valide)

```
{
  "sub": "1fce896c-36bb-42f4-aab2-14fb231f9031",
  "roles": [
    "admin"
  ],
  "jti": "285a66ba-f8a9-40a5-9e7d-b289fa8aecbd",
  "iat": 1634830627
}
```

Utilisation du jeton par React

Dans une application React, le jeton sera géré par un `Context`. Si le jeton n'est pas défini, il faudra forcer l'utilisateur à se rendre sur l'écran de connexion.

Pour cela, la navigation est construite à l'aide d'un test en ligne\ :

- la navigation consomme le contexte du jeton
- si le jeton n'est pas défini, on affiche les écrans `SignIn` et `SignUp`
- sinon on affiche la navigation standard

```
export default function Navigation () {
  return (
    <TokenContext.Consumer>
      {([token, setToken]) => (
        <NavigationContainer>
          {token == null ? (
            <Tab.Navigator>
              <Tab.Screen name='SignIn' component={SignInScreen} />
              <Tab.Screen name='SignUp' component={SignUpScreen} />
            </Tab.Navigator>
          ) : (
            <Tab.Navigator>
              <Tab.Screen name='Home' component={HomeScreen} />
              <Tab.Screen name='TodoLists' component={TodoListsScreen} />
              <Tab.Screen name='SignOut' component={SignOutScreen} />
            </Tab.Navigator>
          )}
        </NavigationContainer>
      )}
    </TokenContext.Consumer>
  )
}
```

API CRUD pour fournir le jeton

La machine virtuelle `Neo4j + GraphQL` fournit l'API CRUD.

En particulier, cette API répond aux mutations suivantes, qui permettent de se connecter ou de s'enregistrer\ :

```
mutation{signIn(username:"...", password:"...")}
mutation{signUp(username:"...", password:"...")}
```

Une autre façon d'écrire les mutations est de leur attribuer des variables paramètres\ :

```
  'mutation($username:String!, $password:String!){signIn(username:$username, password:$password)}'
```

La mutation doit être transmise en POST et accompagnée de la valeur des variables. Voici comment réaliser cela en Javascript avec l'API `fetch`,

disponible en standard\ :

```
const API_URL = 'http://graphql.unicaen.fr:4000'

const SIGN_IN = `
mutation SignIn($username: String!, $password: String!) {
  signIn(username: $username, password: $password)
}
`

export function signIn(username, password) {
  return fetch(API_URL, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      query: SIGN_IN,
      variables: {
        username: username,
        password: password
      }
    })
  })
    .then(response => {
      return response.json()
    })
    .then(jsonResponse => {
      if (jsonResponse.errors != null) {
        throw jsonResponse.errors[0]
      }
      return jsonResponse.data.signIn
    })
    .catch(error => {
      throw error
    })
}
```

Il est important de noter que\ :

- on appelle `fetch` sur une URL
- avec un objet
  - un champ `method`
  - des précisions sur l'entête
  - un corps contenant
  - une requête
  - la valeur des variables
- `fetch` est une promesse qu'il faut retourner. Dans le code qui utilise la fonction `signIn`, il faudra également écrire une promesse\ :

```
    signIn(login, password)
      .then(token => {
        setToken(token)
        setUsername(login)
        props.navigate('Home')
      })
      .catch(err => {
        setError(err.message)
      })
```

- lorsqu'une promesse réussit, elle enchaîne avec le `then` qui appelle la fonction fléchée qu'il contient avec comme paramètre ce qui a été retourné par le `then` de la promesse précédente.
- lorsqu'une promesse échoue, elle enchaîne avec le `catch` qui appelle la fonction fléchée qu'il contient avec comme paramètre ce qui a été propagé (avec `throw`) par la promesse précédente.
- le premier `then` du `fetch` reçoit la réponse, qui est retournée en JSON
- le deuxième `then` reçoit donc la réponse au format JSON
- en Javascript, une erreur est un objet, que l'on construit avec `new Error(message)`. On affiche une erreur sous forme de chaîne de caractères à l'aide de `error.message`.

Modifié le: lundi 9 septembre 2024, 11:31

?

◄ TP Sudoku

Choisir un élément

Aller à...

TP 4 ►

?